Automated Experiment-Driven Management of (Database) Systems

Shivnath Babu, Nedyalko Borisov, Songyun Duan, Herodotos Herodotou, Vamsidhar Thummala
Duke University

Abstract

In this position paper, we argue that an important piece of the system administration puzzle has largely been left untouched by researchers. This piece involves mechanisms and policies to identify as well as collect *missing* instrumentation data; the missing data is essential to generate the knowledge required to address certain administrative tasks satisfactorily and efficiently. We introduce the paradigm of *experiment-driven management* which encapsulates such mechanisms and policies for a given administrative task. We outline the benefits that automated experiment-driven management brings to several long-standing problems in databases as well as other systems, and identify research challenges as well as initial solutions.

1 Introduction

The task of administering a large system continues to remain notoriously hard. There have been a number of efforts in recent years to simplify system administration (e.g., [2, 9, 10]). These efforts include system-level mechanisms like virtualization, computational frameworks like mapreduce, and tools that leverage statistical machine-learning techniques to analyze instrumentation data collected from systems. In spite of these efforts, current solutions for administrative tasks like benchmarking, tuning, troubleshooting, and capacity-planning remain far from satisfactory.

Let us begin with an example scenario. Figure 1 shows the typical installation of an enterprise database system that consists of the production database, one or more standby databases for high availability, a test database used by database administrators (DBAs) and developers, and possibly a staging database for staged updates as they are moved from development to production. Suppose the DBA notices a slowdown of the production database due to some unknown cause. The DBA may collect some monitoring data on the production database in an attempt to diagnose the problem. However, data collection can increase the load on an already under-performing database, forcing the DBA to shift to the test database. The DBA's usual course of action would be:

1. Create a replica of the production environment on the test database.
2. Get more insight into system behavior by performing runs of the production workload on the test database, and collecting instrumentation data. Multiple runs may be required because of system variability.
3. Form hypotheses regarding potential causes of the performance problem. Do further runs under different system configurations to refine or confirm these hypotheses. For example, new indexes, statistics about the data, or resources may be added; hints may be given to the database query optimizer to force it to choose specific query execution plans; database configuration parameter settings may be changed; and so on.
4. When a fix is found, possibly after much trial and error, a careful validation is done to ensure that the fix will work on the production system. Validation may require multiple runs to test correctness and stability.

Note that the above process required the DBA to do a number of *experiments*. Each experiment involved setting up the system in a desired configuration, running a specific workload, and collecting instrumentation data for analysis. Experiments were used (i) to better understand the problem, (ii) during the search process of finding the fix, and (iii) for validating that an accurate and stable fix has been found. We call the overall process an instance of *experiment-driven management*.

Are experiments really needed in the above scenario? Quoting researchers from Oracle [11]: "it is almost impossible to predict the impact of such changes on query performance before actually trying them." Here, "such changes" refer to changes to the database schema (e.g., adding or dropping indexes), updating the statistics (about the data) used by the query optimizer to pick plans, changes to database configuration parameters (e.g., buffer pool sizes), upgrades to the database software or hardware, and others.

Techniques like performance modeling and machine learning [4] applied to system instrumentation data can reduce the need for experiments. However, experiment-driven management is and will remain part and parcel of an administrator's job in the foreseeable future. There are two predominant ways in which instrumentation data is generated from systems today:

- *Preproduction testing:* Instrumentation data can be collected from runs of the system before it goes into production use, e.g., when load/stress tests are done.
- *Production-time monitoring:* Once the system is in production, a variety of products (e.g., HP OpenView, IBM Tivoli) are available for monitoring performance.

Instrumentation data collected by these methods may not be representative of the full space of system behavior. We only get to observe the performance of query plans, query mixes, and database configurations that were actually used. System performance is a complex function of a number of factors. The collected data may quickly become unrepresentative of system behavior if workload or configuration changes (e.g., the mere addition of an index can change the patterns of I/O that a database issues to the storage system).
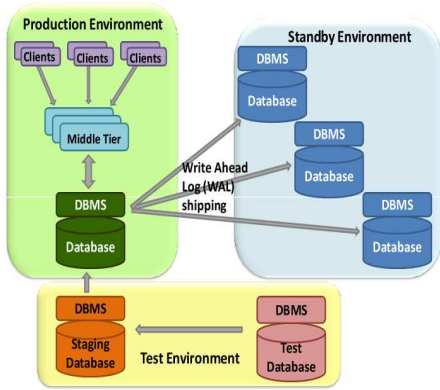
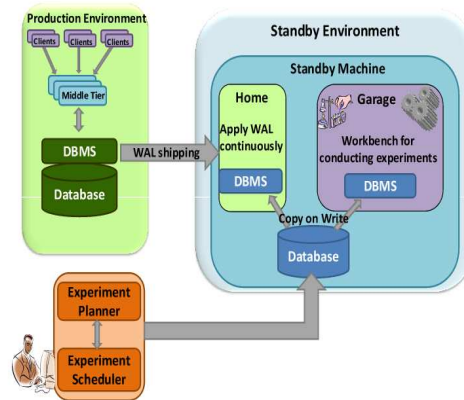Figure 1: Production Database System (DBMS) in an enterprise
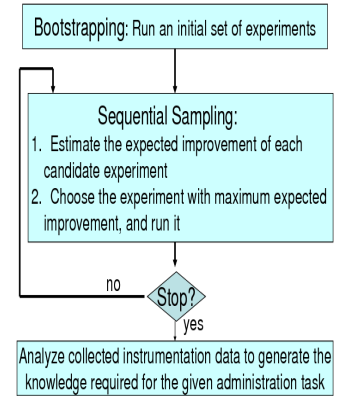

Figure 2: Workbench for experiments

Figure 3: Adaptive sampling for experiment-driven management

1.1 Contributions of this Position Paper

- We observe that experiment-driven management is an essential part of system administration. The process is labor-intensive, and hence, expensive in terms of total cost of system ownership. This domain has largely been overlooked by researchers.

- Solutions that partially or fully automate experiment-driven management have the potential to solve long-standing problems in system management. We provide examples from our own work and the work of others.

- We envision experiments being supported as first-class citizens in database and general systems. We also envision an ecosystem of tools and a well-founded methodology (ideally, a science) that guides how experiments are designed and conducted. Towards this end, we identify research challenges and design principles that address: how experiments are set up, where/when are experiments run, and which experiments are done.

- We give a case study where automated experiment-driven management is applied to tune the large number of configuration parameters in a database system.

2 Role of Experiments in System Management

We begin with a series of examples showing the critical role of experiments in many aspects of system administration.

Benchmarking: Researchers, developers, and administrators devote a lot of time and resources to running experiments as part of benchmarking. An important benchmarking activity is *response surface mapping (RSM)* [5, 7] which involves plotting system performance over a large space of workloads and/or system configurations. RSM is a powerful tool to evaluate design and cost tradeoffs, explore the interactions of workloads and system choices, and identify interesting points such as optima, crossover points, or the bounds of the effective operating range for design choices.

Figure 4 shows a response surface generated by running a TPC-H benchmark query in a PostgreSQL database for different settings of the *effective_cache_size* and *shared_buffers* parameters. The value of *effective_cache_size* is used to de-

termine the chances of a logical I/O hitting in the OS file cache; so its recommended setting is the size of the OS file cache. *shared_buffers* is the size of PostgreSQL's primary buffer cache. Generating each point in Figure 4 can involve a number of experiments; so many hours to days of effort can go into generating such surfaces. In [7], we developed an automated feedback-driven controller that designs and conducts a series of experiments to approximate a response surface efficiently and with statistical rigor.

Tuning configuration parameters: Database, application, and storage servers ship with a large number of configuration parameters like buffer cache sizes, number of I/O daemons, and parameters input to the database query optimizer's cost model. Finding good settings for these parameters is a challenging task because of the complex ways in which parameter settings can affect performance [9]. One technique used by administrators to tune configuration parameters is to conduct a series of experiments where the value of one parameter is varied at a time. Such "one-parameter-at-a-time" techniques can have undesirable consequences when significant cross-parameter interactions exist. We will revisit this problem in Section 4.

Query interactions: A query Q_1 that runs concurrently with another query Q_2 in a database system can impact Q_2's performance negatively or positively. The resource demands of Q_1 and Q_2 could interfere at physical resources like CPU, L1 or L2 cache, and I/O bandwidth, or at internal resources like latches, locks, and buffer caches. On the other hand, Q_1 may benefit Q_2 by reading data useful for Q_2 into the buffer cache. Traditionally, such interactions were not modeled due to the complexity involved. Experiment-driven modeling—where selected query mixes are scheduled, and the collected data used to train statistical models—holds promise [1], even to scheduling map-reduce and scientific jobs [6].

Testing and troubleshooting: Experiments arise naturally in problem debugging and diagnosis. Automated help-desks ask questions (e.g., did you try rebooting?) to cus-
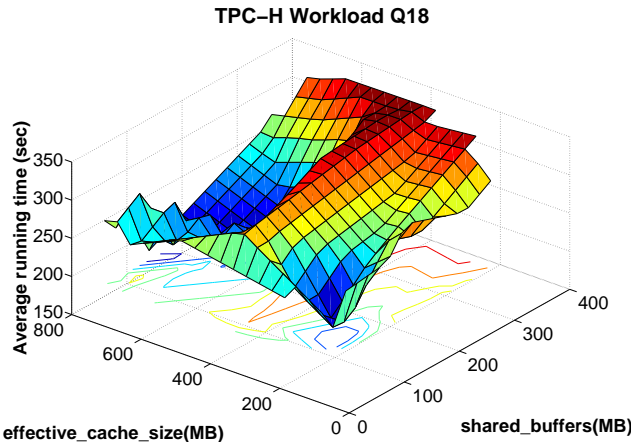
2

TPC–H Workload Q18

Figure 4: 2D projection of a response surface for TPC-H Query 18; Database size = 4GB, Memory = 1GB

tomers calling about service problems. These questions are generated dynamically based on current information. The answers may lead to more questions until there is enough information to diagnose the root cause. A similar approach has been advocated to diagnose system problems [4].

Towards the elusive self-tuning database system: When a performance problem arises in a self-tuning database system, there is a large arsenal of potential *fixes* to choose amongst—reallocating resources like CPU and memory, changing settings of configuration parameters like buffer cache sizes, changing the physical design like indexes, or running tasks like defragmentation and statistics gathering. It is nontrivial to pick the best fix, especially if the fix is some combination of the above fixes. To solve this problem, [10] argues that we need advanced mathematical or statistical models that can map the joint space of $\langle$*workload parameters*, *state from applying fixes*$\rangle$ to database performance metrics. More importantly, we need representative data to train and validate these models, and to maintain the models in the face of workload, resource, and configuration changes. Such data can only come from experiments!

3 Dissecting Experiment-driven Management

Having shown the critical role of experiments in system administration, we now attempt to break experiment-driven management down to its component tasks; which helps in identifying challenges and initial solutions. For ease of presentation and concreteness, we will focus on experiments in database systems. However, the challenges and ideas are more generally applicable.

3.1 Setting Up an Experiment

Consider the scenario from Section 1 where a DBA needs to conduct one or more experiments. Setting up these experiments is currently labor-intensive. Automating the tasks involved will be very useful, but it poses research challenges:

- *Ensuring representative workloads:* How do we find a representative workload to use in an experiment? For example, experiments for troubleshooting a deadlock may need a workload that preserves the fine-grained in-

terleaving of transactions in the production workload. However, such a workload may be invalid for experiments in configuration parameter tuning because changing parameter settings may change the interleaving.
- *Ensuring representative data:* Should experiments be run on a full copy of the production data, or would (faster) experiments on a sample suffice (and if so, can the sample be picked automatically)?

3.2 Where and When to Run Experiments?

Before the database goes into production use, experiments can be done on the production platform itself (Figure 1). If the database is already in production use and serving real users and applications, then experiments could be done on an offline test platform. These solutions are reasonable, but not sufficient because workloads may change and necessitate new experiments, or a test database platform may not exist. We will describe an initial solution that we have prototyped to address such concerns. This solution also serves to highlight the challenges that arise.

The guiding principle behind our solution is: exploit underutilized resources in the production environment for experiments, but never harm the production workload. The two salient features of our solution are:

- **Workbench:** Users designate which resources can be used for running experiments. All resources in Figure 1 are candidates; the production database is the default.
- **Policies:** A policy is specified with each resource that dictates when the resource can be used for experiments. A default policy associated with each resource in Figure 1 could be: "if the CPU, memory, and disk utilization of the resource for its *home use* is below 10% for the past 10 minutes, then the resource can be used for experiments." Home use denotes the regular (i.e., non-experimental) use of the resource.

The design of the workbench is based on splitting the functionality of each resource into two: (i) *home use*, where the resource is used directly or indirectly to support the production workload, and (ii) *garage use*, where the resource is used to run experiments. We will describe this design for standby databases, and then generalize to other resources. All commercial databases support one or more hot standby databases whose home use is to keep up to date with the (primary) production database by applying redo logs shipped from the primary. If the primary fails, a standby will quickly take over as the new primary. Hence, standby databases run the same hardware and software as the production database. Standby databases usually have very low utilization since they only have to apply redo records.

Thus, the standby databases are a valuable and underutilized asset that can be used for on-demand experiments without impacting user-facing queries. However, their home use should not be affected, i.e., the recovery time on failure should not have any noticeable increase. We achieve this property using two *resource containers*: the home container for home use, and the garage container for running

experiments. Our current implementation of resource containers uses the *zones* feature in Solaris [8] which allows resources to be allocated dynamically to a zone, with isolation between different zones. Alternately, resource containers can be implemented using virtual machine technology.

The home container is always responsible for applying the redo log records. When the standby machine is not running experiments, the home container runs on it using all available resources; the garage lies idle. The garage is booted—similar to a machine booting, but much faster—only when the policy kicks in and allows experiments to be scheduled on the standby machine. During an experiment, both the home and the garage containers will be active, with a partitioning of resources as determined by the experiment scheduler. Figure 2 provides an illustration. Both the home and the garage containers run a full and exactly the same copy of the database software. However, on booting, the garage is given a snapshot of the current data in the database. The garage's snapshot is logically separate from the home container's snapshot, but it is physically the same except for copy-on-write semantics. Our current implementation of snapshots and copy-on-write semantics leverages the *Zettabyte File System* [8], and is very efficient.

When experiments are completed or if the primary fails or there is a policy violation, the garage is halted immediately. All resources are then released to the home container continues to function as a pure standby or takes over as the primary as needed. Booting the garage (including snapshots and resource allocation) takes less than a minute; halting takes even less time. The whole process is so efficient that recovery time is not increased by more than a few seconds. While this description focused on the standby resource, the home/garage design applies to all other resources used by the workbench (including the production database).

Further details and an experimental evaluation are given in [3]. Our prototype exposes a number of challenges:

- What mechanisms should systems provide to simplify the task of running live experiments on production systems without affecting the user-facing workload?
- How can we avoid or filter out interference in measurements between the home and garage containers?
- How should policies be specified, e.g., at the resource level or through end-to-end service level agreements?
- Will the home/garage abstraction extend to experiments in large-scale systems, e.g., multi-tier services, systems distributed geographically, and network configuration?
- Can cloud computing—which provides cheap and elastic resources—be leveraged to run live experiments?

3.3 Which Experiments to Run?

Given the infrastructure for conducting experiments before or during production use, let us now consider the question of which experiments to run. Sometimes the answer is easy, e.g., when we want to find the impact of a specific change like the addition of an index. However, the space of potential experiments for various management tasks is often large. Finding the best subset of experiments to do within a limited cost or time budget is nontrivial. While some general guiding principles exist, our experience suggests that good algorithms for experiment selection can differ on a case-by-case basis. Section 4 goes into the details of experiment selection for our case study.

4 Case Study: An Advisor for Tuning Database Configuration Parameters

Our case study focuses on the problem of tuning the large number of configuration parameters like buffer pool sizes, number of I/O daemons, and parameters input to the query optimizer's cost model in database systems. Many database users face issues with the default settings, and resort to trail-and-error tuning or rules-of-thumb specified by database experts [9]. Unfortunately, the behavior of modern database systems is too complex to be captured by static rules. The following observations can be made from Figure 4:

- This surface is complex and contains nonmonotonic and unexpected behavior (performance drops sharply as shared_buffers goes above 15% of available memory).
- Rules-of-thumb settings for *shared_buffers* and *effective_cache_size* are respectively 25% and 50% of available memory. Following these rules gives around 100% worse performance compared to a well-tuned setting.
- The performance impact of changing *effective_cache_size* depends on what *shared_buffers* is set to; indicating an interaction between the two.

Given such complex behavior, experiments (which led to Figure 4) are a must for proper database tuning. Knowing the true response surface gives a lot of useful information:

Q_1: Which parameters impact performance the most?
Q_2: Which parameters display strong interactions that make "tune-one-parameter-at-a-time" efforts futile?
Q_3: What is a high-performance setting of the parameters?
Q_4: What are *robust* regions in the response surface where performance is both satisfactory and stable?

But, how can a tuning advisor generate such surfaces efficiently? Naively conducting experiments for all possible combinations of the parameters will not scale. For example, conducting all experiments for a 5-parameter space with 6 distinct settings per parameter and average run-time of 10 minutes per experiment takes 60 days!

The tuning advisor should aim to produce reasonably-accurate results for Q_1-Q_4 while running as few experiments as possible. We outline a technique, called *Adaptive Sampling*, that can achieve this goal. Adaptive Sampling, illustrated in Figure 3, consists of two phases: *bootstrapping* and *sequential sampling*. Because of space constraints, the detailed technical description and empirical evaluation (with up to 30 configuration parameters) of Adaptive Sampling are provided in [3].

Bootstrapping phase: This phase gets Adaptive Sampling started by running an initial set of experiments. A simple technique is to pick the initial experiments randomly from
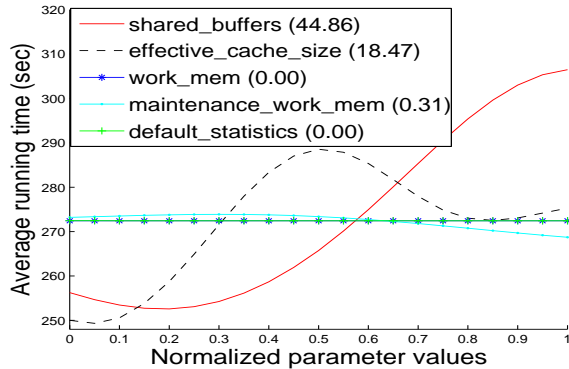
Figure 5: Sensitivity analysis plot for Figure 4

the space of possible experiments. More efficient variants include, e.g., *Latin Hypercube Sampling* [5].

Sequential sampling phase: The sequential sampling phase runs in a loop analyzing the data collected from experiments done so far (and other available data), and plans the experiments to do next. Ideally, the next experiment conducted should be the one that brings in the instrumentation data that improves the accuracy of answers to Q_1-Q_4 the most. Therefore, we need techniques to estimate the potential improvement in accuracy that results from conducting a candidate experiment. These techniques vary depending on which of Q_1-Q_4 we are interested in.

Since space constraints preclude us from giving details, we limit the discussion to some insights. For example, to address Q_1, we need experiments that check whether changing a parameter can cause significant changes in performance. However, to address Q_3, we need experiments that quickly hit high-performance regions. The promising experiments for Q_3 are from two types of regions: (i) regions around high-performing settings known so far, (*exploitation*), and (ii) regions with high uncertainty (*exploration*). For (ii), we need to capture the uncertainty (or confidence intervals) around predicted performance values in different regions. Various criteria—e.g., based on time, cost of experiments, or expected improvement—can be used to decide when to stop sequential sampling. The sequential sampling phase can also plan batches of experiments that are done in parallel (e.g., in a cloud computing platform).

We have done an extensive empirical evaluation of Adaptive Sampling using PostgreSQL and the TPC-H, TPC-W, and RUBiS benchmarks. We give a glimpse of the effectiveness of Adaptive Sampling by presenting its results on tuning PostgreSQL for TPC-H Query 18. (Results for more complex workloads are in [3].) The full response surface for this query (Figure 4) was generated using 99 experiments.

In the bootstrapping phase of Adaptive Sampling, we ran 10 experiments and estimated the individual impact (*sensitivity*) of each parameter on performance (by averaging out the effects of other parameters). Figure 5 shows the sensitivity analysis result for five (among 30) important parameters in PostgreSQL. *shared_buffers* (with an impact score of 44.8) and *effective_cache_size* are identified as the

most important parameters. Note that with only 10 experiments, Figure 5 is able to capture the non-monotonic impact of *shared_buffers* and *effective_cache_size* on performance; which is consistent with what we observe in Figure 4. The sequential sampling phase of Adaptive Sampling focused on the two important parameters identified, and conducted 10 more experiments. The third experiment done in this phase found a setting whose performance was within 95% of the best performance in Figure 4. This sample result shows how a principled approach like Adaptive Sampling can save considerable time and effort in parameter tuning.

5 Concluding Remarks

The availability of a powerful workbench for automated, online experiments enables us to rethink the implementation of several system administration tasks. Emerging mechanisms like virtualization and cloud computing provide the foundations for such a workbench. In general, our Adaptive Sampling algorithm—starting with a small bootstrap set of experiments, and then doing experiments based on estimated benefits (and costs)—applies to many tasks. However, we expect the details to differ, sometimes markedly, posing challenging research problems. In closing, we note that experiments will not fully replace current model-based management practices; rather, there is interesting synergy between them that needs to be explored.

6 Acknowledgments

References

[1] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Modeling and exploiting query interactions in database systems. In *ACM Conf. on Info. and Knowledge Management*, 2008.

[2] I. Cohen, J. Chase, et al. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of OSDI*, 2004.

[3] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. Technical report, Duke Univ., Mar. 2009.

[4] M. Goldszmidt, I. Cohen, A. Fox, and S. Zhang. Three research challenges at the intersection of machine learning, statistical induction, and systems. In *HotOS*, 2005.

[5] C. R. Hicks and K. V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 1999.

[6] P. Shivam, S. Babu, and J. Chase. Active and accelerated learning of cost models for optimizing scientific applications. In *Intl. Conf. on Very Large Databases (VLDB)*, 2006.

[7] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Tech. Conference*, 2008.

[8] *ZFS and Zones in Solaris*. www.sun.com/software/solaris.

[9] D. Sullivan. *Using Probabilistic Reasoning to Automate Software Tuning*. PhD thesis, Harvard University, 2003.

[10] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, 2002.

[11] K. Yagoub, P. Belknap, et al. Oracle's SQL Performance Analyzer. *IEEE Data Engineering Bulletin*, 31(1), 2008.