Rapid Experimentation for Testing and Tuning a Production Database Deployment*

Nedyalko Borisov
Duke University
nedyalko@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

ABSTRACT

The need to perform testing and tuning of database instances with production-like workloads (W), configurations (C), data (D), and resources (R) arises routinely. The further W, C, D, and R used in testing and tuning deviate from what is observed on the production database instance, the lower is the *trustworthiness* of the testing and tuning tasks done. For example, it is common to hear about performance degradation observed after the production database is upgraded from one software version to another. A typical cause of this problem is that the W, C, D, or R used during upgrade testing differed in some way from that on the production database. Performing testing and tuning tasks in principled and automated ways is very important, especially since—spurred by innovations in cloud computing—the number of database instances that a database administrator (DBA) has to manage is growing rapidly.

We present *Flex*, a platform for trustworthy testing and tuning of production database instances. Flex gives DBAs a high-level language, called *Slang*, to specify definitions and objectives regarding running *experiments* for testing and tuning. Flex's orchestrator schedules and runs these experiments in an automated manner that meets the DBA-specified objectives. Flex has been fully prototyped. We present results from a comprehensive empirical evaluation that reveals the effectiveness of Flex on diverse problems such as upgrade testing, near-real-time testing to detect corruption of data, and server configuration tuning. We also report on our experiences taking some of the testing and tuning software described in the literature and porting them to run on the Flex platform.

1. INTRODUCTION

It is estimated that, over the next decade, the number of servers (virtual and physical) in enterprise datacenters will grow by a factor of 10, the amount of data managed by these datacenters will grow by a factor of 50, and the number of files the datacenter has to deal with will grow by a factor of 75 [16]. Meanwhile, skilled information technology (IT) staff to manage the growing number of servers and data will increase less than 1.5 times [16].

*Supported by NSF grants 0917062 and 0964560

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

The implication of this trend is clear. The days where a database administrator (DBA) is responsible for managing one or few production database instances are numbered. In the near future, a DBA will be responsible for the administration of tens to hundreds of database instances. The DBA will need to ensure that the production databases, each serving real applications and users, are available 24x7, and that the databases perform as per specified requirements. We are already starting to see very high database-to-DBA ratios in pioneering companies like Salesforce.

High database-to-DBA ratios are only possible through extensive automation of database administration. Automation of tasks like database installation and monitoring has seen major advances. Significant progress has also been made towards automating administrative tasks like backups, failover, defragmentation, index rebuilds, and patch installation (e.g., [11, 20]). However, more challenging administrative tasks to meet database availability and performance goals are harder to automate. We will first give examples of problems that have often happened in practice, and will continue to happen unless new solutions are developed.

Upgrade problems: Consider an upgrade of the database software from one version to another. The database developers will run standard test suites to test the new version. However, when a customer upgrades her production database instance to the new version, a performance degradation is experienced. Many instances of this problem are documented for database systems such as MySQL [19], Oracle [21], PostgreSQL [22], and others. Typically, some unique characteristic of the production instance—e.g., the scale or correlations in the production data, the properties of the production server hardware or operating environment, or mix of queries in the production workload—causes the problem to manifest during production use but not during offline testing.

Tuning problems: A production database instance will need to be tuned when it is not meeting specified performance requirements. A DBA's usual course of action is to try to replicate the production environment in a test setting. The DBA would then run and monitor parts of the workload on the test database instance to recreate the problem, narrow down its possible causes, and identify a fix for the problem. A highly nontrivial next step is to estimate whether the fix will actually solve the performance problem on the production database instance; multiple trial-and-error steps may be needed. Well-meaning changes to production instances have led to performance or availability problems in the past [7].

Data integrity problems: Corruption of data is a serious problem where bits of data stored in persistent storage differ from what they are supposed to be [4]. Data corruption can lead to three undesirable outcomes: data loss, system unavailability, and incorrect results. Such problems have been caused in production database instances due to hardware problems such as errors in magnetic me-

Name	Description of database administration task in English	Task specification using w,c,r,d representation	Things to note
A/B testing	How would the current database workload have performed if index I_1 had (hypothetically) been part of the production database's configuration?	Run experiment $e=\langle w{=}W(t_{cur}),c,r{=}R(t_{cur}),d{=}D(t_{cur})\rangle$, where $c{=}C(t_{cur})\cup\{I_1\}$, and compare the observed performance with the production database's performance on its current workload $W(t_{cur})$, configuration $C(t_{cur})$, data $D(t_{cur})$, and resources $R(t_{cur})$	Tries a configuration different from the production database's current one
Tuning surface creation (Fig. 9)	How does the throughput obtained for my website's OLTP workload due to my MySQL database change as its *query_cache_size* and *key_buffer_size* parameters are varied in the ranges [0,10GB] and [0,8GB] respectively?	For $q_i{\in}$[0,10GB] & $k_j{\in}$[0,8GB], run experiment e_{ij} = $\langle w{=}W(t_{cur}),c_{ij},r{=}R(t_{cur}),d{=}D(t_{cur})\rangle$, where c_{ij} is $C(t_{cur})$ with two changes: parameter *query_cache_size* is set to q_i and *key_buffer_size* is set to k_j	Gives potential to run multiple independent experiments in parallel
Upgrade planning	Suppose my production PostgreSQL database was running the newer version 9.0 instead of the current version 8.4. Would any performance regression have been observed for the workload and data from 10.00 AM of each day of ***last*** week as well as each day of the ***coming*** week?	For time $t\in$ 24 hour increments from [6/1/2012,10AM] to [6/14/2012,10AM], run experiment e_i = $\langle w{=}W(t),c = C(t),r_i,d{=}D(t)\rangle$, where r_i has the same hardware characteristics as $R(t)$, but runs the newer database software 9.0 instead of 8.4 in order to find the performance difference	Tries a different software image on the production database's past and future states
Data integrity testing	A security patch was applied to my production Oracle DB. The patch could cause data corruption. Run Oracle's DBverify corruption detection tool on the Lineitem and Order tables once every hour. (Alert if corruption is detected)	For time t in 1 hour increments starting now, run experiment e = $\langle w,c{=}C(t),r{=}R(t),d{=}D(t)\rangle$, where w is the DBverify tool run on the Lineitem and Order tables	Runs a custom workload on an indefinite number of future states
Stress testing	If the workload on my production database goes up by 10x in the coming holiday season, then how will it perform?	Run experiment $e=\langle w,c{=}C(t_{cur}),r{=}R(t_{cur}),d{=}D(t_{cur})\rangle$, where w is $W(t_{cur})$ scaled by a factor of 10	Uses a scaled version of workload[1]

Table 1: Listing of some nontrivial uses enabled by the Flex platform. $W(t)$, $C(t)$, $D(t)$, and $R(t)$ respectively denote the respective states of the workload, configuration, data, and resources for the production database instance at time t

dia (bit rot), bit flips in CPU or RAM due to alpha particles, bugs in software or firmware, as well as mistakes by human administrators [24]. It took only one unfortunate instance of data corruption (which spread from the production instance to backups), and the consequent loss of data stored by users, to put a popular social-bookmarking site out of business [18].

The above problems indicate the need to test and tune the production database instance in an efficient and timely fashion with production-like workloads (W), configurations (C), data (D), and hardware and software resources (R). The further W, C, D, and R deviate from what is observed in the production instance, the lower is the *trustworthiness* of the testing and tuning tasks done.

At the same time, the interactions of the testing and tuning tasks with the production database instance have to be managed carefully. First, the performance overhead on the production instance must be minimal. Thus, the resources used for testing and tuning have to be well isolated from those used by the production database instance to serve real applications and users. Second, the impact of potential changes recommended for the production instance have to be verified as thoroughly as possible before they are actually made. These challenges are nontrivial partly because an automated solution is needed to handle the scale where a single DBA manages hundreds of production database instances. The *Flex* platform is designed to address these challenges.

1.1 Flex

The Flex platform enables efficient experimentation for trustworthy testing and tuning of production database instances. Figure 1 gives a high-level illustration of the core concepts that Flex is based on. These concepts will be defined precisely later in the paper.

Flex treats the production database instance as an entity that can evolve over time. As illustrated in Figure 1, this entity is represented in terms of the workload W on the database, the configuration C (such as indexes and configuration parameters) of the database, the data D in the database, and the resources R (such as server hardware and software image) used by the database. Since each of W, C, D, and R can change over time, we will use $W(t)$, $C(t)$, $D(t)$, and $R(t)$ to denote their respective point-in-time states at time t.

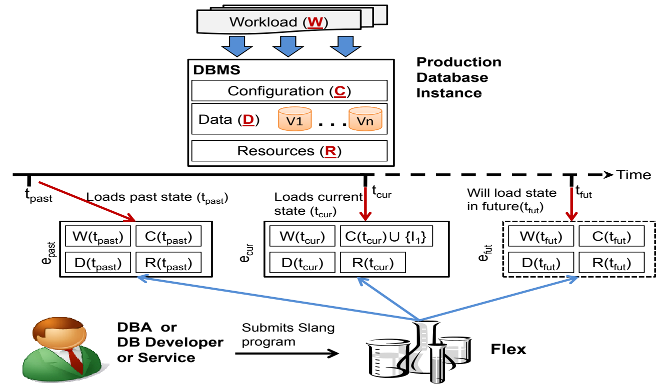


Figure 1: Overview of how Flex enables trustworthy experimentation with state (workload, configuration, and data) derived from the production database instance

For testing or tuning the production database instance, a DBA can ask Flex to run one or more *experiments*. An experiment $e = \langle w,c,r,d\rangle$ starts a database instance on resources r with configuration c and data d, and runs workload w. As illustrated in Figure 1, Flex enables w, c, r, and d to be derived directly from the state of the production database instance at a specific time in the past, current, or future; or they can come from a custom specification.

Table 1 gives a number of nontrivial tasks that can be achieved by the DBA through experiments using Flex. Flex will run each specified experiment automatically and efficiently. Flex ensures that the resources r used in any experiment are well isolated from the resources used by the production database instance. Multiple experiments can be run in parallel subject to the DBA's objectives regarding experiment completion times and number of usable hosts.

1.2 Contributions and Challenges

Slang language: The first challenge in Flex is to develop a language in which DBAs or higher-level services can express a wide spectrum of experiment-driven administrative tasks like those illus-

[1]Similar experiments can be done in Flex with scaled data sizes.

trated in Table 1. We have developed the *Slang*[2] high-level language that supports the key abstractions needed to specify such tasks easily and intuitively. In addition, Slang enables users and services to specify objectives on the number of hosts and completion times that Flex should meet while running the experiments. Sections 2 and 4 describe Slang and its abstractions.

Orchestration of experiments: Given a Slang program, Flex automates the entire *orchestration* from planning and execution of the program, to run-time monitoring and adaptation. The orchestration process has to address multiple challenges in order to run the needed experiments while meeting the objectives specified:

- How much resources to allocate for running the experiments?
- In what order to schedule the experiments on these resources?
- How to efficiently load data needed for experiments from the production database instance on to the allocated resources?

We have developed a novel technique for orchestration in Flex that uses a mix of *exploration* (to learn models to predict experiment running times) and *exploitation* (use of the learned models for planning) to meet the objectives in the Slang program. This technique is both *elastic* (it can grow and shrink its resource usage) and *adaptive* (it can react as unforeseen events such as failures arise or new information such as experiment running times is available). Flex also supports a number of different techniques to transfer evolving data from the production database instance to the resources allocated to run experiments. The details of the orchestration techniques are presented in Sections 5 and 6.

Prototype system: We have implemented the full set of language and system features in a prototype of Flex where the production database instance and experiments all run on a cloud platform such as Amazon Web Services (AWS), Rackspace, or SQL Azure. Cloud platforms are an excellent fit for Flex because these platforms are leading the massive growth in the use of compute and storage resources and the accompanying increase in database-to-DBA ratios. Furthermore, Flex is designed to take advantage of the elastic and pay-as-you-go nature of cloud platforms. The architecture and implementation of Flex are presented in Section 7.

Evaluation: Section 8 presents results from a comprehensive empirical evaluation of the Flex platform. We demonstrate how Flex simplifies and improves the effectiveness of upgrade testing, production database tuning, and data integrity testing. We have taken a testing application from the literature and ported it to run on Flex as a *Flex Service*. We dive into the details in order to demonstrate the ease of developing higher-level testing and tuning applications using Flex. We also give a comparison of this service when it runs on Flex versus when it runs as written originally.

2. ABSTRACTION OF AN EXPERIMENT

Intuitively, a replica of the production database instance has to be created in order to run an experiment. Three steps are involved in order to run an experiment:

- Identifying—possibly also having to provision dynamically—a *host* h that will provide the resources to run the experiment.[3]
- Loading a *snapshot* s of some specific state from the production database instance on to the selected host h.
- Running an *action* a associated with the experiment on the combination of the host h and snapshot s.

In this fashion, an experiment in Flex is represented as $e = \langle a,s,h \rangle$. The rest of this section will elaborate on the three steps listed above, and also clarify how the $\langle a,s,h \rangle$ representation subsumes the $\langle w,c,d,r \rangle$ representation from Section 1.

Host: In this paper, we consider database instances that run on a single (possibly multicore) server. These instances are similar to the main database server product sold by most commercial database vendors as well as popular open-source databases like MySQL and PostgreSQL. (Extending Flex to support parallel database instances is an interesting avenue for future work which is discussed in Section 10.) A host in Flex is a server that can run a database instance as part of an experiment. Flex associates two types of information with a host: the underlying server's hardware characteristics as well as the software image that runs on the server.

A host can be represented in a number of ways, and Flex can adopt any one of these. We have chosen to use a simple methodology that is motivated by how cloud providers represent hosts. The host's hardware resources are represented by a *host type*. As an example, AWS's Elastic Compute Cloud (EC2) supports seventeen different host types currently [1]. Each host type maps to a specification of resources that will be contained in any host allocated of that type. For example, AWS's default *m1.small* type is a 32-bit host with 1.7 GB memory, 1 EC2 Compute Unit, and 160 GB local storage with moderate I/O performance.

The software that runs on a host is represented by an *image*. Each software image has a unique identifier that represents the combination of software components in that image such as the OS, filesystem, and database management system. For example, an image on AWS comes with pre-configured OS and application software that is started when a host is allocated. Most database vendors provide images to run their database software on EC2 hosts.

Snapshot: A snapshot is a point-in-time representation of the entire state of a database system. A snapshot collection process that runs on the production database instance generates a snapshot whenever one is needed. This snapshot collection process—we provide details in Section 6—may also ensure that the snapshots are made persistent on a local or remote storage volume. Three types of state are captured in a snapshot:

- The actual data (D) stored in the database.
- The database configuration (C) which includes information such as indexes and materialized views, the database catalog, and server configuration parameters such as buffer pool settings.
- The database workload (W) in the form of logs such as SQL query logs and transaction logs.

Thus, a snapshot captures the data D, configuration C, and workload W in the production database instance at a point of time. The snapshot needed by one or more experiments has to be transferred to the host allocated to run these experiments. Flex manages this transfer efficiently as we will describe in Section 6.

Action: To run an experiment $e = \langle a,s,h \rangle$, Flex will load the snapshot s on to the host h, and then run the action a. The action a can be specified as a new user-defined executable or from a library of commonly-used actions. As illustrated in Table 1, the following are some commonly-used actions in experiments that Flex runs:

- Replay the workload as captured in the snapshot s [13].
- Update the existing configuration by building a new index or changing one or more server configuration parameters, and then replay the SQL query workload.
- Run a custom workload such as a corruption detection tool or a scaled version of the workload logged in the snapshot s.

3. WALK-THROUGH OF FLEX USAGE

We will begin with a walk-through of how Flex is used for a specific task by a human user such as a DBA or a database devel-

[2]The name *slang* came from Fl{*ex lang*}uage.

[3]It is inadvisable to run experiments on the production database instance since the experiments could cause unpredictable behavior.

```
Action A1 {
  execute: scripts/myisamchk1.sh
  datum: /volume/lineitem.MYI
  readOnly: true
  reboot: false
  before: tests/scripts/before_myisam.sh
  after: tests/scripts/after_myisam.sh
}

Action A2 {
  execute: tests/scripts/myisamchk2.sh
  datum: /volume/order.MYI
  readonly:true
  reboot:false
  before: tests/scripts/before_myisam.sh
  after: tests/scripts/after_myisam.sh
}

Action A3 {
  execute: tests/scripts/fsckchk.sh
  datum: /device
  readonly:true
  reboot:false
}

Credentials loginDetails {
  loginKey: ec2_key
  accessKey: dev_team
  secretKey: dev_pass
  securityGroups: ssh
}

Host testHost {
  image: ami-48aa4921
  type: m1.small
  user: root
  setup: scripts/host_setup.sh
}

Plan P {
  mapping: ( [A1,snap-1] [A2,snap-1] [A3,snap-1]
[A1,snap-2] [A2,snap-2] [A3,snap-2] [A1,snap-3]
[A2,snap-3] [A3,snap-3] )
  deadline: 90
  budget: 3
  releaseHosts: true
  reuseHosts: false
}
```

Figure 2: Example Slang Program #1 that references past snapshots (the full syntax of Slang is given in the technical report [3])

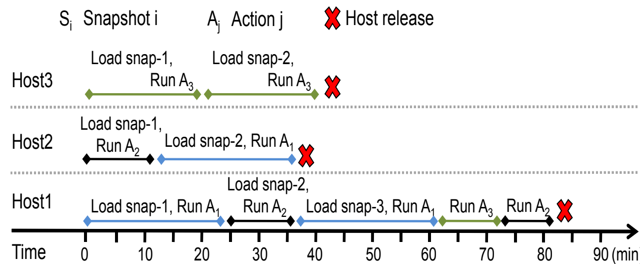


Figure 3: Execution of Example Slang Program #1 from Figure 2 (Section 8.2 gives a detailed description)

oper. (The "user" here can also be a Flex Service for automated testing or tuning.) The task is expressed in English as: *Verify the data integrity of the three collected snapshots—snap-1, snap-2, and snap-3—by running the myisamchk tool on the Lineitem and Order tables as well as the fsck tool on the file-system. Complete this task within 90 minutes. Use m1.small hosts on Amazon Web Services and do not use more than 3 hosts concurrently.*

Example Slang program #1 in Figure 2 shows how the user will express this task in the Slang language. Broadly speaking, Slang provides support for: (i) *definitions* of actions, snapshots, and hosts; (ii) *mappings* that stitch the definitions together into each one of the experiments that needs to be run; (iii) and the *objectives* that should be met while running the experiments.

Flex will parse the input Slang program and do some semantic checks. The extracted definitions, mappings, and objectives are given to Flex's *Orchestrator* which coordinates the scheduling, execution, and monitoring of experiments. Figure 3 shows the complete execution timeline of the example Slang program #1 from Figure 2. Notice from Figure 3 that the Orchestrator has to make choices regarding when to allocate and release hosts, which experiments to schedule on which hosts and when (which, in turn, determines which snapshots are loaded on which hosts), and how to deal with unpredictable events that can arise during the execution.

Three hosts—Host1, Host2, and Host3—were allocated in Figure 3. The figure shows the beginning and end times of each experiment on the respective host where the experiment was scheduled. In this schedule, snapshots snap-1 and snap-2 are loaded on all hosts, while snapshot snap-3 is loaded only on Host1. Note that, in order to minimize concurrent resource usage, while meeting the deadline, Flex released two hosts midway through the execution.

All the snapshots needed in our example task came from past states of the production database instance. Recall from Table 1 that Slang programs may also need to refer to future snapshots. These snapshots will be processed by Flex when they arrive. In this case, the orchestration behaves like the execution of continuous queries over streams of data, as we will illustrate later in the paper.

Statement	Description
Action	Definition of an action a, namely, the executable for a, the data a operates on, and (possibly) setup/cleanup scripts
Host	Host definition used to specify the resources to request and access from the resource provider
Credentials	Access definitions for Flex to use and request resources from the resource provider
Plan	Provides mapping between actions and snapshots (specifying the complete action-snapshot-resource mapping is an optional feature aimed at expert users)
Snapshot	Snapshot specification along with incoming arrival rates

Table 2: Summary of statements in the Slang language

4. SLANG LANGUAGE

All inputs to Flex are specified in Slang. It is natural to ask why a new language had to be developed for Flex. For example, couldn't Flex use an existing workflow definition language (e.g., *BPEL* [5])? To the best of our knowledge, no existing language achieves a good balance between: (i) being powerful enough to express a wide variety of experimentation needs such as the use cases in Table 1, and (ii) being simple enough for humans to use and for extracting information needed for automatic optimization. Our approach to achieve this balance was to come up with the right abstractions in Slang. These abstractions are represented by the five statements summarized in Table 2 and described next.

Action: Each unique type of action a involved in an experiment e = $\langle a,s,h \rangle$ is defined by an `Action` statement. The `execute` and `datum` clauses in the statement specify respectively the executable that needs to be run for a and the data on which a operates. The `Action` statement enables additional requirements and properties to be specified for a such as: (i) the host h needs to be restarted before a is executed; (ii) a modifies the data that it operates on; (iii) a model to estimate the expected execution time of a; and (iv) a specific host type that a should be executed on.

For example, in Figure 2, action A_1 specifies the use of the myisamchk1.sh executable (which invokes MySQL's myisamchk corruption detection tool) operating on the Lineitem table. The statement specifies that A_1 does not modify the data and that the host need not be rebooted before A_1 is invoked. The statement also specifies two executables that are respectively invoked before (e.g., for starting custom monitoring tools) and after (e.g., for some cleanup) the action's execution.

Host: Each unique type of host involved in an experiment e = $\langle a,s,h \rangle$ is defined by a `Host` statement. As discussed in Section 2, the `Host` statement specifies two identifiers: one for the host type and the other for the software image that should be started on these hosts. The identifiers in our example Slang program #1 shown in Figure 2 define a host of type m1.small on Amazon Web Services as well as a software image that contains the Linux Fedora 8 OS with the MySQL 5.2 DBMS. The `Host` statement also supports the specification of an executable to be run on host allocation (e.g., for setting up the operating environment).

Snapshot: `Snapshot` is an optional statement. Most Slang programs, including the one in Figure 2, refer to snapshots that have already been collected. An index entry appears in Flex's *History catalog* for all past snapshots. (The History Catalog is covered in Section 7.) The key for this index entry has the form snap-*id* where *id* is a positive integer identifier given by the History catalog. Slang programs refer to past snapshots using their keys. For example, see the *mapping* clause of the `Plan` statement in Figure 2.

The `Snapshot` statement is needed in a Slang program when the program has to run experiments on future snapshots, i.e., snapshots that will be collected in the future. (The Upgrade planning and Data integrity testing tasks in Table 1 have this property.) In

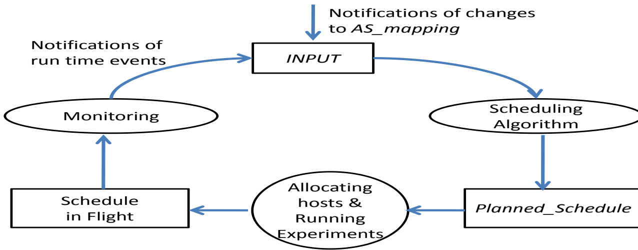


Figure 4: Steps in Flex's orchestration of experiments

this scenario, the `Snapshot` statement is provided to specify the arrival frequency and type of future snapshots. Section 6 will give more details of how snapshots are collected and the types of snapshots that Flex supports. Future snapshots are referred to in a Slang program in the form exp-*id* where id is a positive integer corresponding to the number of the snapshot received after the program is submitted.

Credentials: Flex enables a wide variety of users—e.g., DBAs, database and application developers, and automated services—to run experiments on demand for trustworthy testing and tuning of database instances. An experiment needs to allocate hosts from a resource provider as well as access specific data. Therefore, access control is a necessary feature in Flex so that users and applications cannot allocate resources or access data that their role does not permit. A user or application submitting a Slang program has to include a `Credentials` statement as shown in Figure 2 and Table 2. These credentials are used during orchestration to authenticate Flex to the resource provider and database instances so that the appropriate access controls can be enforced.

Plan: The `Action`, `Host`, `Snapshot`, and `Credentials` statements provide the definitions in a Slang program. The `Plan` statement is the crucial glue that combines the definitions together with the mappings and objectives. The mapping clause of the `Plan` statement takes one of two forms.

Mapping: The first and more common form of mapping is a set of pairs where each pair has the form $\langle a_i, s_j \rangle$ specifying that action a_i should be run on snapshot s_j. The choice of how to schedule the execution of this mapping is left to Flex. The second form of mapping is a set of triples where each triple has the form $\langle a_i, s_j, h_k \rangle$ specifying that action a_i should be run on snapshot s_j and scheduled on host h_k. The second form of mapping is provided for the benefit of higher-level applications and expert users who want fine control over the scheduling of experiments.

Objectives: The `Plan` statement contains multiple clauses for users and services to specify their objectives. The two main types are:

- *Deadline:* A soft deadline by which Flex must complete all the experiments specified in the plan.
- *Budget:* The maximum number of hosts that Flex can allocate concurrently in order to run the experiments. Recall from Sections 1 and 2 that cloud platforms are the typical resource providers for Flex.

In our example program in Figure 2, all three actions A_1, A_2 and A_3 are mapped to all three snapshots snap-1, snap-2 and snap-3. The deadline is 90 minutes and the budget is 3 hosts.

We expect the typical workload of Flex to be bursty. For example, a DBA or testing service may submit many fairly similar Slang programs in a short period of time when a system upgrade is imminent. The releaseHosts and reuseHosts clauses of the `Plan` statement enable Flex to reuse hosts across multiple plans in order to minimize the resource allocation and initialization overhead. For example, allocation of a new host from AWS usually takes a few minutes, and possibly higher under load. If the repeat clause of the `Plan` statement is set to true, then Flex will repeatedly restart the program execution after each deadline. This feature is used mainly when the program refers to future snapshots. Flex will reset the snapshot counter so that the first snapshot after the restart is exp-1, the next is exp-2, and so on. (Example Slang program #2 with this nature is given in the technical report [3] due to space constraints.)

A Slang program should contain at least one `Plan` statement. A useful feature provided by Flex for the convenience of users and services is the ability to specify multiple `Plan` statements in a Slang program.

Name	Description
AS_Mapping	Current set of $\langle a_i, s_j \rangle$ pairs for which the snapshot s_j is available (future snapshots will not be included), but the experiment for $\langle a_i, s_j \rangle$ is not yet complete
status($\langle a_i, s_j \rangle$)	One of "*WAITING*" or "*RUNNING* on host h_k" based on current scheduling status of $\langle a_i, s_j \rangle \in$ *AS_Mapping*
$h_1, \ldots, h_n$	Ordered list of currently-allocated hosts (initially empty)
end(h_k)	The later of current time (NOW) and the time when host h_k will finish its last scheduled action
Act_Time(a, *parameters*)	Model to estimate running time of action a. The default is the constant function returning ∞ when unknown
Deadline	Soft deadline to complete current experiment schedule
Budget	Maximum number of concurrent hosts to run experiments

Table 3: *INPUT*: Inputs to the orchestration process

5. ORCHESTRATION OF EXPERIMENTS

Figure 4 gives an overview of the orchestration of a Slang program. There are three phases which are all running concurrently:

1. Keeping track of the inputs to the orchestration process (denoted *INPUT*). Changes to the *INPUT* will trigger a rerun of the scheduling algorithm.
2. Running the scheduling algorithm based on the most recent *INPUT* in order to generate a *Planned_Schedule*.
3. Based on the *Planned_Schedule*, scheduling and running experiments on hosts as well as monitoring this execution in order to make any updates that are needed to the *INPUT*.

Despite a large number of seemingly related scheduling algorithms proposed in the literature, we had to design a new scheduling algorithm in Flex for the following reasons:

- Unlike many scheduling algorithms, Flex cannot assume that the running times of experiments are known beforehand, or that predetermined performance models are available to predict these running times. Flex uses a careful mix of exploration and exploitation to both learn and use such models adaptively.
- Flex cannot preempt running experiments to adapt the schedule as new information becomes available. Preemption can give incorrect results for testing and tuning.
- Unlike many algorithms from real-time scheduling, Flex cannot assume that every experiment comes with a deadline. Deadlines are optional in Flex for usability reasons.

Next, we describe the *INPUT* and Flex's scheduling algorithm. Details of run-time execution and monitoring are given in Section 7.

5.1 Inputs for Orchestration (*INPUT*)

Table 3 shows the inputs needed for orchestration, which we will denote as *INPUT*. The initial version of *INPUT* comes from the parsing of the input Slang program. Note from Table 3 that *INPUT* only includes the $\langle a_i, s_j \rangle$ pairs form of the *Plan* statement's mapping clause. If a program gives the alternate form of $\langle a_i, s_j, h_k \rangle$ triples, then the program is directly giving the *Planned_Schedule* which Flex simply has to run. *INPUT* can change in one of three

Scheduling Algo (Input=*INPUT* (Table 3), Output=*Planned_Schedule*)

```
1.  /* Total Ordering step: prioritizes earlier snapshots & longer run times */
2.  Order ⟨a_i,s_j⟩ entries in AS_Mapping so that ⟨a,s⟩ precedes ⟨a',s'⟩ if:
     (a) s is an earlier snapshot than s', OR
     (b) s=s' AND Act_Time(a, {s}) ≥ Act_Time(a', {s'})

3.  /* No Preemption step: running actions are never preempted */
4.  For each ⟨a_i,s_j⟩ ∈ AS_Mapping from first to last {
5.     if (status(⟨a_i,s_j⟩) = "RUNNING on host h_k") {
6.        Add_To_Planned_Schedule(a_i, s_j, h_k); }}

7.  /* Exploration step: collects info on actions with unknown run times */
8.  For each ⟨a_i,s_j⟩ ∈ AS_Mapping from first to last {
9.     if (status(⟨a_i,s_j⟩) = "WAITING" AND Act_Time(a_i, {s_j}) = ∞ AND
10.    ∄s: ⟨a_i,s⟩∈AS_Mapping with status(⟨a_i,s⟩) = "RUNNING on host h") {
11.       Let h_k be the first free host in h_1...h_n or a newly allocated host
          with the given Budget. Add_To_Planned_Schedule(a_i, s_j, h_k); }}

12. /* Greedy Packing step: in-order bin-packing of experiments to hosts */
13.  For each ⟨a_i,s_j⟩ ∈ AS_Mapping in the order from first to last with
       status(⟨a_i,s_j⟩) = "WAITING" {
14.    For host h_k in order from the list of hosts h_1 to h_n {
15.       if (h_k is a free host) {
16.          Add_To_Planned_Schedule(a_i, s_j, h_k); CONTINUE Line 13; }
17.       if (Deadline has not already passed) {
18.          if (a) or (b) hold, then Add_To_Planned_Schedule(a_i, s_j, h_k):
                /* when there is an estimate of a_i's running time */
              (a) end(h_k) + Act_Time(a_i, {s_j, h_k}) ≤ Deadline;
                /* when there is no estimate yet of a_i's running time */
              (b) Act_Time(a_i, {s_j, h_k}) = ∞ AND end(h_k) < Deadline;
     }} /* end of loop for host h_k */
19.    if ⟨a_i,s_j⟩ is not yet scheduled, then: if Budget allows, allocate a
          new host h_new and Add_To_Planned_Schedule(a_i, s_j, h_new);
20. }

21. /* Deallocation step: mark unused hosts for release. Marked hosts are
       released subject to the releaseHosts clause in the Plan statement */
22.  For host h_k in hosts h_1 to h_n, mark h_k for release if it is free;

Function Add_To_Planned_Schedule (action a, snapshot s, host h) {
23.  if (status(⟨a,s⟩) = "RUNNING on host h") {
24.     Add ⟨a,s⟩ as the head of h's list in Planned_Schedule; }
     /* status(⟨a,s⟩) is currently "WAITING" */
25.  else if (host h is currently free) {
26.     Add ⟨a,s⟩ as the head of h's list in Planned_Schedule;
27.     Set status(⟨a,s⟩) to "RUNNING on host h";
28.     end(h) = NOW + Act_Time(a, {s, h}); }
29.  else {
30.     Add ⟨a,s⟩ to the end of h's list in Planned_Schedule;
31.     end(h) = end(h) + Act_Time(a, {s, h}); }
}
```

Figure 5: Flex's scheduling algorithm

ways during orchestration, each of which will trigger a rerun of the scheduling algorithm from Section 5.2:

1. If the `Plan` statement's mapping specifies one or more $\langle a_i,s_j \rangle$ pairs for a future snapshot s_j, then these pairs will be added to *AS_Mapping* when, and only when, s_j becomes available.
2. If action a_i's estimated execution time in the initial *INPUT* is ∞ (i.e., it is unknown), then Flex will dynamically learn a model to better estimate a_i's execution time. Any changes to the estimated time will trigger a rerun of the scheduling algorithm.
3. An allocated host that was running experiments becomes free after finishing the experiments scheduled on it.

5.2 Scheduling Algorithm

Given the current *INPUT*, the scheduling algorithm generates the current *Planned_Schedule* (recall Figure 4). Flex will rerun the scheduling algorithm to generate a (possibly) new *Planned_Schedule* whenever the *INPUT* changes. The *Planned_Schedule* consists of an ordered list of currently-allocated hosts $h_1, \ldots, h_n$, with each host h_k having an ordered list of $\langle a_i,s_j \rangle$ pairs that are scheduled to run on h_k.

Figure 5 shows the overall scheduling algorithm. The goal of the algorithm is to minimize the total cost of running all the experiments while trying to ensure that all the experiments complete by the specified deadline.[4] The algorithm processes the *INPUT* using a sequence of five steps that will eventually generate the *Planned_Schedule*.

Total Ordering Step: This step, presented in Lines 1-2 in Figure 5, rearranges the $\langle a_i,s_j \rangle$ pairs in *AS_Mapping* in order to create a totally ordered list. Actions on earlier snapshots are placed before actions on later snapshots. For actions on the same snapshot, actions with longer (possibly incorrect) estimated running times are placed first. Since the $\langle a_i,s_j \rangle$ pairs in *AS_Mapping* are scheduled starting from the beginning of the ordered list, placing the longer actions first tends to increase the chances of meeting the given deadline.

No-Preemption Step: This step is presented in Lines 3-6 in Figure 5. Once an action starts running on a host, Flex will never preempt the action because preemption can interfere with the testing or tuning activities being performed.

Exploration Step: This step is presented in Lines 7-11 in Figure 5. In order to generate good schedules, Flex needs good models to estimate the running times of actions. The rationale behind the Exploration step is that, if the scheduling algorithm is invoked without being given a model for action a_i, then scheduling an instance of a_i upfront (in conjunction with the run-time monitoring that Flex does) can collect valuable information about a_i quickly; and lead to the generation of an efficient schedule overall. Caution is applied to only schedule actions that have unknown times and a similar action is not running already.

Greedy Packing Step: This step uses the current estimates of the execution time of actions in order to do a *bin packing* of the $\langle a_i,s_j \rangle$ pairs (experiments) on to as few hosts as possible—to minimize the number of hosts used—while ensuring that all experiments will complete within the deadline. Lines 12-20 in Figure 5 describe the greedy technique used for bin packing. The two scenarios—when the deadline has passed, and when it has not—are handled differently.

If the deadline has already passed, then the algorithm tries aggressively to complete the remaining experiments as fast as possible: (i) preferably, by running them on a free host if one is available, or (ii) by allocating a new host if the Budget is not violated. If there is still time to the deadline (the typical case), then the algorithm is more conservative. An experiment is scheduled on host h_k if h_k can run this experiment within the deadline in addition to all other experiments already scheduled on h_k. If no such host can be found, then a new host will be added subject to the Budget. Actions with unknown (∞) running times have to be considered during packing as well. Since such action types are given priority in the Exploration step, they are treated more conservatively during packing.

Recall the Total Ordering step that groups $\langle a_i,s_j \rangle$ pairs based on the snapshot. The Greedy Packing step considers the $\langle a_i,s_j \rangle$ pairs in this order for packing on to the hosts which are also considered in a specific order. The combination of the two ordered traversals gives a desired effect: the actions on the same snapshot have a good chance of being assigned to the same host (subject to the Deadline and Budget goals). If multiple actions on a snapshot s_j are assigned to host h_k, then h_k can share the overhead of snapshot loading

[4] Cloud providers like Amazon Web Services charge a per-hour cost for each host used during that hour.

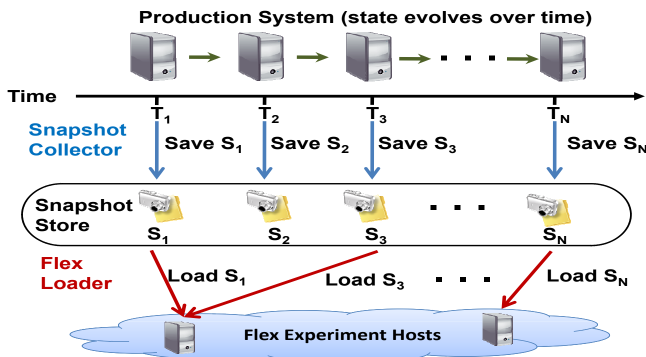


Figure 6: Snapshot management and Flex loading

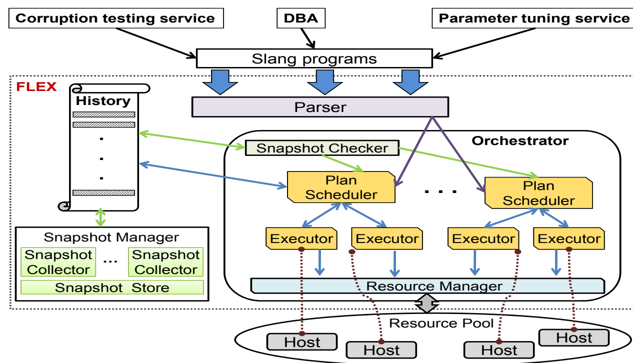


Figure 7: Architecture of the overall Flex platform

across these actions; which can reduce experiment running times significantly as we will see in Section 8.

Deallocation Step: The final step of the algorithm (Lines 21-22) marks any unused hosts for release. These hosts can be released subject to the releaseHosts clause in the `Plan` statement.

6. SNAPSHOT MANAGEMENT

Recall that a snapshot is a point-in-time state of the production database instance. This section discusses how snapshots are collected on the production database instance and then loaded on to hosts for running experiments. An overview of the process is given in Figure 6.

6.1 Snapshot Collection on the Production DB

A snapshot is collected as follows [10]:

1. Take a global read lock on the database, and flush the database-level dirty data to disk. For MySQL, e.g., this step performs "FLUSH TABLES WITH READ LOCK" with a preliminary flush done to reduce locking time.
2. Lock the file-system to prevent accesses, and flush its dirty data to disk. For example, for the XFS file system, this step runs the xfs_freeze command with a preliminary sync done to reduce locking time.
3. Run the storage volume's snapshot command to create a snapshot, and release the locks on the file-system and the database. For example, this step runs the ec2-create-snapshot command for *Elastic Block Store (EBS)* volumes on AWS.

The snapshot collection process typically causes less than a second of delay on the production database. A copy-on-write (COW) approach is used to further reduce overheads when multiple snapshots are collected over time. With COW, multiple copies need not be created for blocks that are unchanged across snapshots. The snapshot is then reported to the History catalog.

6.2 Loading a Snapshot for an Experiment

Before running an experiment $\langle a_i, s_j, h_k \rangle$, the snapshot s_j will be loaded on host h_k. The action a_i is started only after Flex certifies that s_j has been loaded. We categorize the loading process along two dimensions: *type* and *mechanism*.

Load Type specifies what part of the snapshot is copied to the host from where the snapshot is stored (e.g., on the production database or the S3 store on AWS). There are two load types:

- `Full`: Here, the full data in the snapshot is copied to the host (similar to a full backup).
- `Incremental`: This type works only when an unmodified copy of an earlier snapshot of the same storage volume is present on the host. Here, only the differences between the earlier and later snapshots are copied to the host (like incremental backup).

Load Mechanism specifies how much of the snapshot s_j has to be copied to the host before Flex certifies that s_j has been loaded and the action a_i can be started. There are two load mechanisms:

- `Eager`: Here, all the data needed for the snapshot has to be copied before the snapshot is certified as loaded.
- `Lazy`: Here, the snapshot is certified as loaded before all the data needed for the snapshot has been copied; and the action is started. Data copying proceeds in the background during which data blocks that are accessed on behalf of the running action will be prioritized.

Two systems utilities—namely, Linux's *Logical Volume Manager (LVM)* and Amazon's EBS—were used in careful combinations to implement the four possible snapshot loading techniques in Flex. Both utilities are already available to the public. In our implementation, we created a layer that leverages their functionality to achieve the four possible snapshot combinations. Snapshots of LVM volumes provide the implementation of generating increments (changed blocks since the previous snapshot) based on COW as well as metadata to identify which blocks have been changed.

EBS provides the implementation of `Lazy` that we use in Flex. `Lazy` combines: (i) regular push-based movement of the snapshot to the host, with (ii) pull-based (prioritized) movement of snapshot data accessed by an experiment on the host. Thus, `Lazy` needs an interposition layer that can schedule the data movement based on data access patterns. `Incremental+Lazy` achieves the following: suppose the source machine has snapshot s, its newer version s', and the increments Δ for s' with respect to s. A host, where an experiment has to run on snapshot s', currently has s only. Flex starts the experiment on the host over an interposed storage volume that uses `Lazy` to move only the increments Δ from the source to the host, and applies these increments to s.

Testing and tuning tasks tend to have specific needs regarding the load mechanism. For example, tuning tasks predominantly need the `Eager` mechanism in order to obtain trustworthy results: the run-time measurements that are generated in an experiment should correspond to what would have been obtained on the production database instance. With `Lazy`, these measurements can become tainted during the experiment due to unpredictable snapshot copying latencies.

7. IMPLEMENTATION OF FLEX

Figure 7 shows the overall architecture of the Flex platform. We describe the roles of each component.

Parser: The *Parser* extracts the definitions, mappings, and objectives from the Slang program submitted by a user or service, and performs the syntax checks as well as some basic semantic checks.

Orchestrator: The *Orchestrator* is the heart of Flex and is, in turn, composed of zero or more *Plan Schedulers*, a *Snapshot Checker*, zero or more *Executors*, and a *Resource Manager*. Once a parsed Slang program is obtained from the Parser, the Orchestrator instantiates one Plan Scheduler for each `Plan` statement in the program. (Recall that a Slang program can contain multiple `Plan` statements.) Each Plan Scheduler proceeds with its own independent scheduling. However, updates done or information gathered on behalf of actions are visible to all schedules.

Plan Scheduler: A Plan Scheduler starts with some initial checks of the feasibility of the objectives in the corresponding `Plan` statement, e.g., whether the Budget is enough to allocate at least one host. If the checks succeed, then the scheduling algorithm from Section 5.2 is run to generate the *Planned_Schedule*. The Scheduler will retrigger the scheduling algorithm if any of the events described in Section 5.1 were to occur. If the algorithm generates a new *Planned_Schedule*, then the new schedule becomes effective immediately (subject to Flex's policy of no preemption). The Plan Scheduler also instantiates an Executor for each host allocated by the schedule, and maintains a *host-to-action queue* that stores the ordered list of experiments assigned to this Executor by the current *Planned_Schedule*.

Executor: Any experiment in Flex is run by an Executor. When an Executor is started by the Plan Scheduler, it contacts the Resource Manager for allocating a host. The allocation as well as connection to the host use the authentication information provided in the `Credentials` statement. The Executor first runs any host-setup actions specified in the corresponding `Host` statement in the Slang program. Then, the Executor repeats the following steps until it is terminated:

- Get the next $\langle a_i, s_j \rangle$ pair from the head of the host's host-to-action queue. Mark $\langle a_i, s_j \rangle$ as *RUNNING* on this host.
- Reboot the host if the action a_i's `Action` statement in the program specifies it.
- Load the snapshot s_j on the host if the present copy is modified (Section 6 gives the details).
- Run the executable for the action a_i.

To track the running time and resource usage by each action, monitoring probes are performed once every *MONITORING_INTERVAL* (a configurable threshold that defaults to 10 seconds). The monitoring data collected is recorded persistently in the History catalog after the action completes.

The *Act_Time*(a, *parameters*) model (see Table 3) to compute the estimated running time of an action a is updated whenever (i) an experiment involving a completes, and whenever (ii) the running time of an (incomplete) experiment involving a starts to overshoot its current estimated running time by a *DEVIATION_THRESHOLD* (a configurable threshold that defaults to 5%). In our current implementation, the *Act_Time*(a, *parameters*) model is implemented as a running average of all the running times of a observed from (i) and (ii) so far. The scheduling algorithm will be rerun whenever there is a change in the estimated running time of an action since the model is part of the *INPUT* (Table 3).

Flex is robust to the failure of an experiment $\langle a_i, s_j \rangle$ running on a host h or the failure of h itself—the Executor for h will detect these failures—since the $\langle a_i, s_j \rangle$ pair will continue to be (re)considered by the scheduling algorithm until the experiment is complete. An Executor terminates when its Plan Scheduler sends a terminate signal; the Executor will clean up all used resources and release the host.

Resource Manager: The Resource Manager implements a general interface that supports any resource provider (e.g., Amazon Web Services, Rackspace, SQL Azure) from which hosts can be allocated on demand. The Executor uses the interface provided by the Resource Manager to allocate, establish connections with, and terminate hosts.

Snapshot Checker: The Snapshot Checker checks the History catalog periodically for newly available snapshots. When a new snapshot s is detected, the Checker maps s to the appropriate Plan Schedulers (based on their `Snapshot` statement and the mapping clause in their `Plan` statement). These Plan Schedulers will be notified of the new snapshot—the *AS_Mapping* in *INPUT* will change—and they will rerun the scheduling algorithm to update their current schedule with the newly available snapshot.

History Catalog: The History catalog is the persistent information repository of the entire Flex platform. This repository stores information about actions, snapshots, hosts, and plans, starting from the information in the Slang program to the execution-time information collected on actions and hosts through monitoring. Apart from its use internally in Flex, the catalog is also useful to DBAs as well as higher-level services implemented on top of Flex. The current implementation of the History catalog is in the form of seven tables in a PostgreSQL database. (Because of space constraints, the schema of these tables is given in the technical report [3].) In future, we plan to port the catalog to a distributed database or NoSQL engine.

Figure 7 shows two other components that are not internal to Flex, but are important components of the overall Flex platform: the *Snapshot Manager* and *Flex Services*.

Snapshot Manager: The Snapshot Manager is responsible for collecting snapshots regularly from the production database instance, possibly storing them persistently on remote storage for disaster recovery, and updating the History catalog as snapshots become available. Snapshot Managers are becoming an essential part of the IT environment in modern enterprises given the growing importance of near-real-time disaster recovery and *continuous data protection*. Section 6 describes the Snapshot Manager implemented as part of the Flex prototype.

Flex Services: A number of manageability tools have been proposed in the literature that rely fully or in part on experiments done using production-like workloads, configuration, data, and resources (e.g., [2, 4, 6, 9, 14, 26, 27]). Flex can benefit each tool—if the tool runs on Flex as a Flex Service—in multiple ways:

- Providing a common high-level interface to specify the experiments needed by the tool. The tool can then focus on determining what experiments to do rather than how to implement scheduling, execution, and fault-tolerance for the experiments.
- Automatically learning performance models and generating time- and host-efficient schedules for the experiments.

Section 8.4 gives a case study based on one such tool called *Amulet* [4] that we have ported to run as a Flex Service.

8. EVALUATION

This section presents a comprehensive evaluation of Flex. This evaluation uses the Amazon Web Services (AWS) cloud platform [1] as the resource provider. (Note that the applicability of Flex is not limited to AWS or to cloud platforms.) We present insights from (i) real-life scenarios that we recreated, (ii) evaluating the major components of Flex, and (iii) porting existing testing and tuning applications to run as Flex Services. Most of our experiments consider a production MySQL database instance that runs on an AWS host with an XFS or Ext3 Linux file-system and storage provided by AWS's Elastic Block Store (EBS) or the local disk space of the host. 50GB EBS storage volumes are used by default.

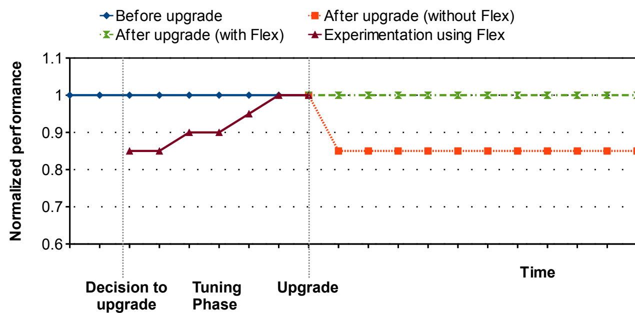


Figure 8: A performance degradation experienced after upgrading the production MySQL database instance, and how Flex helps to avoid the problem

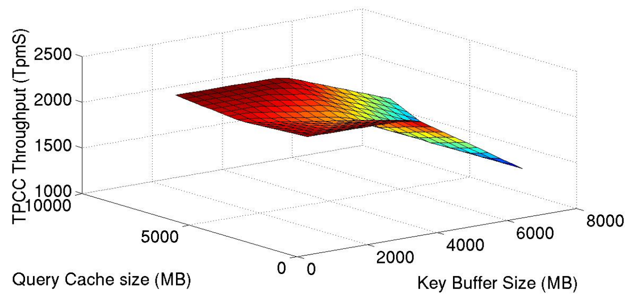


Figure 9: TPC-C throughput for various configurations

8.1 Benefits of Flex in Real-Life Scenarios

We begin with a problem that happened in real life in order to illustrate how Flex can be used to minimize the occurrence of such problems. Suppose a DBA has to upgrade the production database instance from MySQL 4.1 version to MySQL 5.0.42 version. The timeline in Figure 8 shows the scenarios with and without Flex. DBAs usually run benchmark workloads before an upgrade. In this case, the benchmark workloads ran fine, but performance dropped noticeably when the production instance was upgraded. This problem happened due to a bug in the 5.0 version of MySQL which slows the group commit of transactions. A lock gets acquired too early, causing other concurrent transactions to stall until the lock is released. This bug only showed up on the high transaction rate seen in the production database.

With Flex, the DBA can write a Slang program (like the A/B testing or Upgrade planning use-cases in Table 1) to test the upgrade first on production-like workloads, configuration, data, and resources; without affecting the production database instance. Thus, the bug is noticed long before the actual upgrade has to be done. The DBA has time to further use Flex to try different configurations, find a fix, and verify that the fix will work on the production instance. To resolve this bug, the DBA either needs to disable the collection of binary logs or set the MySQL parameter *innodb_flush_log_at_trx_commit* to a value different from 1. When this parameter is set to 1, MySQL flushes the log after each transaction. A value of 0 flushes the log once per second, while a value of 2 writes the log at every transaction but flushes once per second.

Our next real-life application is tuning surface creation (see the tuning surface creation use-case in Table 1). For a representative workload taken from a past database state, the DBA wants to get a feel for the (hypothetical) response times for different settings of server configuration parameters [9]. Specifically, the DBA has a MySQL database that runs on an EC2 m1.large host and uses a 100GB volume of local storage. For the workload, we used the popular TPC-C benchmark with 100 warehouses, a warm up time of 5 minutes, and measurement time of 20 minutes. The parameters that the DBA is interested are key_buffer_size (the buffer used to store index blocks when accessing the tables) and query_cache_size (the amount of memory allocated for caching query results).

The DBA writes a Slang program to specify the experiments. Flex orchestrates the program to produce the monitoring data used to generate the throughput surface in Figure 9. The z-axis represents the throughput that the MySQL database can sustain given the different configuration parameter settings. After producing this tuning surface with little effort, the DBA can confidently find pick a configuration for good performance on the production database.

8.2 Impact of Scheduling Algorithm

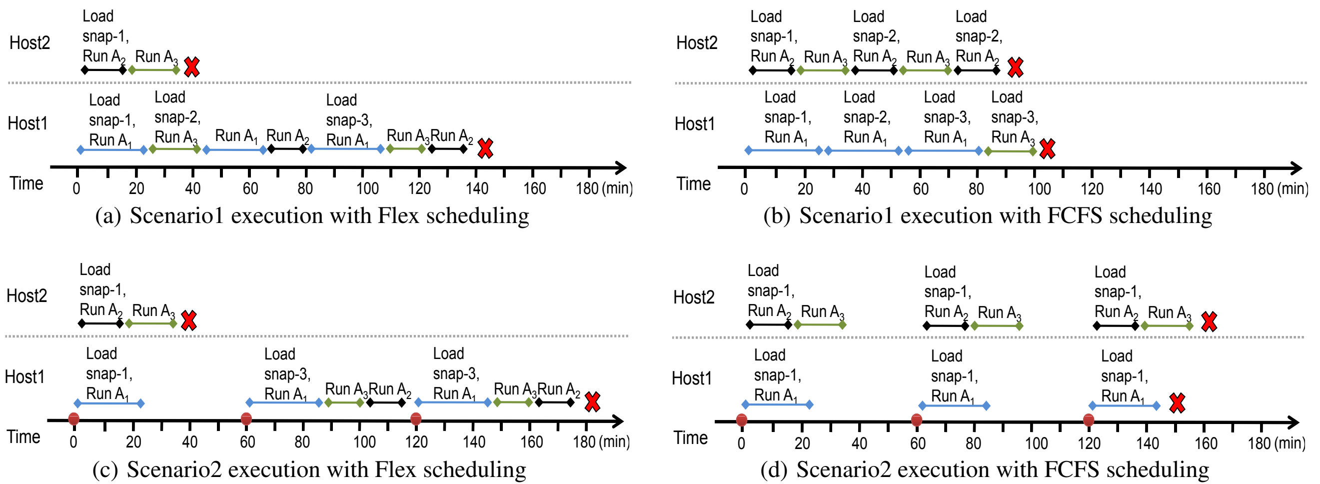
Recall the DBA's task presented in Section 3 and its Slang program from Figure 2. The details of this task in our empirical setup are as follows: action A_1 runs myisamchk on table Lineitem (10GB size); action A_2 runs myisamchk on table Order (5GB); and action A_3 runs fsck on the full 50GB volume. All three snapshots are `Full` and `Lazy` on the storage volume of size 50GB and are available when the Slang program is submitted for execution. Since none of the actions modify the snapshot data, the DBA has specified that no host reboot or snapshot reload is required (see Figure 2).

Schedule and Execution: Figure 3 presents the actual execution of the Slang program. (All execution timelines are drawn to scale to represent the actual execution of the actions.) None of the actions have been run before by Flex, so there is no history information in the History catalog. Thus, the scheduling algorithm starts with the exploration step, and tries to obtain models for each of the actions. For the exploration step, the algorithm uses the maximum 3 hosts allowed by the DBA. An action that runs on the first snapshot (snap-1) is scheduled on each host.

Action A_2 on snap-1 completes first, and gives a better model for action A_2. This event triggers the scheduling algorithm. The scheduling algorithm identifies that there are no actions with unknown models and in the "*WAITING*" state; thus, it proceeds with the Greedy Packing step. Note that actions A_1 and A_3 will be prioritized by the Total Ordering step as their estimated running times are still unknown (∞). The next action that completes execution is A_3 on snap-1; the scheduling algorithm is invoked again (the model for action A_3 is updated). The scheduling algorithm now places A_3 on snap-2 in Host3's host-to-action queue (based on the Total Ordering step, action A_3 has higher completion time than A_2 for snap-2). When action A_1 on snap-1 completes, better models for all actions become available. The scheduling algorithm now realizes that only 1 host is needed to complete all actions within the deadline. Thus, Host2 and Host3 are released when they complete their current actions.

As Figure 3 shows, action A_3 on snap-3 is taking less time than the same action executed on snapshots snap-1 and snap-2. This behavior is a result of the snapshot-loading mechanism. During the execution of action A_1 on snapshot snap-3, most of the snap-3 data needed by action A_3 is pre-fetched; so the data-intensive A_3 finishes faster. For action A_2, the effect of pre-fetched data is smaller as this action touches less data.

We further explore the impact of the scheduling algorithm by tightening the objectives and increasing the amount of data that needs to be verified from the first DBA task. In this way, we force the decisions of the scheduling algorithm to have a higher impact on the overall execution. The setup for this scenario (we refer to it as Scenario1) is the same as before except for: Lineitem is now 11GB and Order is 7.5GB. The Budget was changed to 2 hosts of EC2 m1.large type, and Deadline is now 3 hours (180 minutes).



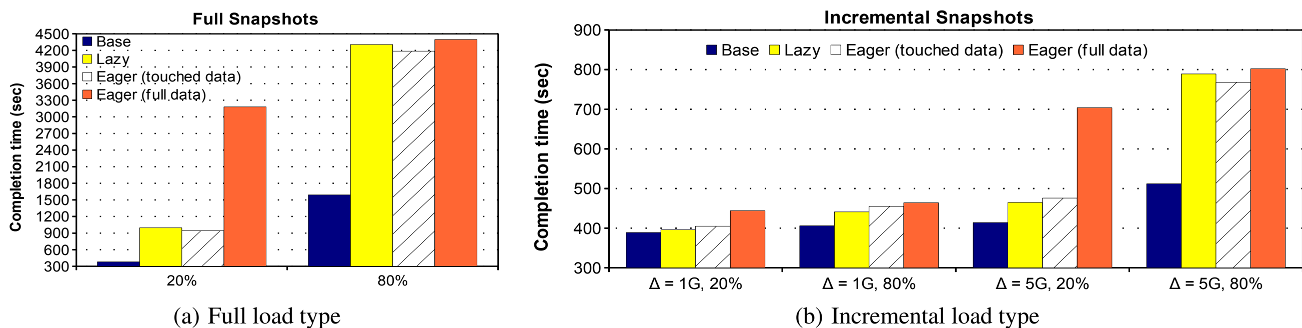
(a) Scenario1 execution with Flex scheduling

(b) Scenario1 execution with FCFS scheduling

(c) Scenario2 execution with Flex scheduling

(d) Scenario2 execution with FCFS scheduling

Figure 10: Evaluation of the scheduling algorithms for Scenario1 and Scenario2



(a) Full load type

(b) Incremental load type

Figure 11: Running basic action a on snapshots with incremental and full load types

For comparison purposes, we created a baseline scheduling algorithm which executes the actions on an FCFS (First Come First Serve) basis. (Recall the challenges we outlined in Section 5 that scheduling algorithms face in the Flex setting; which make FCFS a very practical baseline.) Intuitively, FCFS tries to complete actions as fast as possible, ignoring the flexibility given by the deadline. FCFS prioritizes actions on the earliest available snapshots, and as soon as a host completes an action, it proceeds with the next available action. The Flex scheduling algorithm does not have any prior information on the running times of actions, so that it does not have any starting advantage over the FCFS scheduler. Execution timelines for Scenario1 for the Flex and the FCFS algorithms are depicted in Figure 10(a) and Figure 10(b) respectively.

As seen in Figures 10(a) and 10(b), all actions can be run by just one host and still complete within the given deadline of 3 hours. (That would be the optimal schedule.) However, none of the scheduling algorithms realize this opportunity. The exploration phase of the Flex scheduling algorithm uses the maximum number of two hosts. These hosts are needed to build the models for the execution of the actions. After that, Flex converges quickly to the optimal schedule. FCFS uses the two hosts all the time. Note that FCFS did not converge to the optimal strategy and placed a higher concurrent resource demand than needed to meet the DBA's requirements subject to the specified deadline. This behavior is undesirable under bursty workloads when a number of Slang programs will be submitted over a short period of time.

If the deadline was set by the DBA to a lower value of 100 minutes, then both algorithms will produce similar execution timelines. Nevertheless, Flex has an advantage because it exploits the short execution times of actions that have already preloaded data; see action A_3 on snap-3 in Figure 10(a).

The results observed so far are for cases where all the snapshots are available. In our next scenario (termed Scenario2), we changed the snapshots from "already available" to "expected to come in future". The snapshots in Scenario2 arrive at times 0, 60 and 120. That is, there is a Snapshot Manager that takes snapshots every hour, and reports them to Flex. Figures 10(c) and 10(d) show the actual execution of Flex and the baseline FCFS scheduling in Scenario2. Each red dot on the time axis marks the arrival of a snapshot. The conclusions from Scenario1 hold here as well. The hosts are underutilized most of the time by FCFS in Scenario2. However, Flex will adapt quickly to the optimal schedule of using 1 host. In summary, these experiments illustrate the elastic and adaptive nature of Flex's scheduling algorithm.

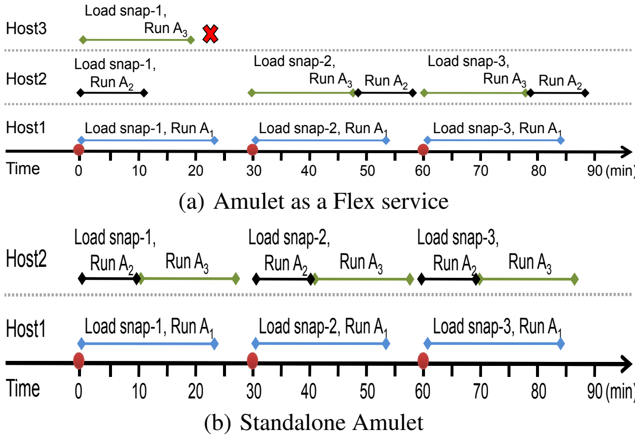
8.3 Impact of Snapshot Loading Techniques

In Section 6, we described two options each for the snapshot loading type and mechanism. We now evaluate the four resulting choices empirically: (i) `Full` + `Eager`; (ii) `Full` + `Lazy`; (iii) `Incremental` + `Eager`; and (iv) `Incremental` + `Lazy`. To investigate the performance impact of the snapshot loading process, we use a basic action a that reads all data pages from the snapshot, extracts the records, validates the data page's checksum and each record's checksum, and verifies that all fields in every record are consistent with the database schema.

Figure 11(a) shows the results for loading the full snapshot data with lazy and eager mechanisms. The base bar represents execution of the action when the data is completely loaded and available on the host. Note that the y axis starts from 300 seconds in order to make the graph more legible. We varied the amount of data that the action touches, and used values of 20% (10GB out of 50GB) and 80% (40GB) as represented on the x-axis. The bar "Lazy"

Module	Lines of Code
Angel Language	300
Input Definitions & Validation	2800
Configuration & Utilities	1100
Main modules	1000
Optimizer	2000
Models Learning	1500
Execution & Monitoring	4000
Cloud communications	1500

Table 4: Lines of code in standalone Amulet



(a) Amulet as a Flex service

(b) Standalone Amulet

Figure 12: Amulet running as standalone and as a Flex service

represents the execution of the action when the snapshot is loaded in a lazy manner. The "Eager (touched data)" bar represents the time to execute the action after loading only the data that the action touches. The "Eager (full data)" bar represents the loading of the complete snapshot data and the following action execution time. Eager (touched data) slightly outperforms lazy due to the fact that lazy loading process also loads some data that is not touched by the action.

Figure 11(b) shows the results for loading the snapshot incrementally (only the delta changes are loaded) with lazy and eager mechanisms. Note that, to use incremental snapshot loading, an earlier version of the snapshot data should be present on the host. We varied the delta change (denoted Δ), i.e., the amount of changed data between the earlier version and the snapshot that needs to be loaded. We used Δ of 1GB and 5GB. The action a touches 20% and 80% of the data. This percentage also reflects the data that is part of the delta. That is, when the action touches 20% (10GB) of the overall data, then it also touches 20% of the Δ change. For Δ = 5GB, the action touches 9GB of the regular snapshot data, and 1GB of the Δ change. Again, we see that lazy outperforms or is equal to the eager mechanism.

In summary, incremental + lazy is a robust strategy that outperforms or is equal to the other options, followed by incremental + eager, full + lazy, and last full + eager. However, recall from Section 6 that incremental + lazy cannot be used for all actions. If an action needs trustworthy measurements of running time as what would have been observed on the production database instance, then the eager mechanism is mandatory.

8.4 How Flex Reduces Development Effort

To study the utility of Flex as a platform for rapid development of testing and tuning applications, we ported the *Amulet* testing tool [4] to run as a Flex Service. Amulet aims to verify the integrity of stored data proactively and continuously. Amulet runs *tests* which are special programs that perform one or more checks—e.g., comparison of a disk block with its stored checksum, comparing data in a table T versus the corresponding data in an index on T, or comparing the recent versions of data in a table versus the database log—in order to detect corruption in stored data. To satisfy all the data correctness and performance constraints specified by the DBA, Amulet plans and conducts a series of tests. For planning purposes, Amulet has a long training phase where it first runs tests in order to collect performance data for learning models.

Table 4 shows the lines of code for different modules in the standalone implementation of Amulet. The total lines of code are 14200 (a reasonable amount for Amulet's functionality). Note that execution and cloud communications are major parts of the source code, comprising 5500 lines. Porting Amulet to run as a Flex service directly makes this part of the Amulet code obsolete. Converting Amulet's testing plans to Flex's Slang language required around 700 lines of code. By creating a single Slang program, Amulet's model learning functionality can also be offloaded to the Flex platform; reducing the total code base to 6300 lines (taking into account some changes to create the input for Flex). This significant code base reduction shows how Flex allows developers to focus on the functionality of the service rather than the low-level details of scheduling and running experiments; especially on remote cloud platforms where failures are more common than on local clusters.

Next, we investigate the behavior of Amulet when run as a standalone tool and as a Flex service. Figure 12 shows the respective runs on snapshots that arrive over time. Note that both runs are almost identical. Flex starts with 3 hosts to learn the models of the actions. Amulet uses only 2 hosts as the model learning is performed before the execution of the testing schedule. When Flex learns the models, it converges quickly to the Amulet plan. There is a difference in the order of actions A_2 and A_3, but the order does not affect the DBA's requirements.[5]

This result illustrates how Flex can provide testing and tuning tools with fairly good performance while freeing them from the nontrivial complexity of generating training data, model learning, and scheduling. Also note that a Slang program (omitted due to space constraints) with less than 100 lines of code is all it takes to obtain the monitoring data needed for Figure 9. Adding this program to a library makes it reusable by other Flex users.

8.5 Scalability Test for Bursty Workloads

Since we expect the typical workload of Flex to be bursty, Flex is designed to run multiple Slang programs concurrently. As a scalability test, we had twenty different applications connecting to Flex and submitting Slang programs. To run all programs, Flex allocated 40 hosts from the resource provider (AWS). The experiments from all programs were completed as per the requirements specified.

9. RELATED WORK

A/B testing has become a popular methodology that companies use to test new user-interfaces or content features for websites. Tests are run on a random sample of live users to gauge user reactions fairly quickly and accurately. An example is the Microsoft *ExP* platform for analyzing experimental features on websites [17]. Facebook has a related, but architecturally different, initiative called *Dark Launch* [8]. Dark launch is the process of enabling a feature without making it visible to users. The goal is to test whether the back-end servers will be able to handle the workload if the feature were to be made generally available. Both of these are examples of platforms that are related to Flex's goal of easy experimentation in production deployments. However, Flex differs from them in two major ways: focusing on database testing and tuning and not tar-

[5] An order of actions can be enforced using the $\langle a_i, s_j, h_k \rangle$ triples form of mapping; see Section 4.

geting experiments on live applications or users, leading to a very different architecture.

Salesforce's *Sandbox* initiative underscores the need for platforms like Flex [23]. Sandbox provides a framework that developers can use to create replicas of the production data for testing purposes. The Dexterity automated testing framework verifies database correctness and performance by performing regression or ad-hoc testing based on the database schema [12]. *JustRunIt* is an automated framework for experimentation [27]. JustRunIt uses virtual machines to replicate the workload and resources. While all these frameworks automate the low-level details of running experiments, each one of them is specific to one or more tasks. Frameworks like *Chef*, *Puppet*, and *Scalr* take declarative system specifications as input and use them to configure and maintain multiple systems automatically as well as start and stop dependent services correctly.

Compared to the above frameworks, Flex innovates in a number of ways by providing: (i) the Slang language with key abstractions suited for specifying experiments and objectives, (ii) a new scheduling algorithm with a mix of exploration and exploitation to meet the given objectives, (iii) adaptive and elastic behavior to reduce resource usage costs, and (iv) multiple techniques to transfer evolving data from the production database to hosts that run experiments.

A large body of work focuses on minimizing the impact on the production database instance while performing administrative tasks such as online index rebuild [20], database migration [11], and defragmentation. This line of work helps the migration from one host to another or from one configuration to another. However, before doing a change to the production database, the DBA needs to determine the benefits and drawbacks of this change. Here is where Flex helps by allowing the DBA to experiment before actually doing the change. As future work, we can integrate these services with Flex so that the DBA can easily experiment with changes and then apply them non-intrusively to the production database.

A number of testing and tuning tools from the literature such as Amulet, *HTPar*, and *iTuned* can be implemented as Flex Services easily. Section 8.4 described our experiences in porting the Amulet testing tool to run as a Flex Service. HTPar is a regression testing tool [14]. iTuned is tuning tool for database configuration parameters [9]. To find a good configuration from an unknown tuning surface like Figure 9, iTuned proceeds iteratively: each iteration issues a set of experiments with different server configurations to collect performance data.

10. FUTURE WORK

We presented Flex, a platform that enables efficient experimentation for trustworthy testing and tuning of production database instances. Flex gives DBAs a high-level language to specify definitions and objectives regarding running experiments for testing and tuning. Flex orchestrates the experiments in an automated manner that meets the objectives. We presented results from a comprehensive empirical evaluation that reveals the effectiveness and usefulness of Flex. We envision three interesting avenues for future work:

- *Using Flex to create self-tuning cloud databases:* Consider a Flex Service that automatically tunes production database instances running on a cloud platform like AWS or SQL Azure. This service will continuously monitor databases for performance degradation, e.g., tracking Service-Level-Agreement violations. Based on the workload observed, the service will use tuning tools like index wizards and iTuned [9] to come up with potential fixes. The service will then perform experiments to find and validate the best fix, which will then be applied to the production database to complete the self-tuning loop.
- *Extending Flex to support parallel databases:* A promising direction is the integration of Flex with *Mesos* which is a cluster manager that provides efficient resource isolation and sharing across distributed applications [15]. Flex can use Mesos to create a "distributed host" which will encapsulate a set of hosts required to run an experiment for a parallel OLTP database or a NoSQL engine.
- *Making the Flex platform richer:* Many opportunities exist to integrate Flex with orthogonal tools like fine-grained workload replay [13] or the *UpSizeR* that enables data sizes to be scaled up or down in a trustworthy manner [25]. Flex can also be extended with fine-grained access control when providing access to production data; enforced currently by the coarse-grained Credentials statement in Slang.

11. REFERENCES

[1] *Amazon Web Services*. aws.amazon.com.
[2] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic Exploration of Datacenter Performance Regimes. In *Automated Control for Datacenters and Clouds*, 2009.
[3] N. Borisov and S. Babu. Rapid Experimentation for Testing and Tuning a Production Database Deployment. Technical report, Duke University, 2012. http://bit.ly/Hz6U5w.
[4] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Warding off the Dangers of Data Corruption with Amulet. In *SIGMOD*, 2011.
[5] *Business Process Execution Language*. http://bit.ly/HI1LFY.
[6] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Exact Cardinality Query Optimization for Optimizer Testing. *PVLDB*, 2009.
[7] *Data corruption in CouchDB*. couchdb.apache.org/notice/1.0.1.html.
[8] *Facebook dark launch*. http://on.fb.me/RWsOO.
[9] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *VLDB*, 2009.
[10] *Running MySQL on Amazon EC2 with EBS*. http://bit.ly/b7SWwg.
[11] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, 2011.
[12] D. J. Farrar. Schema-driven Experiment Management: Declarative Testing with Dexterity. In *DBTest*, 2010.
[13] L. Galanis, S. Buranawatanachoke, et al. Oracle Database Replay. In *SIGMOD*, 2008.
[14] F. Haftmann, D. Kossmann, and E. Lo. Parallel Execution of Test Runs for Database Application Systems. In *VLDB*, 2005.
[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX*, 2011.
[16] The Exploding Digital Universe. http://bit.ly/bzgTBq.
[17] R. Kohavi, T. Crook, and R. Longbotham. Online Experimentation at Microsoft. In *Workshop on Data Mining Case Studies and Practice Prize*, 2010.
[18] *Data corruption at Ma.gnolia.com*. en.wikipedia.org/wiki/Gnolia.
[19] *MySQL upgrade from 4 to 5*. http://bit.ly/sV9PIf.
[20] *Oracle online index rebuild*. http://bit.ly/trDrGe.
[21] *Oracle upgrade regression*. http://bit.ly/uOHwB1.
[22] *PostgreSQL TPCH bug*. http://bit.ly/rIJK1w.
[23] *Salesforce Sandbox*. http://bit.ly/7Bi1jU.
[24] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *ICDE*, 2010.
[25] *UpSizeR*. http://upsizer.comp.nus.edu.sg/upsizer/.
[26] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer. *DEB*, 2008.
[27] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *USENIX*, 2009.