

Integrated Management of the Persistent-Storage
and Data-Processing Layers in Data-Intensive
Computing Systems

by

Nedyalko Borisov

Department of Computer Science
Duke University

Date: _____

Approved:

Shivnath Babu, Supervisor

Jeffrey Chase

Sandeep Uttamchandani

Jun Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

ABSTRACT

Integrated Management of the Persistent-Storage and
Data-Processing Layers in Data-Intensive Computing Systems

by

Nedyalko Borisov

Department of Computer Science
Duke University

Date: _____

Approved:

Shivnath Babu, Supervisor

Jeffrey Chase

Sandeep Uttamchandani

Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

Copyright © 2012 by Nedyalko Borisov
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Over the next decade, it is estimated that 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. Meanwhile, skilled information technology (IT) staff to manage the growing number of servers and data will increase less than 1.5 times. Thus, a system administrator will face the challenging task of managing larger and larger numbers of production systems. We have developed solutions to make the system administrator more productive by automating some of the hard and time-consuming tasks in system management. In particular, we make new contributions in the Monitoring, Problem Diagnosing, and Testing phases of the system management cycle.

We start by describing our contributions in the Monitoring phase. We have developed a tool called Amulet that can continuously monitor and proactively detect problems on production systems. A notoriously hard problem that Amulet can detect is that of data corruption where bits of data in persistent storage differ from their true values. Once a problem is detected, our DiaDS tool helps in diagnosing the cause of the problem. DiaDS uses a novel combination of machine learning techniques and domain knowledge encoded in a symptoms database to guide the system administrator towards the root cause of the problem.

Before applying any change (e.g., changing a configuration parameter setting) to the production system, the system administrator needs to thoroughly understand the effect that this change can have. Well-meaning changes to production systems have led to performance or availability problems in the past. For this phase, our Flex tool enables administrators to evaluate the change hypothetically in a manner that is fairly accurate while avoiding overheads on the production system. We have conducted a comprehensive evaluation of Amulet, DiaDS, and Flex in terms of effectiveness, efficiency, integration of these contributions in the system management cycle, and how these tools bring data-intensive computing systems closer the goal of self-managing systems.

Contents

Abstract	iv
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Monitoring	4
1.2 Diagnosing	6
1.3 Testing	7
1.4 Integrated and Automated System Management	8
2 Monitoring with Amulet	10
2.1 Introduction	11
2.1.1 Amulet: Challenges and Overview	17
2.2 Related Work	23
2.2.1 The Dangers of Data Corruption	23
2.2.2 Dealing with Data Corruption	24
2.3 Modeling of Tests	26
2.3.1 Data-Dependent Attributes	27
2.3.2 Resource-Dependent Attributes	29
2.3.3 Transient Effects	30
2.4 The Angel Declarative Language	31

2.4.1	Objectives	33
2.5	Optimizer	37
2.5.1	Testing Plans in Amulet for a Volume V	38
2.5.2	Selecting the Snapshot Interval	39
2.5.3	Selecting the Plan Window	40
2.5.4	Selecting the Test-to-Snapshot Mapping	41
2.5.5	Selecting the Schedule of Test Execution	42
2.5.6	Handling Non-cost Optimization Objectives	45
2.6	Orchestrator	47
2.7	Experimental Evaluation	50
2.7.1	Experimental Methodology and Setup	50
2.7.2	End-to-End Processing of Angel Programs	52
2.7.3	Evaluation of Amulet’s Optimizer	58
2.8	Conclusions and Future Work	60
3	Diagnosing with DiaDS	61
3.1	Introduction	62
3.1.1	Challenges in Integrated Diagnosis	64
3.1.2	Contributions	65
3.2	Related Work	67
3.2.1	Independent DB and Storage Diagnosis	67
3.2.2	System Diagnosis Techniques	68
3.3	Overview of DiaDS	70
3.4	Modules in the Workflow	74
3.4.1	Identifying Correlated Operators	74
3.4.2	Dependency Analysis	76

3.4.3	Symptoms Database	78
3.4.4	Impact Analysis	83
3.5	Experimental Evaluation	85
3.5.1	Setup Details	86
3.5.2	Scenario 1: Volume Contention due to SAN Misconfiguration	87
3.5.3	Scenario 2: Database-Layer Problem Propagating to the SAN-Layer	94
3.5.4	Scenario 3: Concurrent Database-Layer and SAN-Layer Problems	95
3.5.5	Discussion	96
3.6	Conclusions and Future Work	97
4	Testing with Flex	99
4.1	Introduction	100
4.1.1	Flex	104
4.1.2	Contributions and Challenges	105
4.2	Abstraction of an Experiment	107
4.3	Walk-through of Flex Usage	110
4.4	Slang Language	111
4.5	Orchestration of Experiments	116
4.5.1	Inputs for Orchestration (<i>INPUT</i>)	117
4.5.2	Scheduling Algorithm	118
4.6	Snapshot Management	121
4.6.1	Snapshot Collection on the Production DB	122
4.6.2	Loading a Snapshot for an Experiment	123
4.7	Implementation of Flex	124
4.8	Evaluation	129

4.8.1	Benefits of Flex in Real-Life Scenarios	129
4.8.2	Impact of Scheduling Algorithm	131
4.8.3	Impact of Snapshot Loading Techniques	135
4.8.4	How Flex Reduces Development Effort	136
4.8.5	Scalability Test for Bursty Workloads	139
4.9	Related Work	139
4.10	Future Work	141
5	Conclusions and Future Work	143
	Bibliography	146
	Biography	155

List of Tables

2.1	Data integrity tests to detect and possibly repair data corruption at the application level	13
2.3	Data integrity tests to detect and possibly repair data corruption at the file-system level	15
2.4	Data integrity tests to detect and possibly repair data corruption at the hardware level	15
2.5	Examples of objectives that a SA may have regarding timely detection and repair of data corruption	17
2.6	Amulet notation	32
2.7	Summary of important Angel statements	33
2.8	Pricing model used in Amulet’s evaluation	52
3.1	Anomaly scores for query operators	90
3.2	Anomaly scores for volume metrics	91
4.1	Listing of some nontrivial uses enabled by the Flex platform	102
4.2	Summary of statements in the Slang language	111
4.3	Inputs to the Flex’s orchestration process	117
4.4	Lines of code in standalone Amulet	137

List of Figures

1.1	The future growth trends (Digital Universe, 2011)	2
1.2	Our vision to help the system administrators	3
2.1	Overview of Amulet’s optimization and orchestration	18
2.2	Amulet Example 1 execution timeline	18
2.3	Effect of data-dependent attributes on the completion time of the fsck, xfs_check, and myisamchk tests	27
2.4	Effect of resource-dependent attributes and concurrent test execution on the execution time of myisamchk tests	28
2.5	Angel program for Example 1 in Table 2.5	34
2.6	Illustration of RPO	35
2.7	Finding the least-cost plan for given objectives O_1-O_n	37
2.8	Selection of plan interval	40
2.9	Selection of plan mapping	41
2.10	Selection of plan schedule	44
2.11	Linear search algorithm to find the best plan for a volume	45
2.12	Binary search algorithm to find the best plan for a volume	46
2.13	Amulet’s Orchestrator	47
2.14	Actual execution timeline for Case 2	52
2.15	Actual execution timeline for Case 3	57
2.16	Actual execution timeline for Case 4	57

2.17	Characteristics of the testing plan space and the performance of Amulet’s Optimizer	57
3.1	Example database/SAN deployment	63
3.2	Taxonomy of scenarios for root-cause analysis	64
3.3	DiaDS’s diagnosis workflow	72
3.4	Example codebook	80
3.5	Query plan, operators, and dependency paths	88
3.6	Important performance metrics collected by DiaDS	89
3.7	Sensitivity of anomaly scores to the number of satisfactory samples	90
3.8	Sensitivity of anomaly scores to noise in the monitoring data	91
4.1	Overview of how Flex enables trustworthy experimentation	105
4.2	Example slang program #1 that references past snapshots	109
4.3	Execution of example slang program #1	110
4.4	Steps in Flex’s orchestration of experiments	115
4.5	Example slang program #2 that references future snapshots	115
4.6	Flex’s scheduling algorithm	120
4.7	Snapshot management and Flex loading	122
4.8	Architecture of the overall Flex platform	125
4.9	A performance degradation after upgrade	129
4.10	TPC-C throughput for various configurations	131
4.11	Evaluation of the scheduling algorithms	133
4.12	Running basic action on snapshots with incremental and full load types	134
4.13	Amulet running as standalone and as a Flex service	137
5.1	Overview of the system management cycle.	144

1

Introduction

Current manufacturers are increasing the capacity and speed of the computer hardware rapidly which allows businesses to grow their infrastructures and systems at a fast rate. These improvements allow the businesses to provide better services by increasing the types of services they provide, serving user requests faster, providing better product recommendation, allowing better fraud detection, and others. However, the increased amount of infrastructure and systems requires a set of skillful System Administrators (SA) to maintain and keep the systems trouble free. Recent reports (Digital Universe, 2011) estimate that the growth trends over the next decade will be as follows: the amount of servers will increase by 10 times, the stored data by 50 times, the number of files by 75 times, and the number of skillful administrators will increase by only 1.5 times (Figure 1.1 visualizes these trends). These trends show a gap between the growth of the systems versus the SAs who will manage these systems. Natural solutions to overcome this gap are: I) increasing the number of SAs, or II) improving the productivity of the SAs by automating their tasks.

Approach I - increasing the number of SAs - requires the companies to invest in hiring and training SAs at a rate close to the rate at which the systems are growing.

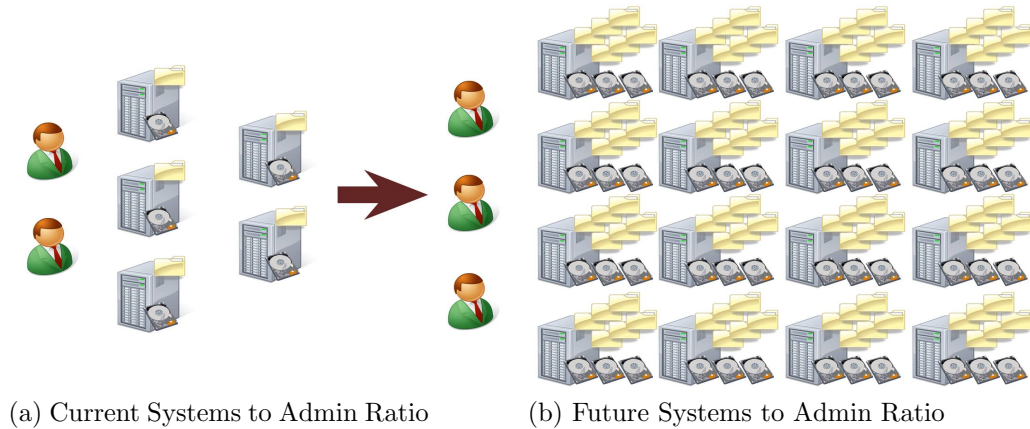


FIGURE 1.1: The future growth trends (Digital Universe, 2011)

However, the growth rate of systems is higher than the growth rate of the human population, which makes this approach unsustainable in the future. Furthermore, the current popularity of cloud services in terms of Infrastructure-as-a-Service (IaaS) or Software-as-a-Service (SaaS) allows regular users to lease a large number of system resources in a matter of minutes. In such settings, especially in the IaaS case, the management of the leased resources is left to the user; requiring the average user to be a skilled SA. The expectation that all users will be able to manage the leased resources is unrealistic, which could be a barrier to the use of cloud services. History teaches us that the simple and intuitive systems have a better adoption rate by the users, which focuses our attention to the second approach of reducing the burden of systems administration.

Approach II - reducing the burden of the SAs by automation of their tasks - is based on the idea of making SAs more productive (e.g., the approach of “Doing More With Less”). One way to make the SAs more productive is by creating self-managing systems. However, there are nontrivial challenges involved in achieving this type of systems. Another way to reduce the burden of SAs is to automate the most time-

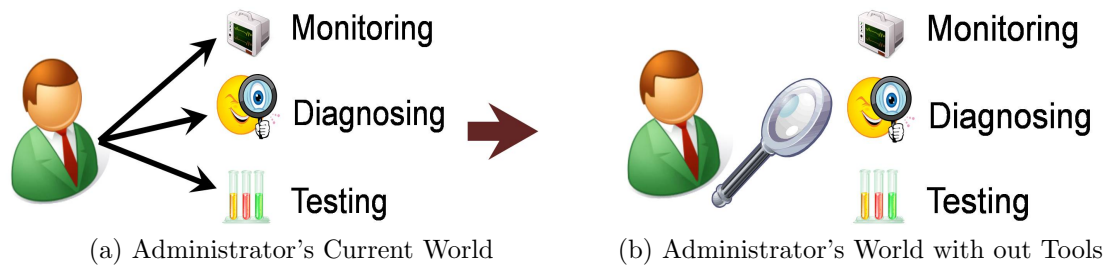


FIGURE 1.2: Our vision to help the SAs.

consuming and burdensome tasks among their duties. We embraced this idea and the dissertation explores the potential to make SAs more productive.

We started by identifying the burdensome tasks of the SAs. We identified three major task categories:

- **Monitoring** - checking the health of the production system and identifying problems that can arise in future
- **Diagnosing** - finding the root cause of the problems on the production system
- **Testing** - verifying that a certain “change” (e.g., configuration parameter change, upgrade of the system) will impact the production system in expected ways

Figure 1.2a depicts the three categories and their interaction with the SA. Currently, the tasks that fall in these categories are predominantly handled by the SAs manually. In this dissertation, we present the tools that we developed in each category to help the SAs and shift the view as showed in Figure 1.2b - the tools provide either the solution or the information/automation needed for the solution. The challenges involved in each category are discussed in the following sections in this chapter.

It is important to note that the software market is dynamic and new systems are constantly created that add to the software stack or replace an older system

from the stack. Thus, creating managing tools that can manage only one specific system is unproductive effort as during system replacement the managing software needs to be rewritten. To avoid this issue, we adopted the approach of creating managing applications that are not specific to the system they manage, but specific to the class of systems they manage - e.g., tools that can manage Relational Database Management Systems (RDBMSs), rather than tools specific to a single RDBMS like PostgreSQL or Oracle.

1.1 Monitoring

The Monitoring category contains tasks that i) collect monitoring data and ii) detect problems on the production systems. We define *Monitoring Data* as any metadata related to the state of the system that can help either to detect or diagnose a problem. Depending on the information included in the monitoring data, we further divide it into three types: i) performance, ii) utilization and iii) statistics. For example, for a DataBase Management System (DBMS), performance data consists of metrics such as response time of a query, throughput of the database system, I/O read/write response time; utilization data - cache buffer used/free space, percent cpu utilization; and statistics data - number of cache hits, number of issued I/Os, number of records returned by a query or an operator. Tasks that detect problems on the production system are responsible for raising an alarm when a problem occurs. We define *Problem* as something causing the production system to violate the Service Level Agreements (SLAs) or behave erratically.

There are a lot of tools that collect monitoring data such as Ganglia, IBM TotalStorage Productivity Center and Hyperic. However, all of these tools only collect monitoring data and their problem detection part is limited to a certain metric exceeding a certain threshold. This limitation restricts the ability to monitor the health of the production system which partly explains the fact that most of the problems

are reported by the user of the systems. Being able to detect problems is a hard task especially if it has to be done in a proactive manner (detect the problems before they affect the users). Next, we outline some of the challenges involved in the automation of this category for detecting problems.

Collecting monitoring data consists of continuously taking samples of the production system metrics (e.g., I/O rate, buffer hits) at certain points in time defined by the monitoring time interval (the rate at which the samples should be collected). Depending on the number of metrics monitored, each sample can have a measurable impact on the production deployment. Furthermore, the impact on the system is also affected by the rate at which these samples are taken. The number of metrics per sample and the rate of sample collection are proportionally related to the overhead on the production deployment. This relation outlines the first challenge in this category: What should be the rate at which to collect monitoring data and what metrics to monitor? The more data is collected, the better information there will be to detect and diagnose a problem; but it will have high overhead, so a sweet spot needs to be determined where the monitoring data is enough and introduces minimum overhead.

As mentioned earlier, currently available tools have a problem detection mechanism that is limited to detecting whether one metric or a set of metrics exceed specified thresholds. This mechanism cannot be used to detect correctness problems. For example, when a user requests a particular data set, only half of the data set is returned or a wrong data set is returned. This observation outlines the next challenge: How to automatically detect problems that cannot be expressed as relations in the conventional monitoring data? Another challenge is: How to detect problems in a proactive manner? (That is, to identify the problems before the user detects them.) Proactively detecting a problem and fixing it before the system user

sees it is the best possible scenario for enhancing the user experience, and to also give an “always up and running system” feeling.

1.2 Diagnosing

The Diagnosing category includes tasks that i) analyze the monitoring data and ii) identify a root cause for a problem. These tasks require the SAs to exercise all of their experience and skills. First, SAs start with filtering the monitoring data and identifying the data relevant to the observed problem. Then they drill down into the data, and every component that it is related, to better understand what causes the problem. Sometimes, to be able to do the analyzes, the SAs need to replicate the environment and the conditions when the problem happened. This process may require creating a replica of the production system and doing experiments with the workload (see Section 1.3 for details of replicating the production environment). Furthermore, different system interactions are hard to replicate, which makes the analyzes even more challenging. Another factor that makes root cause identification challenging is the pressure applied on the SAs to resolve the problem in a timely manner (especially if the current problem has brought down the production system).

Automating this category requires tools that are able to filter and analyze the monitoring data. As discussed in the previous section, for performance reasons, the monitoring data might be collected in longer intervals and/or could include only a small set of metrics. Thus, filtering the monitoring data should be performed carefully, and the relevant data identified correctly. Having to analyze large amounts of data is infeasible for humans. Also, automation is not trivial due to complex relationships between the effect of a problem and its root cause. This observation outlines the next challenge: How to identify the monitoring data relevant to the observed problem?

After identification of the monitoring data relevant to the problem, the next step is to analyze the monitored system metrics and pinpoint a root cause or a set of root causes. Automation of this process requires propagating the SA's expertise to the diagnosis tool. The first challenge in this step is: How to express the SA's expertise to the diagnosis tool? Note that the expression of SA's expertise should be intuitive to the administrator so that they can provide the correct information. The diagnosis tools need to store this expertise and use it to analyze the problem. It is important that the tool can provide explanations for the identification of a root cause. SAs need to see the reasoning of the diagnosis tool to trust the tool's identified root cause. This fact outlines the next challenge in this step: How can diagnosis tools use the SA's expertise and provide an explanation mechanism?

1.3 Testing

The Testing category includes tasks that: i) set up and ii) run an experiment. Here, experiment is defined as any change that the SA wants to examine or to study the effect of that change. An important part of running an experiment is the environment in which this experiment is run. Running experiments on the production system could lead to serious issues like system crashes or performance degradation, thus it is recommended that experiments are run on separate resources. The first task is setup for an experiment. How close the setup environment is to the production deployment affects how dependable the results are. SAs prefer more dependable results as there will be fewer unexpected events when applying the change on the production system. Providing automation of the setup reduces human errors like running the wrong experiment or running an experiment in a wrong environment.

Another challenge in this category arises from the complexity of an experiment. The more complex an experiment is, the more side effects it might have. Thus when testing is performed, proper monitoring needs to be enabled to watch for any

side effects. This challenge is closely related to the similarity of the experimental environment to the production one.

One form of testing that requires changes to the production environment is *A/B testing*. For example, how would the current database workload have performed if index I_1 had (hypothetically) been part of the production database's configuration? Thus, the challenge is to create a replica of the production system with the new change(s) and obtain the performance of the system.

1.4 Integrated and Automated System Management

To help administrators with their cumbersome and time-consuming tasks, we developed tools that help automate the task categories presented in Sections 1.1 - 1.3. The tools need to provide the following characteristics in order for the SAs to trust and make the best use of them:

- **Intuitive and Easy to Use Interface:** To lessen the burden of the SAs, they need to be able to use the management tool with ease and not be buried in yet another complicated system.
- **Supporting a Range of Systems:** It is not practical for SAs to have large amounts of managing applications as it is hard to keep track of all of them and integrate the information together.
- **Complete View and Transparency:** If SAs do not trust a managing tool, they will not use it. Providing a complete view and explanation of the management task is crucial for the SAs to adopt the tool.
- **Proactive Responses:** Providing timely notifications to the SAs is essential for the managing tools as otherwise the SAs may not be able to take actions in a timely manner.

When developing the three managing tools that are covered in this dissertation, we considered all of the four SA considerations:

- **Amulet** belongs to the Monitoring category. Chapter 2 provides the details of this tool.
- **DiaDS** belongs to the Diagnosing category. Chapter 3 provides the details of this tool.
- **Flex** belongs to the Testing category. Chapter 4 provides the details of this tool.

2

Monitoring with Amulet

We start with our contributions in the first category: Monitoring. We developed a tool, called Amulet, that continuously monitors and proactively detects data corruption on the production system. Data corruption is an unfortunate byproduct of the complexity of modern systems and could be caused by hardware errors, software bugs, and mistakes by human administrators. The dominant practice to deal with data corruption today involves administrators writing ad hoc scripts that run data-integrity tests at the application, database, file-system, and storage levels. This manual approach is tedious, error-prone, and provides no understanding of the potential system unavailability and data loss if a corruption were to occur.

We developed Amulet - a tool that addresses the problem of verifying the correctness of stored data proactively and continuously. Amulet provides:

- a declarative language for SAs to specify their objectives regarding the detection and repair of data corruption
- optimization and execution algorithms to ensure that the SA's objectives are met robustly and with least cost, e.g., using pay-as-you-go cloud resources

- timely notification when corruption is detected, allowing proactive repair of corruption before it impacts users and applications.

2.1 Introduction

Data corruption—where bits of data in persistent storage differ from what they are supposed to be—is an ugly reality that database and storage SAs have to deal with occasionally; often when they are least prepared (e.g., CouchDB Data Loss, 2010; Treynor, 2011; Oracle Corrupted Backups, 2007; Krioukov et al., 2008; Subramanian et al., 2010; Zhang et al., 2010). Hardware problems such as errors in magnetic media (*bit rot*), erratic disk-arm movements or power supplies, and bit flips in CPU or RAM due to alpha particles can cause data corruption. Bugs in software or firmware as well as mistakes by human SAs are more worrisome. Bugs in the hundreds of thousands of lines of disk firmware code have caused corruption due to *misdirected writes*, *partial writes*, and *lost writes* (Krioukov et al., 2008). Bugs in storage software (Treynor, 2011), OS device drivers, and higher-level layers like load balancers (e.g., Brunette, 2008) and database software (CouchDB Data Loss, 2010) have caused corruption and data loss. Recent trends make data corruption more likely to occur than ever:

- Production use of fairly new data management systems: A bug in the CouchDB NoSQL system caused data loss because writes were not being committed to disk (CouchDB Data Loss, 2010). A recent bug (Treynor, 2011) triggered by a storage software update caused 0.02% of Gmail users to lose their email data (which had to be restored from tape).
- Use of large numbers of commodity “white-box” systems in datacenters instead of more expensive servers. The lower price comes from the use of less reliable hardware components that are more prone to corruption and failures (Bairavasundaram et al., 2008; Schroeder et al., 2009).

- More software layers due to virtualization and cloud services: Customers of the Amazon Simple Storage Service (S3) have experienced data corruption where the data they got back on reads was different from the data they had stored originally (Brunette, 2008).

For many companies, their data is the key driver of their business. Data loss can have serious consequences, even putting companies out of business. It took only one unfortunate instance of file-system corruption (which spread to data backups), and the consequent loss of data stored by users, to put the once popular social-bookmarking site Ma.gnolia.com out of business (Magnolia, 2009).

Most systems have a first line of defense to corruption in the form of detection and repair mechanisms. Storing checksums, both at the software and hardware levels, is a common mechanism used to detect corruption (e.g., Oracle HARD, 2004). Storing redundant data—e.g., in the form of error correcting codes (ECC) or replicas—as well as duplication of work—e.g., writing to two separate hosts—lowers the chances of data loss due to corruption from bit flips, partial writes, and lost writes. Despite these mechanisms, recent literature (Krioukov et al., 2008) as well as plenty of anecdotal evidence show that problems due to corruption happen, and more frequently than expected (Bairavasundaram et al., 2008; Schroeder et al., 2009). A particularly dangerous scenario that the authors as well as others (e.g., Magnolia, 2009; Oracle Corrupted Backups, 2007) have come across is as follows. The data on the production system gets corrupted, but the corruption goes unnoticed by users and SAs. In majority of scenarios, data corruption is only noticed/detected when affected data is accessed. As backups of the production data are taken, the corruption propagates to backups. A disaster strikes (Application detects corruption) the production system at some point, and the SA has to bring the system back online from a backup. However, the system keeps crashing when started from a corrupt backup. The SA

Table 2.1: Data integrity tests to detect and possibly repair data corruption at the application level

System	Test	Description	Does Repair	Runs Online
Microsoft Exchange Server	Eseutil	Uses checksums and structure rules (knowledge of logical schema and physical properties) to check the mailbox database for errors in messages, folders, or attachments; checks whether all data pages are correct and match their checksums. Eseutil is the primary check and recovery utility for Microsoft Exchange Server, its focus is on the jet database engine (Eseutil Tutorial).	Yes	No
	Isinteg	Performs similar checks as Eseutils, however this tool lacks the repair ability and its focus is on detecting corruption in the information store part of the Microsoft Exchange Server (Isinteg).	No	No
Any File	Tripwire	Creates a database with the hash and attributes of the content of all files in the system. Checks for mismatch between the current state of the files and the information stored in the database. Tripwire continuously monitors the system for file modifications and records all changes along with the complete information of what, where, when and who performed the change (Tripwire, 2008).	No	No
	Parchive (par & par2)	Uses an error correcting code to create parity data. Checks whether the given file's content matches its stored parity data (Parchive).	Yes	No

is forced to go long into the past to manually find a corruption-free backup; keeping the system down for a long time—seven days in one case we are aware of, and forever for Ma.gnolia.com—and losing many days of updates to the data.

Table 2.2: Data integrity tests to detect and possibly repair data corruption at the database level

System	Test	Description	Does Repair	Runs Online
Oracle	Health Monitor	Uses checksums and structure rules (metadata, logs and system rules) to check the database content, data pages, logs and system metadata. (Oracle Health Monitor)	No	Yes
	DBVerify	Uses checksums and structure rules (head & tail info) to check the database content and data pages. Can check backups and specific parts of the database. This tool verifies only the correctness of the stored data and does not check the logs (Costa et al., 2000; DBVerify; DBVerify Usage; Oracle Recover; Oracle Repair).	No	No

Continued on the next page

Table 2.2 – Continued from the previous page

System	Test	Description	Does Repair	Runs Online
	DBMS_Repair	Uses checksums and structure rules (metadata and system rules) to check and repair all table structures, marks data pages as corrupted if there is a checksum mismatch. This tool will repair corrupted table structures but will not recover data (DBMS Repair; Oracle Recover; Oracle Repair).	Yes	No
	RMAN	Performs backups and uses checksums and structure rules (metadata, logs and system rules) to find corrupted data pages and restored them from a backup (Oracle Recover; Vargas, 2008).	Yes	No
DB2	db2inspect, Inspect command	Uses checksums and structure rules (page headers, properties of records) to check the database, data pages, records and the database memory structures (DB2 Inspect Command; db2inspect Tool; Inspect vs db2dart).	No	Yes
	db2dart	Uses checksums and structure rules (page headers, properties of records) to check and repair the database, data pages, and records (db2dart Tool; Inspect vs db2dart).	Yes	No
	Check Data, Check Index	Use checksums and structure rules (table and index metadata) to check the table data pages, index data pages and index entries point to the correct records (DB2 Check Utilities).	No	Yes
	db2ckbkp	Uses checksums and structure rules (metadata and page headers) to check the data page consistency in a backup. Determines if a backup is valid for restore operation (db2ckbkp Tool).	No	No
Microsoft SQL Server	Checkdb, Check-table	Use structure rules (page headers, properties of indexes) to check and repair storage of the data pages and index entries point to the correct records (SQL Server checkDB).	Yes	No
	Restore	Uses backups to restore and recover the full database content, tables, data pages and logs (SQL Server Restore Command).	Yes	No
MySQL	myisamchk	Uses checksums and structure rules (flags, table metadata) to check and repair data pages, records and indexes to point to correct records. This suite contains 5 distinct tests that do increasingly rigorous and time-consuming checks (myisamchk Command; Subramanian et al., 2010).	Yes	No
	mysqlchk	Uses checksums and structure rules (flags, table metadata) to check and repair data pages, records and indexes to point to correct records. Works against all storage engines, but its functionality is limited by the storage engine API (mysqlcheck Command; Subramanian et al., 2010).	Yes	Yes

End of Table 2.2

Table 2.3: Data integrity tests to detect and possibly repair data corruption at the file-system level

System	Test	Description	Does Repair	Runs Online
XFS	xfs_check	Uses the file-system's journal (log of operations done) and verifies the integrity of the inode hierarchy and that the content of the inodes is in sync with the stored file-system data. Checks superblock, free-space maps, inode maps and data blocks (Fixing XFS).	No	No
	xfs_repair	Uses the file-system's journal (log of operations done) to repair the integrity of the inode hierarchy and the connections between the the content of the inodes and the stored file-system data. Repairs superblock, free-space, and inode maps (Fixing XFS).	Yes	No
Ext3 & Ext4	fsck	Uses the journal (in ext4) and file structures (in ext3) to check the superblock, file pathnames, data block connectivity, and the file and inode reference counts (Admin's Choice, 2009).	Yes	No
	Stellar Phoenix Recovery	Uses the journal (in ext4) and file structures (in ext3) to recover lost files and folders, and the superblock content and placement (Stellar Phoenix Linux Data Recovery).	Yes	No
FAT* & NTFS	ScanDisk	Uses file structures to check the disk headers, FAT media, file content, file attributes, lost disk space, incorrect free space, internal clusters, and file crosslinks (Quirke, 2002).	Yes	No
	Chkdsk	Uses file structures and the physical structure of a disk to check the file content, file attributes, free disk space, and file crosslinks (Laurie, 2011).	Yes	No
ZFS	zpool scrub	Uses built-in block checksums and block replication mechanisms in the ZFS file-system to check file content for errors. ZFS built-in redundancy (ranging from mirroring to RAID-5) with the transaction Copy-on-Write (COW) mechanism are used to repair block corruptions (Hablador, 2009; Cromar, 2011; Sun Microsystems ZFS, 2006; Zhang et al., 2010).	Yes	Yes

End of Table 2.3

Table 2.4: Data integrity tests to detect and possibly repair data corruption at the hardware level

System	Test	Description	Does Repair	Runs Online
Hard Disk Drive	Scrubbing	Uses checksums stored at the level of disk media blocks to verify that each block's content matches its checksum. Each hardware vendor has a specific implementation of the scrubbing algorithm (Schwarz et al., 2004).	No	No
RAID	Scrubbing	Uses stored parity information or replicas at the RAID level (instead of per disk) to verify the content of each data block (Krioukov et al., 2008).	Yes	Yes

Thus, systems have developed a second line of defense in the form of data-integrity tests (hereafter, *tests*). A test: (a) performs checks in order to detect specific types of data corruption, and/or (b) repairs specific types of data corruption. Just like software bugs, prevention is better than cure in the case of data corruption. However, prevention is not always possible. Tables 2.1, 2.2, 2.3 and 2.4 list popular tests for detecting and repairing corruption that occur in different systems as well as in different system layers such as storage, file-system, and database. Tests have the following characteristics:

- Tests perform more sophisticated detection and repair of corruption than is possible automatically during regular system operation through mechanisms like checksums and RAID (Subramanian et al., 2010).
- Barring few exceptions, tests have been developed to be run offline when the system is not serving a workload. If a workload changes the data concurrently with a test execution, the test may detect (and worse, fix) spurious corruptions. The workload could also return incorrect results because of modifications made by the test. As one example, it is recommended that the file-system be unmounted while running the *fsck* test.
- Most of the tests are very resource-intensive.

Because of the above characteristics of tests, database and storage SAs often struggle with questions on when and where to run tests. If the SA is not proactive in running tests, then, when corruption strikes eventually, high system downtime and data loss (and possibly, loss of the SA's job) will result.

SAs usually have specific objectives in mind for proactive detection and repair of data corruption. Table 2.5 gives examples of such objectives. To our knowledge, no system today helps SAs specify objectives like these easily, and automates the

Table 2.5: Examples of objectives that a SA may have regarding timely detection and repair of data corruption

	Description of Example Objectives in English
1	If the <i>myisamchk</i> test detects corruption in the <i>lineitem</i> table in my MySQL OLTP DBMS, then I want to have immediate access to an older corruption-free version of the table that is less than 1 hour old.
2	(A security vulnerability patch was applied in the ext4 file-system that my production DBMS is using. I am afraid that the patch may inadvertently cause data corruption.) Run the <i>fsck</i> file-system test at least once every hour. Notify me immediately of any corruption detected.
3	My production DBMS runs on an Amazon EC2 m1.large host. I have the same objectives as in 1, but I am willing to spend up to 12 dollars per day for additional resources on the Amazon cloud to meet these objectives. How recent of a corruption-free version of the data can I have immediate access to if a corruption were to be detected?
4	My objectives are a combination of 1 and 2, but I want the time intervals to be 30 minutes instead of 60. I am cost conscious. What minimum number of m1.small EC2 hosts should I rent to run tests?

nontrivial task of running tests to meet these objectives. The result is usually a convoluted mix of ad hoc scripts and testing practices with nobody having a clear idea of the downtime and data loss a potential corruption can cause. In an informal survey we conducted among SAs of clients of a large company, a pressing need to improve the current state of testing was expressed.

2.1.1 Amulet: Challenges and Overview

A typical *database software stack* that production systems use is shown in Figure 2.1. Different levels of the software stack maintain different sources of data. All these data sources have to be kept corruption-free to guarantee correct behavior, good performance, and availability of applications running on the software stack.

The database level has data in tables as well as plenty of metadata such as indexes, materialized views, and information in the database catalog. Databases store their data and metadata as files and directories in a file-system or directly as blocks on *volumes*. The file-system level has files containing data stored by the database level, as well as metadata such as the directory structure, *inodes* (indexes storing file-to-

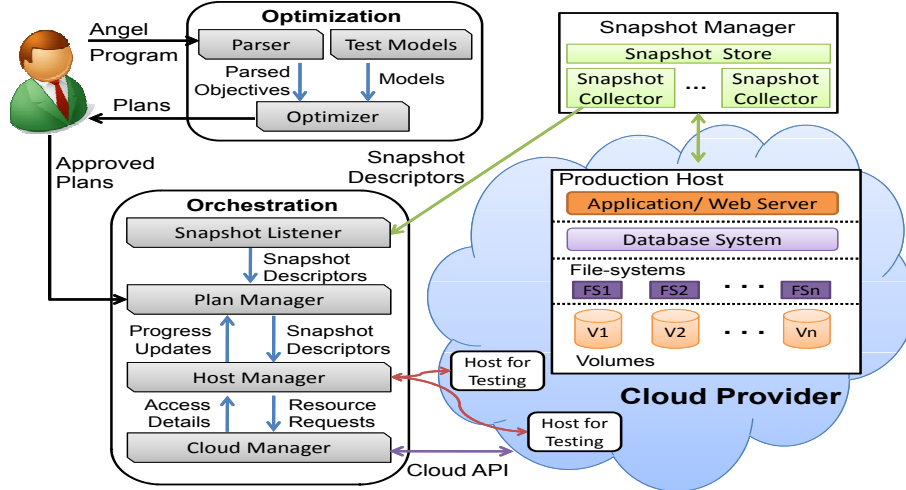


FIGURE 2.1: Overview of Amulet’s optimization and orchestration

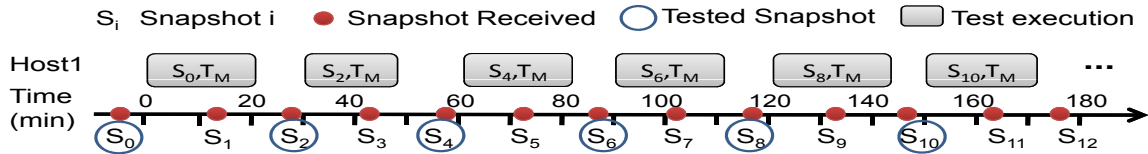


FIGURE 2.2: Actual execution timeline, in minutes, of a testing plan in Amulet for Example 1 from Table 2.5. Each box denotes a run of the *mysamchk* (T_M) test on the respective snapshot S_i . The horizontal width of each box corresponds to the test execution time.

block mappings) and *journals* (log of operations done). A file-system, in turn, stores its own data and metadata on a volume. A volume provides an interface to read and write blocks of data. Beneath this interface, the volume may be a physical block device (e.g., a hard disk or solid state drive) or a logical entity (e.g., representing storage on a networked server or a combination of partitions from multiple hard disks).

Proactive Testing for Data Corruption: Tests are run to verify the correctness of data. Tables 2.1, 2.2, 2.3 and 2.4 list commonly-used tests at each level of the database software stack. For example, MySQL’s *mysamchk* suite contains five different tests invoked through distinct invocation options: fast, check-only-changed, check (default), medium-check, and extended-check. These tests apply checks of increas-

ing sophistication and thoroughness to verify the correctness of tables and indexes in the database. The checks include verifying page-level and record-level checksums as well as verifying that each index entry points to a valid record in the corresponding table, and vice versa. The `fsck` and `xfs_check` tests verify the correctness of metadata and data in the `ext3` and `XFS` file-systems respectively. For example, they ensure consistency between the file-system journal and the data blocks, and verify that all the data block pointers in the inodes are correct.

The first challenge that Amulet faces is how to run a test automatically. Most of the tests in Tables 2.1, 2.2, 2.3 and 2.4 cannot be run concurrently with the regular workload on the production system because of performance and correctness problems. The tests can consume significant CPU or I/O resources. The tests may also have to lock large amounts of data, making response times for the production workload slow and unpredictable. Amulet addresses these problems using the following three-step approach to run tests automatically:

1. *Create snapshot:* A *snapshot* is a persistent copy of a point-in-time version of the data needed for a test. Snapshots can be taken at the database, file-system, or volume levels. In this work, we focus on volume-level snapshots because they capture the data needed for any test in the software stack. While the time to create the first snapshot will depend on the volume size, later snapshots only need to copy the changes since the last snapshot (similar to incremental backups).¹
2. *Run tests:* A snapshot is loaded on to one or more *testing hosts* where tests are run. As shown in Figure 2.1, testing hosts are different from the production host to avoid performance problems.

¹ Production deployments that need near-real-time disaster recovery take snapshots regularly and store them on cloud storage (Wood et al., 2010).

3. *Apply changes*: If tests detect and repair corruption in a snapshot, then the administrator can choose to apply these changes or load the repaired snapshot on the production system.

Amulet’s Declarative Language: Making it easy and intuitive for administrators to declare objectives like those in Table 2.5 poses a nontrivial language design problem. A dissection of the examples in Table 2.5 reveals the important features that are needed:

- Specification of one or more tests t , and associating t with the data and type of resources on which it should be run.
- Specification of *tested recovery points* to be maintained over a recent window of time. These points enable quick system recovery in case serious corruption is detected.
- Ability to declare different objectives such as the minimum number of test runs per time window or a cost budget for provisioning pay-as-you-go resources to run tests.
- Ability to combine multiple objectives as well as to specify an optimization objective.

Angel, Amulet’s declarative language, is designed to support such features. The semantics and simplified syntax of *Angel* are described in Section 2.4 and Table 2.7.

Amulet’s Optimization Phase: Amulet can run a comprehensive suite of tests, including new user-defined ones, to detect and possibly repair data corruption anywhere in the software stack. To use Amulet, as shown in Figure 2.1, a user or application submits a declarative *Angel* program that references one or more volumes on the production system. For each volume V , the program specifies: (a) the

tests to be run on data contained in V , and (b) the objectives to be met. For volume V , Amulet’s *Optimizer* will generate an efficient execution strategy—called a *testing plan*—using an optimization algorithm that maximizes or minimizes one objective subject to satisfying all other objectives. Amulet’s *Orchestrator* will execute the testing plan automatically and continuously by provisioning testing hosts and scheduling tests on a resource provider.

Figure 2.2 shows an actual execution timeline of a testing plan P for an Angel program corresponding to Example 1 from Table 2.5. Plan P uses one testing host that runs the `mysamchk` test on snapshots taken from the production host. One snapshot is tested every 30 minutes, and each test takes around 20 minutes to complete. As we will see in Section 2.7, this plan minimizes execution cost while meeting the objective of continuously maintaining a tested recovery point for a past 1-hour window. This testing plan, while simple, illustrates a number of challenges facing Amulet.

Characterizing the testing plan space: A testing plan has multiple aspects. First, there is a provisioning aspect that determines how many testing hosts are used to meet the specified objectives. Second, there is a scheduling aspect that determines the rate at which snapshots are tested and how test runs are scheduled on the provisioned hosts. Third, there is a sustainability aspect that determines whether the plan will continuously meet the specified objectives as time progresses. Section 2.5 gives a formal characterization of a testing plan in Amulet, thereby defining the space of testing plans.

Developing a cost model for tests: To find whether a plan enumerated from the testing plan space will meet the objectives specified in an Angel program, the *Optimizer* needs models to estimate the execution times of tests scheduled by the plan. A novel component of Amulet is a library of models to estimate test execution times.

The library currently covers tests for the MySQL database and the ext3 and XFS file-systems; discussed further in Section 2.3.

Finding a good testing plan: For each volume referenced in an Angel program, the Optimizer has to find a good plan from a huge plan space. We propose a novel algorithm for this optimization problem that considers all three aspects of testing plans: provisioning testing hosts, scheduling tests on snapshots and hosts, and ensuring plan sustainability over time. While our algorithm is not guaranteed to find the optimal plan, we show empirically—based on comparisons with an exhaustive search algorithm—that our algorithm is very efficient and finds the optimal plan most of the time.

Amulet’s Orchestration Phase: After submitting an Angel program, the administrator can view the testing plans generated, and when satisfied, submit the plans to Amulet’s Orchestrator for execution. The Orchestrator executes testing plans continuously by working in conjunction with a Snapshot Manager and a resource provider, both of which are external to Amulet. The Snapshot Manager notifies the Orchestrator when a new snapshot of a volume on the production system is available for testing. The Orchestrator allocates testing hosts from the resource provider which, currently, can be any infrastructure-as-a-service cloud provider. A major challenge faced by the Orchestrator is in dealing with unpredictable events arising during plan execution:

- *Repairs:* It is impossible to predict when a corruption will be detected and a repair action needs to be taken.
- *Straggler hosts:* A host used to run tests on the cloud may become slow temporarily, causing the test execution schedule to lag behind the optimizer-planned schedule.

- *Wrong estimates:* Lags in the testing schedule can also be caused by inaccurate estimates of test execution times from the models.

Rather than complicating the Optimizer or making unrealistic assumptions, Amulet’s solution is to reserve a cost budget in each testing plan that the Orchestrator can use to provision additional hosts on demand to deal with unpredictable events; discussed in Section 2.6. The novel effect is that a testing plan has a statically-planned component generated by the Optimizer as well as an adaptive component managed by the Orchestrator. Section 2.7 will present comprehensive experimental results from a prototype of Amulet running on the Amazon cloud.

2.2 Related Work

A number of recent empirical studies show that corruption of critical data is a reality and occurs much more commonly than assumed previously. It is perhaps surprising that the database research community has paid little attention to this problem.

2.2.1 The Dangers of Data Corruption

Bairavasundaram, Goodson, Schroeder, Arpaci-Dusseau, and Arpaci-Dusseau analyzed corruption instances recorded in more than 10,000 production and development storage systems. Their main focus was on studying *silent data corruption* which is corruption undetected by the disk drive or by any other hardware component. Among corruption instances logged over a 41-month period among a total of 1.53 million disk drives of various types, the authors found more than 400,000 instances of checksum mismatches. The study also showed that cheaper nearline SATA disks (and their adapters) develop checksum mismatches an order of magnitude more often than the more expensive and carefully-engineered SCSI disks. However, corruption can also occur in the latter which are enterprise-class drives.

Schroeder, Pinheiro, and Weber analyzed memory errors collected over a period of 2.5 years in the majority of servers used by Google. The authors found that the rate of data corruption in DRAM is orders of magnitude higher than previously reported, with more than 8% of dual in-line memory modules (DIMMs) affected by errors per year. Memory errors can be classified into soft errors, that corrupt bits randomly but do not leave physical damage; and hard errors, that corrupt bits in a repeatable manner because of a physical defect. Memory errors found in the study were dominated by hard errors, rather than soft errors as assumed previously.

Injecting faults into the database software stack provide insights into system behavior and data loss under different types of corruption. A recent study used fault injections into a popular open-source DBMS (MySQL) to show that certain types of data corruption can harm the system, e.g., causing system crashes, data loss, and incorrect results (Subramanian et al., 2010). The authors also point out that concurrency control and recovery features of database systems are not designed to detect or repair corrupted data or metadata resulting from hardware, software, or human errors. A similar study has been done for the ZFS file-system that, compared to popular Linux file-systems like ext3 and XFS, has novel features like *end-to-end checksums* for corruption detection (Zhang et al., 2010). The authors show that while ZFS is very resilient to disk-level corruption, memory-level corruption can lead to crashes and incorrect results.

2.2.2 Dealing with Data Corruption

The techniques categorized as the first line of defense in Section 2.1 check for data correctness during reads and writes in the production workload; usually based on additional stored information like parity bits and checksums. These techniques are not sufficient to prevent or detect corruption caused by complex issues such as lost and misdirected writes due to bugs in the software stack (Bairavasundaram et al.,

2008). For example, Krioukov, Bairavasundaram, Goodson, Srinivasan, Thelen, Arpaci-Dusseau, and Arpaci-Dusseau show the inability of techniques like parity-based RAID to avoid data corruption. The authors also show how common techniques used in RAID can spread data corruption across multiple disks and cause data loss. The first line of defense adds performance overheads during workload execution. Graefe and Stonecipher present a technique for an online B-tree verification with minimum overhead. However, enterprise systems have historically preferred performance over the (wrongly assumed) rare chance of data corruption. Amulet addresses these problems by enabling complex and resource-intensive tests like those in Tables 2.1, 2.2, 2.3 and 2.4 to be run in a timely fashion. To our knowledge, Amulet is the first system of its kind that works across different point-in-time copies of data to detect and repair data corruption efficiently in the end-to-end software stack.

System designers take different approaches to tackling data corruption. Some systems are designed resilient, i.e., end-to-end checksums, parity/checksum verifications at every level in the system stack in the datapath, where as others rely on offline detection tools or utilities. Modern file-systems like ZFS fall under the former category. Most enterprise systems fall under the latter category, sacrificing integrity in favor of performance. Due to proprietary nature of these applications, detection and correction utilities are also tightly controlled by the vendors. Section 2.3 and Tables 2.1, 2.2, 2.3 and 2.4 provided a fairly comprehensive listing of tests used in systems today.

Modeling the performance of tests or improving their efficiency has received little attention. For example, most tests still use single-threaded execution and cannot exploit multicore CPUs. Currently, every test t is an opaque execution script to Amulet apart from the model used to estimate t 's execution time. With more visibility into tests, Amulet can do a better job of optimizing t 's execution. A promising

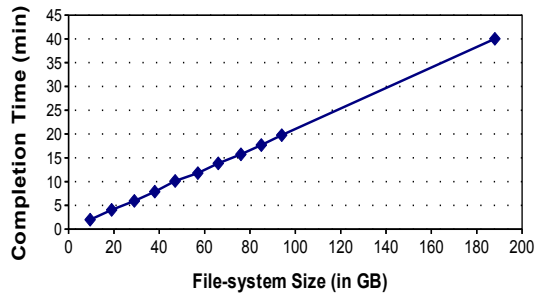
development in this regard is the writing of tests in declarative languages like SQL as done by Gunawi et al. (2008).

Amulet’s goal of early detection and repair of data corruption forms a crucial part of disaster recovery planning. Wood, Cecchet, Ramakrishnan, Shenoy, van der Merwe, and Venkataramani argue that cloud computing platforms are well suited for offering disaster recovery as a service due to (a) the cloud’s pay-as-you-go pricing model that can lower costs, and (b) the cloud’s use of elastic virtual platforms. Amulet is a proof-of-concept system for this argument applied to the problem of data corruption.

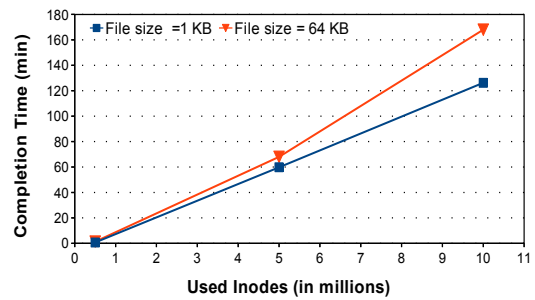
The concept of declarative system management is gaining currency. Chef, Puppet, and Microsoft SQL Server’s policy-driven manager are now popular tools that take declarative specifications as input, and then configure and maintain systems automatically (Guo et al., 2009). However, unlike Amulet, these tools do not support objectives that are specified declaratively and optimized automatically.

2.3 Modeling of Tests

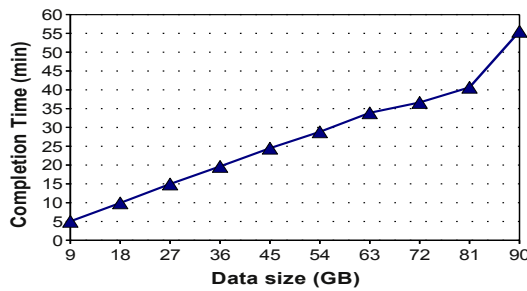
For each test t , Amulet’s Optimizer needs a model to estimate t ’s run-time behavior—e.g., execution time, usage of CPU, memory, and I/O resources—when t is run on given data and system resources. We divide the input parameters for a test model into three categories: (i) data-dependent attributes, (ii) resource-dependent attributes, and (iii) attributes to capture transient effects. In this section, we discuss attributes in the test model and their impact for the `myisamchk`, `fsck`, and `xfstest` tests from Tables 2.2 and 2.3. Our focus is on models for estimating test execution time. Note that we generate separate models for the same test when invoked with significantly different options. For example, `fsck` has separate models based on whether it is invoked to check file-system metadata compared to data. The `fsck` metadata test



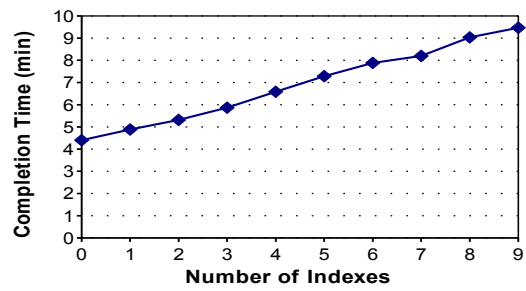
(a) Varying the file-system size for fsck test on ext3



(b) Varying the number of used inodes for xfs_check on XFS



(c) Varying the input data size for myisamchk medium test



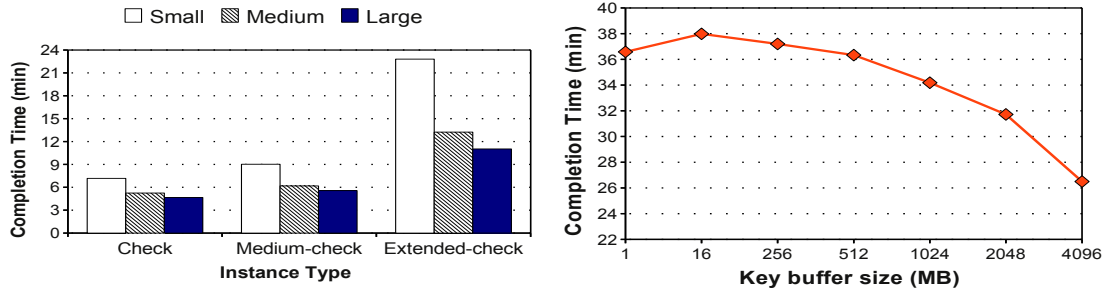
(d) Varying number of indexes for myisamchk medium test

FIGURE 2.3: Effect of data-dependent attributes on the completion time of the fsck, xfs_check, and myisamchk tests

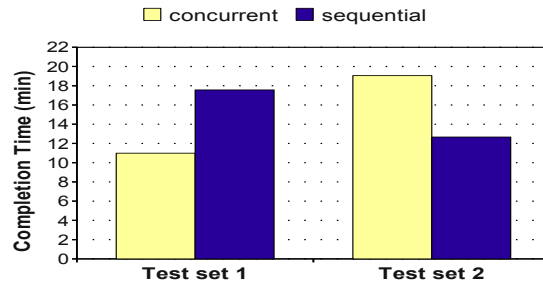
involves verifying the superblock, inodes, and free block list, while the data test does a full scan to find all bad blocks.

2.3.1 Data-Dependent Attributes

Data-dependent attributes have a first-order impact on test execution times. Fortunately, this impact can be captured fully based on properties that correspond to the size of the data and are easily measurable. Different data-dependent attributes are relevant depending on whether the test is at the application, database, file-system, or storage levels. For fsck and xfs_check, we explored a wide range of attributes to capture the file-system data: size of the file-system, total number of inodes, number of used inodes, average file size, files per directory, and size of inodes and data blocks. Based on a comprehensive empirical analysis, we found that there are three



(a) Effect of Amazon EC2 instance type on myisamchk tests for a 14GB database (b) Effect of MySQL's key_buffer_size parameter for myisamchk extended test



(c) Concurrent vs sequential execution of myisamchk tests w/ diff. invocation options

FIGURE 2.4: Effect of resource-dependent attributes and concurrent test execution on the execution time of myisamchk tests

attributes with the most impact: total inodes, used inodes, and average file size. Total inodes represents the total available capacity of the file-system, while used inodes are indicative of the used space.

The block check version of fsck on the ext3 file-system verifies the correctness of the used portion of the file-system as well as that of the unused inodes and free blocks. Thus, as shown in Figure 2.3a, the execution time of the fsck data test depends on the file-system size. In contrast, the optimized xfs_check test for the XFS file-system verifies only the used portion of the file-system. Thus, the execution time of xfs_check depends on the number of used inodes as shown in Figure 2.3b, and is independent of the total file-system size.

The average file-size plays a role on the test run-time – a large file has multiple in-direct data pointers in the file inode requiring random block access. As xfs_check

is much more efficient compared to fsck, a significantly larger number of used inodes was used in those experiments.

For the myisamchk tests, the total input data size to the test is the primary data-dependent attribute. Figure 2.3c shows the linear effect of the input data size on test execution time. In addition to the data size, the number of indexes present on table columns affects the execution time of certain myisamchk tests. Once again, the effect is linear as shown in Figure 2.3d.

In summary, the data-set attributes used for both database and file-system models are general properties that can be easily calculated for the data-set to be tested.

2.3.2 Resource-Dependent Attributes

The execution time of a test is expected to vary with the hardware and software resources allocated to run the test. The trend among infrastructure-as-a-service cloud providers is to provide hardware resources such as CPU, memory, and storage in terms of a small number of discrete choices. For example, the hardware resource space is discretized into micro, small, medium, and large node configurations on the Amazon cloud. Such discretization vastly simplifies Amulet’s task of modeling test execution times for varying resource properties. Given the discrete resource types, Amulet can generate samples to train models by running tests on the different resource types.

Figure 2.4a shows the execution times of myisamchk tests for different Amazon EC2 host types for a total database size of 14 GB. We observe a 100% speedup for the myisamchk extended test between a small host (1.7 GB memory with 1 EC2 compute unit) and a large host (7.5 GB memory with 4 EC2 compute units). In general, we observed that the more thorough myisamchk tests are both compute- and memory-intensive. Execution times of fsck did not vary significantly across the different Amazon EC2 host types. fsck is I/O-intensive and does not exploit the

increased CPU or memory resources when going from the small to the large hosts.² Further increase of the throughput did not improve the run-times since the test is limited by the fsck parallelism.

Software-level resources like caches can also impact test execution times. For example, the `myisamchk` tests use a cache called the *key buffer*. The size of this cache, given by the `key_buffer_size` input parameter, affects the execution time of certain `myisamchk` tests. Figure 2.4b shows the variation in the execution time of the `myisamchk` extended test for input data of size 12 GB.

2.3.3 *Transient Effects*

Test execution times can vary due to transient effects like warm versus cold caches or resource contention when tests are run concurrently. A major advantage Amulet enjoys in this context is that Amulet’s Orchestrator can ensure that tests are run in configurations similar to the ones used during test modeling. Furthermore, as we discuss next, Amulet’s Optimizer prioritizes predictable behavior (i.e., *robust testing plans*) over potential optimality since the former is more important in proactive testing.

Warm Vs. cold caches: Data caching for the memory/disk interface within a host does not benefit tests such as `fsck` and `xfs_check` that access data directly at the block level. However, as mentioned before, such caching benefits higher-level tests such as `myisamchk`. Data caching at the host/network interface while using networked storage benefits most tests. We have observed up to 2x differences in test execution times for warm Vs. cold caches in this context. The test models have been enhanced to account for these effects.

² We did manage to improve the execution time of the `fsck` data test by 40% using software-RAID-like techniques on the Amazon cloud.

Concurrent execution of tests on the same host: Concurrent execution makes test execution times difficult to estimate. Figure 2.4c illustrates this complexity. Test sets 1 and 2 are runs of two `mysamchk` tests on two different tables. The tests in each set lead to different outcomes when run in concurrent versus sequential fashion. The tests in set 2 cause memory thrashing when run concurrently. Since the number of distinct tests is not large,³ we can train models to estimate execution times for tests run concurrently in pairs. As such, if Amulet’s Optimizer does not have a model to estimate the running time of concurrent tests, it will simply choose not to consider testing plans with concurrent tests. The goal of the Optimizer is to find a robust testing plan, where a robust plan is one whose performance is almost never much worse than promised.

Time-of-day effects: External workload on the IaaS provider may influence the test completion time. However, for our test bed with Amazon we did not observe any significant variation in the test run-times as a function of time-of-day.

In summary, there are only a few popular tests, so test models can be built once and reused. As Amulet controls the resource on which tests are run, the tool can ensure that tests are run in similar settings to the one used for the modeling. Furthermore, the discretization and isolation provided by the cloud environment significantly helps limit the attribute permutations in the models.

2.4 The Angel Declarative Language

This section and Table 2.7 summarizes the main statements in the Angel⁴ declarative programming language. An Angel program specifies tests as well as the objectives

³ From our experience, most SAs prefer to use standard tests that come with each system, rather than writing new tests.

⁴ Angel script was a form of writing used by the philosopher-king Solomon in amulets he designed for various life’s problems (King Solomon’s Amulets)

Table 2.6: Notation used in this chapter

Name	Description
O_1-O_n	SSO, RPO, SIO, TCO, or CO objectives specified per volume in an Angel program (Table 2.7 gives a summary)
O_{opt}	Optimization objective for a volume in an Angel program
$t, s, h, \tau, x,$ and d	Used to denote respectively tests, volume snapshots, hosts, time intervals, numeric constants, and currency values
τ_{rpo}	The time interval in an RPO (see Table 2.7)
d_{co}	The maximum cost budget in a CO (see Table 2.7)
P	Testing plan for a volume in an Angel program
P_W	Plan P 's window. The plan repeats every P_W time units
P_I	Time interval between successive snapshots in plan P
P_M	Test-to-snapshot mapping in plan P
P_S	Schedule of test execution in plan P
P_R	P 's cost budget reserved to handle unpredictable events
$ExecTime(t)$	Execution time of test t
$Time(s_i)$	Time when i^{th} snapshot s_i , $1 \leq i \leq \frac{P_W}{P_I}$, is available in plan P relative to the start of the plan window P_W
$start(h)$	Time when host h is first used in the plan window
$end(h)$	Time when host h will finish its last scheduled test for the plan window ($end(h)$ can be $> P_W$)
$cost(P_S),$ $cost(h)$	Cost incurred for the plan schedule P_S or a host h for one plan window

to be met while running these tests. Figure 2.5, which we will use for illustration in this section, shows an Angel program for Example 1 in Table 2.5.

Tests: Angel's **Test** statement defines a test t by specifying the command to run t as well as references to t 's input data (specified by a **Data** statement) and the type of host on which to run t (specified by a **Host** statement). The **Test lineCheck** statement in Figure 2.5 defines the *myisamchk medium* test for MySQL from Table 2.2. Angel's **Repair** statement enables a repair action to be associated with a test t for invocation if t detects a corruption (as indicated by a specific return code from t).

Data: Angel's **Data** and related statements define the input data for a test, including the volume that the data belongs to, the data type (from a set of supported types), and the data properties. The properties, which are specific to the type of data, form inputs to the models that the Optimizer uses to estimate test execution times. The

Table 2.7: Summary of important Angel statements. $Op \in \{\leq, \geq\}$. t , τ , x , and d are constants of respective types test, time interval, numeric, and currency

Name	Simplified Specification Syntax
Test	<code>Test(Data: data, Host: host, exec scripts, ...)</code>
Input data for test	<code>Data(Volume: V, type, properties, ...)</code>
Host to run test	<code>Host(type, setup scripts, ...)</code>
Volume	<code>Volume(Host: production host where V is located, path on host, volume id, properties, ...)</code>
Repair action	<code>Repair(Test: t, t's return code, exec scripts, ...)</code>
SSO: Safe Snapshot Objective	List of tests $\{t_1, t_2, \dots, t_k\}$, for volume V
RPO: Recovery Point Objective	<code>Recovery_point</code> $\leq \tau$, for volume V
SIO: Snapshot Interval Objective	<code>Snapshot_interval</code> $Op \tau$, for volume V
TCO: Test Count Objective	<code>Test_count(t)</code> $Op x$, in time interval τ , for volume V
CO: Cost Objective	<code>Cost</code> $\leq d$, in time interval τ , with reservation $x\%$, for volume V
O_{opt} : Optimization Objective	Maximize (when Op is \geq in an SIO or TCO), Minimize (when Op is \leq in an RPO, SIO, or CO)
Notification	SQL triggers on event tables in log database

Optimizer does semantic checks to ensure that the Angel program specifies values for all input parameters required by the model for each test in the program. While helper tools are available to extract these values from the corresponding input data, the SA can also specify values based on their domain knowledge.

Hosts: The primary use of Angel’s `Host` statement is to define a host type (from a set of supported types) for a test t so that the Orchestrator guarantees that t will always be run on hosts of that type. Figure 2.5 shows 2 host definitions (*prodHost* and *testHost*). The current implementation of the Orchestrator supports all the EC2 host types on the Amazon cloud.

2.4.1 Objectives

An Angel program can specify one of five types of objectives. These objectives can be used independently or combined together to specify a variety of requirements for running tests. Each objective O references a unique volume. The Optimizer will

```

RecoveryPointObjective rpo{
  timeInterval: 60,
  volume: vol1
}

SafeSnapshot safe {
  test: lineCheck
  volume: vol1
}

Test lineCheck {
  host: testHost,
  command: myisamchk,
  data: dataLine,
  parameters: --medium-check
  after: scripts/result.sh
  before: scripts/prepare.sh
}

Repair repair1 {
  test: lineitemsCheck
  exitCode: 1
  repair: scripts/repair.sh
}

Volume vol1 {
  mapTo: /volume
  device: /dev/sdn
  volumeID: vol-XXX
  host: prodHost
  filesystem: fs1
  sizeInGB: 50
}

FileSystem fs1 {
  type: xfs
  freeze: `xfs_freeze -f <mountDir>`
  unfreeze: `xfs_freeze -u <mountDir>`
}

Database db1 {
  volume: vol1
  start: scripts/mysql_start.sh
  connect: mysql
  stop: scripts/mysql_stop.sh
  freeze: `flush tables with read lock`
  unfreeze: `unlock tables`
}

Data dataLine {
  Type: TABLE,
  database: db1,
  parent: vol1,
  specifier: lineitem.MYI
  dataSize: 24672
  indexes: 0
}

Host prodHost {
  image: ami-48aa4921
  type: m1small
  user: root
  setup: scripts/host_setup.sh
  isProduction: true
}

Host testHost {
  image: ami-48aa4921
  type: m1small
  user: root
  setup: scripts/host_setup.sh
  isProduction: false
}

Access loginDetails {
  accountID: XXX
  loginPK: ec2
  accessKey: XXXX
  secretKey: XXXX
  securityGroups: ssh
}

```

FIGURE 2.5: Angel program for Example 1 in Table 2.5

partition the objectives in an Angel program based on the volumes referenced, and generate one testing plan per volume.

Safe Snapshot Objective (SSO): An SSO for a volume V specifies a list of tests t_1, \dots, t_k that Amulet must run on every snapshot s given for V by the Snapshot Manager. Note that a volume-level snapshot will contain the input data needed for any test. (The Optimizer checks to ensure semantic consistency across all the statements in a program.) If none of the tests t_1, \dots, t_k find corruption on s , then Amulet will label s as corruption-free. If a repair action is associated with a test t , then Amulet will run the repair on snapshot s if t were to detect a corruption in s .

Recovery Point Objective (RPO): When one or more tests detect corruption in a snapshot of a volume V , it is useful to have another snapshot of V in the recent past where the corruption did not exist. This objective usually arises from recovery needs. If a corruption causes an application to crash or misbehave, then the SA may have to find a recent *recovery point* quickly (Wood et al., 2010). A recovery point

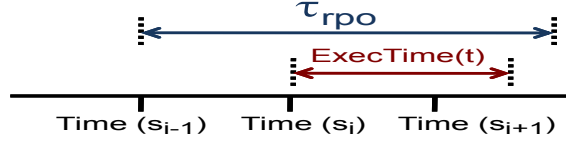


FIGURE 2.6: Illustration of RPO

is a corruption-free snapshot from which the application can be brought back online quickly.

Figure 2.6 shows two successive snapshots s_{i-1} and s_i for a volume on which a test t is run. If a corruption were to be detected when t runs on s_i , then the SA wants s_{i-1} to be a recovery point. An RPO expresses this requirement. RPO specifies the maximum (sliding) time interval τ_{rpo} into the past within which a recovery point must exist for a volume if corruption were to be detected on a snapshot. Amulet should ensure that the sum of test t 's expected running time and the snapshot interval between s_{i-1} and s_i is not greater than τ_{rpo} (as illustrated in Figure 2.6). Using notation from Table 2.6, the RPO mandates:

$$ExecTime(t) + Time(s_i) - Time(s_{i-1}) \leq \tau_{rpo} \quad (2.1)$$

An RPO together with an SSO with a list of tests t_1, \dots, t_k is a powerful combination to express recovery points. Now Amulet should ensure that all of t_1, \dots, t_k will finish on s_i in time $\leq \tau_{rpo} - (Time(s_i) - Time(s_{i-1}))$. If any of these tests were to detect a corruption in s_i , then s_{i-1} will serve as a recovery point that is not more than τ_{rpo} into the past. At this point, the SA or the Snapshot Manager can decide how to proceed regarding the production system: restart the system with snapshot s_{i-1} , ignore the corruption for now, etc. Amulet does not interfere with these policies.

Snapshot Interval Objective (SIO): For a volume V , an SIO has the form: `Snapshot_interval Op τ` , for time interval τ and `Op` in $\{\geq, \leq\}$. `Snapshot_interval` refers to the expected time interval between two successive snapshots of V . Note that

Amulet does not control the Snapshot Manager which collects snapshots from the production system. SIOs express the feasible snapshot intervals that Amulet should consider during plan selection.

Test Count Objective (TCO): For a volume V , a TCO has the form: `Test_count(t) Op x` , in a time interval τ for a test t . Here, `Test_count` specifies the number of unique snapshots of V in the interval τ on which Amulet should run test t . The typical use of TCOs are to express requirements of the form: “Run t at least four times every day.” A plan chosen by the Optimizer for this TCO will do the intuitive thing of spacing out the four test runs uniformly in the specified time interval of 1 day.

Cost Objective (CO): A CO for a volume V has the form: `Cost $\leq d$` , in a time interval τ , with (optional) reservation x . Here, d is a cost measure for the resource provider from which the Orchestrator will allocate resources to run tests. Costs may be specified in real or virtual currency units depending on the provider. The CO applies to the entire testing plan that the Optimizer generates for V . For example, a CO for the Amazon cloud could be (see Table 2.5): “The testing plan can use up to 12 U.S. dollars per day.” A CO can be specified only if the *pricing model* that the resource provider uses to charge for resource usage is input to Amulet. Section 2.7 describes the pricing model that Amulet uses for the Amazon cloud.

The CO specifies a fraction of the overall cost budget that is reserved for the Orchestrator to respond to three types of unpredictable events that can occur during the execution of the testing plan: repairs, straggler hosts, and inaccurate predictions of test execution times by the models used in the Optimizer (recall Section 2.1.1). Section 2.6 will discuss how the Orchestrator uses the reserved cost budget to provision hosts on demand to handle these unpredictable events. When Amulet is used on the Amazon cloud, by default, the reservation is set to the cost of one host that can run all the tests specified in the Angel program.

Find the Least-Cost Plan $P=\langle P_W, P_I, P_M, P_S, P_R \rangle$ that Satisfies the n Objectives O_1-O_n (without O_{opt}) for a Volume V in an Angel Program

1. Use Figure 2.8 to pick the snapshot interval P_I for O_1-O_n ;
2. Select the plan window P_W , and scale O_1-O_n to P_W (Section 2.5.3);
3. Use Figure 2.9 to pick the test-to-snapshot mapping P_M for O_1-O_n, P_W, P_I ;
4. Use Figure 2.10 to pick the test execution schedule P_S for P_W, P_I, P_M ;
5. P_R is available from the CO in O_1-O_n , or a default is used;

FIGURE 2.7: Finding the least-cost plan for given objectives O_1-O_n

Optimization Objective (O_{opt}): An appropriate **Maximize** or **Minimize** optimization objective (O_{opt}) can be specified along with an RPO, SIO, TCO, or CO for a volume V in an Angel program. **Maximize** can be used when Op in the objective is \geq , and **Minimize** can be used when Op is \leq . Specifically, applying **Maximize** to an SIO or TCO of the form **value** \geq *const* asks to make *const* as high as possible. Applying **Minimize** to an RPO, SIO, or CO of the form **value** \leq *const* asks to make *const* as low as possible. One and only one O_{opt} is allowed per volume. The default is **Minimize** CO if no O_{opt} is specified in the program.

Notifications: Users can request notifications when certain events or sequences of events occur when a plan is executed by the Orchestrator. A comprehensive logging framework in the Orchestrator logs a range of events of interest, including snapshot collection, start and completion of tests, detection of corruption, application of repairs, and lags in the testing schedule. The logs are persisted into a database system which runs continuous queries to provide the notifications specified in the Angel program. Our current implementation uses the PostgreSQL database system and provides notifications using triggers.

2.5 Optimizer

In this section, we will discuss the algorithm used by Amulet's Optimizer. Given an Angel program, the Optimizer first partitions the objectives in the program based on

the volumes referenced, and then selects one testing plan per volume. The selection of the testing plan is treated as the following optimization problem:

Testing-plan Selection Problem for Volume V : *Given n objectives O_1 - O_n (each of type SSO, RPO, SIO, TCO, or CO) and an optimization objective O_{opt} (of type Maximize or Minimize on one of O_1 - O_n) for a volume V , find the testing plan (if any) that meets all of O_1 - O_n while giving the best (maximum or minimum, as appropriate) value for O_{opt} .*

We will first describe the structure of a testing plan generated by Amulet’s Optimizer, and then describe the various stages employed by the Optimizer as it attempts to find the optimal plan from the large plan space. In Section 2.7, we will compare our Optimizer with an *exhaustive optimizer* that works by systematically enumerating plans from this large plan space (and scales poorly because of this nature).

2.5.1 Testing Plans in Amulet for a Volume V

Formally, a testing plan P contains five components:

1. *Snapshot interval P_I* is the uniformly-spaced minimum time interval between consecutive snapshots that the plan needs to test to meet all the objectives specified.
2. *Window P_W* is a time interval such that the plan repeats every P_W time units. The plan processes $\frac{P_W}{P_I}$ snapshots per window.
3. *Test-to-snapshot mapping P_M* specifies, for each snapshot s in the plan window, the set of tests that need to be run on s .
4. *Test execution schedule P_S* specifies the number and respective types of testing hosts to use, and when to run each test from P_M on these hosts.

5. *Reserved cost budget* P_R is the part of the plan’s total cost budget that is reserved for the Orchestrator to deal with unpredictable events that can arise during plan execution.

The core of Amulet’s Optimizer is a *cost-optimal* planning algorithm that can find the minimum-cost plan (if valid plans exist) to meet a given set of objectives. We will begin in Sections 2.5.3–2.5.5 by describing the stages in which the cost-optimal planning algorithm works. As illustrated in Figure 2.7, each stage selects one of the five components of the minimum-cost plan, going in the order P_I , P_W , P_M , P_S , and P_R . Section 2.6 will describe how the Orchestrator uses the reserved cost budget P_R .

If the Angel program’s optimization objective is not cost minimization, then Amulet uses a higher-level planning algorithm, described in Section 2.5.6.

We use $ExecTime(t)$ to denote the running time of test t as estimated by the models available to the Optimizer. If all input parameters required by the model(s) for t have not been specified in the Angel program, then the Optimizer will catch this error during its semantic checking phase. Since the Optimizer reads the model for each test t from a model library, an Amulet user can override an existing model or supply a model for a new test they have defined. A model can be as simple as a function that always returns a constant value for $ExecTime(t)$ (e.g., representing t ’s running time for the host type specified in the program). With this feature, the SA can ask useful *what-if questions*, e.g., “how much additional cost will be needed to meet my objectives if test execution times were to increase by $x\%$?”.

2.5.2 *Selecting the Snapshot Interval*

The Optimizer’s goal in this stage is to pick the maximum value that P_I can have while meeting all the RPO, SIO, and TCO objectives specified. Maximizing P_I translates into minimizing the number of snapshots that need to be processed. Con-

Selecting the Plan Snapshot Interval P_I in a Testing Plan

Inputs: Objectives O_1 - O_n (with syntax from Table 2.7)

1. $P_I^{min} =$ Snapshot interval from the Snapshot Manager;
2. $P_I^{max} = \infty$; $\tau_{rpo} = \infty$;
3. if (O_1, \dots, O_n contains RPO: **Recovery_point** $\leq \tau$) {
4. $\tau_{rpo} = \tau$; $P_I^{max} = \frac{\tau_{rpo}}{2}$; /* test ≥ 2 snapshots in τ_{rpo} to meet RPO */}
5. for (every Objective O in O_1, \dots, O_n) {
6. if (O is SSO: List of tests $\{t_1, \dots, t_k\}$) {
7. for (Test t in t_1, \dots, t_k)
8. $P_I^{max} = \text{Min}[P_I^{max}, \tau_{rpo} - \text{ExecTime}(t)]$; } /* Equation 2.1 */
9. if (O is TCO: **Test_count**(t) $\geq x$, in time interval τ)
10. $P_I^{max} = \text{Min}[P_I^{max}, \tau_{rpo} - \text{ExecTime}(t), \frac{\tau}{x}]$; /* Equation 2.1 */
11. if (O is SIO: **Snapshot_interval** $\geq \tau$)
12. $P_I^{min} = \text{Max}[P_I^{min}, \tau]$;
13. if (O is SIO: **Snapshot_interval** $\leq \tau$)
14. $P_I^{max} = \text{Min}[P_I^{max}, \tau]$;
15. }
16. if ($P_I^{max} < P_I^{min}$) return “No feasible P_I exists for O_1 - O_n ”;
17. else set $P_I = P_I^{max}$;

FIGURE 2.8: Selection of P_I (notation used from Tables 2.6 and 2.7)

sequently, the cost of the plan is minimized—which is our goal—since more snapshots mean higher test execution and host requirements.

Figure 2.8 shows the steps involved in this stage. The algorithm goes through the objectives one by one, while maintaining an upper (P_I^{max}) and lower (P_I^{min}) bound on feasible values of P_I . Finally, the largest feasible value of P_I , if any, is selected.

2.5.3 Selecting the Plan Window

Recall from Section 2.4 and Table 2.7 that the objectives RPO, TCO, and CO for a volume V in an Angel program specify time intervals. The plan window P_W serves as a mechanism for the Optimizer to consider the intervals in all objectives in a uniform fashion. P_W is picked as the least multiple of P_I ($P_W = n \times P_I$, $n \in \mathbb{N}$) such that P_W is greater than or equal to the maximum among: (a) the time intervals in RPO, TCO, and CO objectives, and (b) $\text{ExecTime}(t)$ for each test t specified in an SSO

Selecting the Test-to-Snapshot Mapping P_M in a Testing PlanInputs: Scaled objectives O_1 - O_n , Plan Window P_W , Snapshot Interval P_I

1. for (every test t referenced in an SSO or TCO in O_1, \dots, O_n) {
2. $COUNT_t^{min} = 0$; $COUNT_t^{max} = \frac{P_W}{P_I}$; }
3. for (every Objective O in O_1, \dots, O_n) {
4. if (O is SSO: List of tests $\{t_1, \dots, t_k\}$) {
5. for (Test t in t_1, \dots, t_k) /* t has to run on all $\frac{P_W}{P_I}$ snapshots */
6. $COUNT_t^{min} = \text{Max}[COUNT_t^{min}, \frac{P_W}{P_I}]$; }
7. if (O is TCO: $\text{Test_count}(t) \geq x$, in time interval P_W)
8. $COUNT_t^{min} = \text{Max}[COUNT_t^{min}, x]$;
9. if (O is TCO: $\text{Test_count}(t) \leq x$, in time interval P_W)
10. $COUNT_t^{max} = \text{Min}[COUNT_t^{max}, x]$;
11. }
12. $P_M = \emptyset$;
13. for (every test t referenced in an SSO or TCO in O_1, \dots, O_n) {
14. if ($COUNT_t^{max} < COUNT_t^{min}$) return “No feasible P_M exists”;
15. else {
16. Map test t to $COUNT_t^{min}$ snapshots spread uniformly across the $\frac{P_W}{P_I}$ snapshots in the plan window. Add the mappings to P_M ; }
17. }

FIGURE 2.9: Selection of P_M (notation used from Tables 2.6 and 2.7)

or TCO objective. Picking $P_W > \text{ExecTime}(t)$ for all tests (i.e., Case (b) above) is needed to ensure the *sustainability* of schedules as we will explain in Section 2.5.5.

Once P_W has been determined, the corresponding parameters in all TCO and CO objectives are scaled proportionately to P_W . For example, a CO that specifies a cost budget (d_{co}) of U.S. \$10 in 1 hour, will be scaled to a cost budget of U.S. \$15 for $P_W = 1.5$ hours. Note that the time interval in an RPO (τ_{rpo}) is independent of P_W , and should not be scaled.

2.5.4 Selecting the Test-to-Snapshot Mapping

For the $\frac{P_W}{P_I}$ snapshots in a plan window, this stage decides which tests need to be run on which snapshots. Figure 2.9 shows the steps involved. For each test t specified in an SSO or TCO, the algorithm in Figure 2.9 maintains upper ($COUNT_t^{max}$) and

lower ($COUNT_t^{min}$) bounds on how many snapshots t should be run on. Test t is mapped at uniformly-spaced intervals to the minimum number of snapshots that t needs to be run on. Note that the tests in an SSO should be run on all $\frac{P_W}{P_I}$ snapshots (Lines 4-6 in Figure 2.9).

2.5.5 Selecting the Schedule of Test Execution

After the test-to-snapshot mapping P_M has been generated, the Optimizer selects the schedule as well as the minimum number of hosts needed for running these tests. This stage, whose steps are shown in Figure 2.10, is by far the most complex one in the Optimizer. Note that the Optimizer is only identifying a good schedule. The schedule will be executed—including actual host allocation and test runs on the resource provider—only after the testing plan is submitted to the Orchestrator.

The self-explanatory Lines 1-13 in Figure 2.10 give an overview of the greedy algorithm used to select the test execution schedule. The algorithm goes through the snapshots s_i in one plan window in order from $i=1$ to $i=\frac{P_W}{P_I}$, as well as the tests t_{ij} that have been mapped to s_i . A host h_k is identified to run t_{ij} on s_i in one of the three ways listed respectively in Lines 5, 7, and 9.

The first way (described in Lines 14-24) is by means of *test grouping*, where t_{ij} will be run concurrently with another test or group of tests on a host that has already been allocated to the plan. Line 16 comes from Amulet’s goal to generate a *robust plan* (Babcock and Chaudhuri, 2005), i.e., a plan whose chances of performing worse than estimated is low. If there is no model to estimate how the concurrent execution of a set of tests will perform, then the Optimizer takes the low-risk route of avoiding such executions.

Line 19 (similarly, Line 29) addresses the important issue of *schedule sustainability* which ensures that the plan generated for one window can be run continuously for multiple windows that come one after the other. Using notation from Table 2.6, let

Selecting the Schedule of Test Execution P_S in a Testing Plan

Inputs: Plan Window P_W , Snapshot Interval P_I , Test mapping P_M , Resource provider's pricing model $cost(\dots)$, and Plan cost budget d_{co}

1. $P_S = \emptyset$;
2. for (each of the $\frac{P_W}{P_I}$ snapshots s_i in P_M) {
3. for (each test t_{ij} mapped to snapshot s_i in P_M) {
4. for (each host h_k added so far to P_S and whose host type matches the host type needed to run t_{ij}) {
5. [Line 14] if (t_{ij} can be scheduled by grouping t_{ij} with an already-scheduled test t' on s_i and h_k) {
6. Update P_S to add the new Grouped(t', t_{ij}) test instead of t' on h_k .
Go to the next test; }
7. [Line 25] else if (t_{ij} can be scheduled on h_k after all currently-scheduled tests on h_k have completed) {
8. Update P_S to schedule t_{ij} on h_k ; Go to the next test; }
9. [Line 35] else if (t_{ij} can be scheduled on a new host h') {
10. Update P_S to schedule t_{ij} on h' ; Go to the next test; }
11. else return "Could not find a feasible schedule P_S for P_M ";
12. }}}
13. return "Minimum-cost P_S is now available for the testing plan";
14. **Function invoked from Line 5:** Check grouping of test t_{ij} with a (possibly grouped) test t' scheduled on snapshot s_i and host h_k {
15. t_{ij} can be grouped with t' if all four conditions (a)-(d) hold {
16. /* avoids risky plans */
17. (a) A model is available to estimate grouped execution times for the types of tests t' and t_{ij} ;
18. (b) The grouping runs the tests faster than running them serially:
 $ExecTime(\text{Grouped}(t', t_{ij})) < ExecTime(t') + ExecTime(t_{ij})$;
19. (c) The grouping will not violate RPO:
 $Time(s_{i-1}) + \tau_{rpo} \geq end(h_k) - ExecTime(t') + ExecTime(\text{Grouped}(t', t_{ij}))$;
20. (d) The schedule with grouping is sustainable:
 $P_W + start(h_k) > end(h_k) - ExecTime(t') + ExecTime(\text{Grouped}(t', t_{ij}))$;
21. } if (t_{ij} can be grouped with t' on h_k) {
22. $end(h_k) = end(h_k) - ExecTime(t') + ExecTime(\text{Grouped}(t', t_{ij}))$;
23. return true; }
24. else return false; }
25. **Function invoked from Line 7:** Check if test t_{ij} can be scheduled on host h_k after all currently-scheduled tests complete on h_k {
26. t_{ij} can be scheduled on h_k if all three conditions (e)-(g) hold {
27. (e) t_{ij} can be started on h_k before the next snapshot s_{i+1} arrives:
 $Time(s_{i+1}) > end(h_k)$; /*smoothing the load in P_W */
28. (f) The new schedule will not violate RPO:
 $Time(s_{i-1}) + \tau_{rpo} \geq \text{Max}[end(h_k), Time(s_i)] + ExecTime(t_{ij})$;

Continued on the next page

FIGURE 2.10: Selection of P_S (notation used from Tables 2.6 and 2.7)

```

29.     (g) The new schedule is sustainable:
         $P_W + start(h_k) > \text{Max}[end(h_k), Time(s_i)] + ExecTime(t_{ij});$ 
30.     }
31.     if ( $t_{ij}$  can be scheduled on  $h_k$ ) {
32.          $end(h_k) = \text{Max}[end(h_k), Time(s_i)] + ExecTime(t_{ij});$ 
33.         return true; }
34.     else return false; }

35. Function invoked from Line 9: Check if test  $t_{ij}$  can be scheduled on a new host  $h'$  to be
    added to  $P_S$ {
36.      $t_{ij}$  can be scheduled on  $h'$  if both conditions (h) and (i) hold {
37.         (h) The host type of  $h'$  matches the host type needed for  $t_{ij}$ ;
38.         (i) The new schedule will not violate plan  $P$ 's CO in the window:
             $cost(P) + cost(h') + P_R \leq d_{co};$ 
39.         if ( $t_{ij}$  can be scheduled on  $h'$ ) {
40.              $start(h') = Time(s_i); \quad end(h') = Time(s_i) + ExecTime(t_{ij});$ 
41.              $cost(P) = cost(P) + cost(h'); \quad$  return true; }
42.         else return false; }

```

FIGURE 2.10: Selection of P_S (notation used from Tables 2.6 and 2.7)

$start(h)$ denote the time (relative to the start of the plan window) when a host h is first used to run a test in the window. $end(h)$ denotes the corresponding time when host h will finish its last scheduled test for the window. ($end(h)$ can be greater than P_W .) For the schedule to be sustainable across multiple successive windows, we need:

$$P_W + start(h) > end(h) \tag{2.2}$$

This condition ensures that by the time host h is needed to run tests for a plan window, all tests scheduled on h for the previous plan window will have completed. In fact, tests scheduled on h for all past windows will have completed because our technique from Section 2.5.3 to select the window size P_W ensures that no test run will span more than two consecutive windows.

The second way (described in Lines 25-34) to schedule test t_{ij} in Figure 2.10 is to run t_{ij} on a host h_k after all tests currently scheduled on h_k complete. Apart from the standard checks for RPO violation (Line 28) and sustainability (Line 29), the Optimizer also checks (Line 27) whether t_{ij} can be started over s_i on h_k before

Find Best Plan for Objectives O_1-O_n and Optimization Objective O_{opt}

1. Plan $P =$ Least-cost plan from Figure 2.7 for O_1-O_n ;
2. if (no valid P found) return “No plan found for O_1-O_n and O_{opt} ”;
3. if (O_{opt} is minimize for a CO or maximize for an SIO in O_1-O_n) {
4. return “Found best plan P for O_1-O_n and O_{opt} ”;
5. } else if (O_{opt} is minimize for an RPO or SIO or maximize for a TCO) {
6. Use the following steps to create a new objective O^{new} {
7. if (O_{opt} is minimize for an RPO or SIO) {
8. O^{new} is `Snapshot_interval` $\leq \frac{P_W}{\frac{P_W}{P_I} + 1}$, where P_W and P_I are respectively
 plan window and snapshot interval in P ; }
9. else { O^{new} is `Test_count(t)` $\geq x+1$, where P satisfies `Test_count(t)` $\geq x$ for the
 TCO on which it has a minimize; }
10. }
11. Plan $P^{new} =$ Least-cost plan from Figure 2.7 for O_1-O_n, O^{new} ;
12. if (valid P^{new} found) { Set $P = P^{new}$; Go To Line 5 and repeat; }
13. else return “Found best plan P for O_1-O_n and O_{opt} ”;
14. }
15. else return “Unsupported optimization objective O_{opt} ”;

FIGURE 2.11: Linear search algorithm to find the best plan for a volume

the next snapshot s_{i+1} arrives. The aim here is to achieve a balanced test execution workload (to the extent possible) throughout the window.

If it is not feasible to schedule t_{ij} on a host that is already allocated, then the third way is to schedule t_{ij} on a new host added to the plan (described in Lines 35-42). The addition of a new host should not overshoot any cost budget specified (Line 38). This step uses the resource provider’s pricing model. Note that allocation of a new host to run t_{ij} will not violate schedule sustainability (Line 40) because $ExecTime(t_{ij}) \leq P_W$ from Section 2.5.3.

2.5.6 Handling Non-cost Optimization Objectives

So far we focused on finding a testing plan that minimizes cost while meeting all the given objectives. Amulet’s Optimizer can handle non-cost optimization objectives as well, and does so by repeatedly invoking the cost-optimal planning algorithm with increasingly stricter objectives until no valid plan can be found. We have developed

Find Best Plan using Binary Search for Objectives O_1-O_n and Optimization**Objective O_{opt}**

1. Plan P = Least-cost plan from Figure 2.7 for O_1-O_n ;
2. if (no valid P found) return “No plan found for O_1-O_n and O_{opt} ”;
3. if (O_{opt} is minimize for a CO or maximize for an SIO in O_1-O_n) {
4. return “Found best plan P for O_1-O_n and O_{opt} ”;
5. else if (O_{opt} is minimize for an RPO or SIO or maximize for a TCO) {
6. Lower Bound (LB) = 1; Upper Bound (UB) = P_W ;
7. if (O_{opt} is minimize for an RPO or SIO) {
8. $UB = \tau$, where τ is the time interval of the RPO or SIO objective }
9. if (O_{opt} is maximize for an TCO) {
10. $UL = x$, where x is the TCO, $\text{Test_count}(t) \geq x$ }
11. $middle = \frac{UB+LB}{2}$;
12. Use the following steps to create a new objective O^{new} {
13. if (O_{opt} is minimize for an RPO or SIO) {
14. O^{new} is **Snapshot_interval** $\leq middle$ }
15. else { O^{new} is **Test_count**(t) $\geq middle$ }
16. }
17. Plan P^{new} = Least-cost plan from Figure 2.7 for O_1-O_n, O^{new} ;
18. if (valid P^{new} found) {
19. Set $P = P^{new}$;
20. if (O_{opt} is maximize) { $LB = middle + 1$; }
21. else { $UB = middle - 1$; }
22. }
23. else { $UB = middle - 1$; }
24. if ($LB \leq UB$) { Go To Line 11 and repeat; }
25. else return “Found best plan P for O_1-O_n and O_{opt} ”;
26. }
27. else return “Unsupported optimization objective O_{opt} ”;

FIGURE 2.12: Binary search algorithm to find the best plan for a volume

two algorithms for non-cost optimization showed in Figure 2.11 and Figure 2.12. These algorithms differ in how the stricter objectives are generated: one algorithm does so in *linear* increments while the second algorithm uses a *binary-search* technique to improve efficiency.

Consider the objective of minimizing the interval τ_{rpo} in an RPO. (Example 3 from Table 2.5 has this optimization objective.) It emerges from Equation 2.1 that the way to achieve lower values of τ_{rpo} is by reducing the snapshot interval $P_I =$

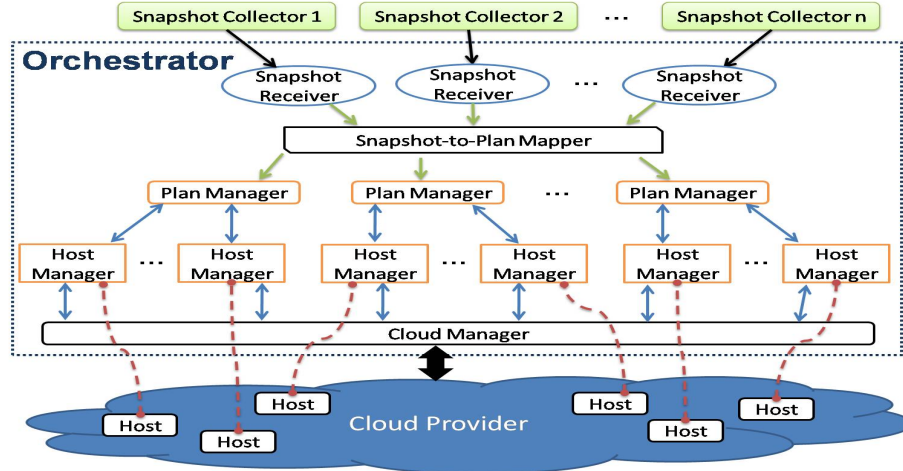


FIGURE 2.13: Amulet’s Orchestrator

$Time(s_i) - Time(s_{i-1})$. ($ExecTime(t)$ cannot be changed for the host type specified by the Angel program to run test t .) Given a current valid plan P , this rationale can be used to check whether lowering the snapshot interval in P to add one or more snapshots to the plan window will still give a valid testing plan P^{new} . This process of creating stricter objectives continues until no valid plan can be found; at that point, the minimum feasible τ_{rpo} is known, namely, it is the RPO interval in the valid plan found for the previous set of objectives.

2.6 Orchestrator

Recall from Section 2.1 and Figure 2.1 that testing plans are submitted to Amulet’s Orchestrator for execution. The Orchestrator will execute each submitted plan continuously by working in conjunction with the Snapshot Manager and resource provider. Figure 2.13 shows the multi-threaded design of the Orchestrator which has three concurrent execution paths—snapshot management, host management, and plan management—that we will discuss next.

Snapshot Management: The external Snapshot Manager (see Figure 2.1) informs the Orchestrator about the availability of a new snapshot s by sending a

descriptor for s . Snapshots are never copied to the Orchestrator. Since s is at the level of a volume V , the *Plan Manager* in charge of the testing plan for V is notified. In turn, the *Host Managers* responsible for hosts allocated to this plan from the external resource provider get notified. Recall that the plan generated by the Optimizer was based on $\frac{P_W}{P_I}$ uniformly-spaced snapshots per plan window. The Plan and Host Managers determine which snapshot in the window, if any, s should be mapped to.

Host Management: A Host Manager is responsible for using and monitoring a host allocated to a testing plan from the resource provider. Amulet’s implementation supports any infrastructure-as-a-service cloud provider (e.g., Amazon Web Services, Joyent, Rackspace) as the resource provider by using an appropriate *Cloud Manager* (Figure 2.13). The Host Manager uses the API provided by the Cloud Manager to allocate, establish connections with, and terminate hosts as well as to load snapshots on to allocated hosts.

Plan Management: A Plan Manager is responsible for shepherding the execution of a testing plan P through one or more plan windows until P is terminated. The Plan Manager’s role is straightforward at the conceptual level if P behaves as the Optimizer estimated when P was generated. The challenge is when the Plan Manager has to deal with unpredictable lags in the actual schedule of execution from the Optimizer-estimated schedule, and with repair actions that need to be run when corruption is detected.

Dealing with Lags: This process involves two steps: (i) identifying *straggler hosts* on which the lag is observed; and (ii) allocating one or more *helper hosts* for each straggler host subject to the reserved cost budget P_R earmarked for the Orchestrator to deal with unpredictable events. A testing host h is marked as a straggler when two conditions hold:

1. The actual execution time of tests for a snapshot s on h has overshoot the corresponding estimated time by more than an allowed slack. (The slack is used to prevent overreaction.) Straggler hosts are prioritized based on the age of s .
2. The next snapshot s' on which host h is scheduled to run tests has become available.

Each helper host h' for a straggler host h will take a share of h 's workload adaptively. The helper host h' will be terminated if, on completing the execution of tests on a snapshot, it is found that the corresponding testing host h is no longer a straggler.

Handling Repairs: This process also involves two steps:

1. The first test t that detects a corruption on a snapshot s , and has an associated repair action, will cause a *repair host* to be allocated to run the repair. Repairs for any future tests that detect corruption on s will be run on the same host in order to generate a single fully-repaired snapshot. Note that applying repairs offers much less scope for parallel execution compared to running tests to detect different types of corruption.
2. Once the repairs complete, a snapshot is taken to preserve the repaired version of s , and the repair host is terminated.

When a new helper or repair host is needed, the Plan Manager checks whether it has enough remaining budget from P_R to allocate a new host. If not, the Plan Manager will repeat the check at frequent intervals. In the worst case—e.g., if estimates of test execution times from models were significantly lower than actual execution times—an RPO or TCO objective will eventually get violated before a host can be

allocated. In that case, the Orchestrator will terminate the plan and send a notification to the SA.

2.7 Experimental Evaluation

To the best of our knowledge, no other tool supports the functionality that Amulet provides. So, we evaluated Amulet for correctness and we present a couple of use cases along with insights from the developing and testing process.

2.7.1 *Experimental Methodology and Setup*

Methodology: We have implemented Amulet with all the components and algorithms as described in the previous sections. We now present a comprehensive evaluation of Amulet when run using the Amazon Web Services platform as the infrastructure-as-a-service cloud provider. Section 2.7.2 considers the end-to-end execution, with both optimization and orchestration, of Angel programs in Amulet. For ease of exposition, we consider four scenarios that are simple in terms of Amulet’s functionality, but illustrate the challenges that Amulet has to deal with. Amulet’s power will become more clear in Section 2.7.3 where we consider both the efficiency (how fast?) and effectiveness (how good is the selected plan?) of the Optimizer while generating testing plans for huge plan spaces.

Database software stack and tests: For the production system, we choose a popular database software stack composed of MySQL as the database system, XFS or ext3 as the file-system, and Amazon’s *Elastic Block Storage (EBS)* volumes for persistent storage (we used 50GB volumes) (Running MySQL on Amazon EC2 with EBS). For each layer of the stack, we chose a representative test from Table 2.2: `myisamchk` for MySQL database integrity checking, and `fsck` and `xfcheck` for file-system integrity checking. Execution-time estimation models for these tests were trained and validated on the Amazon cloud (see Section 2.3).

Snapshots and storage: We implemented a Snapshot Manager that automates periodic volume-level snapshots (currently, the only type supported by the Amazon cloud). When the XFS file-system is used, the Snapshot Manager freezes the MySQL database as well as the XFS file-system (all caches are flushed to the disk) before a volume-level snapshot is taken (Running MySQL on Amazon EC2 with EBS). This process finishes within seconds. For the ext3 file-system, only the database is frozen since ext3 does not support the freeze feature of XFS.

Amazon provides two persistent storage services: Simple Storage Service (S3) and Elastic Block Storage (EBS). EBS provides much faster data access rates than S3, but has lower redundancy. Amazon supports snapshots of EBS volumes with the caveat that these snapshots are stored in S3. Specifically, when the Snapshot Manager initiates a snapshot, Amazon copies the EBS volume data to S3. All but the first snapshot request to the same EBS volume will copy to S3 only the changed data since the last snapshot.

Amazon does not provide direct access to data in a snapshot s . Instead, Amulet can create an EBS volume from s , and attach this volume to a testing host h that needs to run tests on s . This process copies data in a background fashion from the snapshot stored in S3 to h —prioritizing block read/write requests from h —making the volume accessible in h before the data movement is complete. Snapshot creation and restore times depend on bandwidth constraints and the amount of data that needs to be copied from S3 to EBS or EBS to S3. In our experiments, we observed an average bandwidth of 20 MB/s in both directions. This process can be made much faster by removing the intermediate copy to S3, which is part of our future work.

Plan costs: Recall from Section 2.4.1 that a pricing model for the resource provider has to be input to Amulet in order to specify cost objectives in Angel programs. For evaluation purposes, we used a pricing model motivated by how

Table 2.8: Pricing model used in our evaluation

Resource Used	Pay-as-you-go Pricing Method
Testing hosts	Hosts of the Small type (see Figure 2.4a) cost \$0.085 per hour. Medium / Large types cost \$0.17 / \$0.34 per hour respectively
Storage	\$0.10 per month per 1 GB of persistent storage used
I/O to storage	\$0.10 per 1 million block I/O requests to persistent storage
Snapshot access	\$0.05 per 1000 store or load requests for snapshots

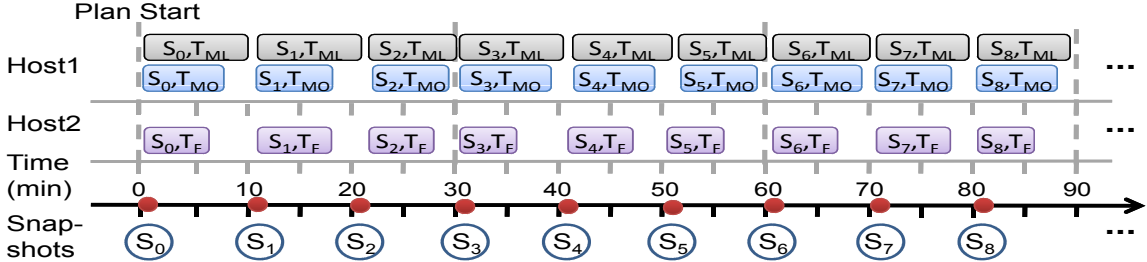


FIGURE 2.14: Actual execution timeline on the Amazon Cloud for Case 2 in our evaluation

resource usage is charged in a pay-as-you-go fashion on the Amazon cloud (Amazon Web Services). Table 2.8 outlines this pricing model in terms of how the four main types of resources used in a testing plan are charged. Given a testing plan P , Amulet’s Optimizer will use the pricing model to find how much P ’s use of each resource will cost in one plan window; and add all the per-resource costs to estimate P ’s total cost per plan window. The total number of block I/O requests to persistent storage is computed based on the total input data size for each test and the file-system’s block size. This strategy was chosen based on our empirical observations. Enhancing each test model to estimate the number of I/O requests that the test will make is part of our future work.

2.7.2 End-to-End Processing of Angel Programs

Case 1: Maintaining a Tested Recovery Point

Angel program: We first submit to Amulet the Angel program corresponding to Example 1 from Table 2.5. The program specifies two objectives, an RPO and an

SSO, for a single volume. The time interval τ_{rpo} in the RPO is 60 minutes. The SSO specifies a single test: a myisamchk medium test (denoted T_M) on a database table of size approximately 10 GB with no indexes. The test has to be run on hosts of type *Small* (m1.small on the Amazon cloud). The Snapshot Manager sends snapshot descriptors announcing new snapshots to Amulet every 15 minutes on average.

Testing Plan from the Optimizer: The model for estimating T_M 's execution time returns an estimate of 20 minutes when invoked by the Optimizer for this setting. The minimum-cost plan P generated by the Optimizer's algorithm in Figure 2.7 for this setting has:

- $P_I = 30$ minutes
- $P_W = 60$ minutes ($\frac{P_W}{P_I} = 2$ snapshots s_1 and s_2 per window)
- P_M consists of test T_M mapped to both snapshots in P_W
- P_S assigns one Small host to run T_M on s_1 and s_2
- Since no CO is specified, P_R takes the default value which in this case is the cost of one Small host (see Section 2.4.1)

Orchestration Timeline: Figure 2.2 shows the actual execution timeline of plan P when it is submitted to and run by the Orchestrator on the Amazon cloud. The meaning of each important symbol used in the figure is described at the top. The execution of P is shown for three plan windows, i.e., a total of 3 hours. When P is submitted, the Orchestrator starts by requesting the needed testing host from the cloud provider. When the host is available, the Orchestrator starts the plan execution (0 minutes in the timeline in Figure 2.2).

The Orchestrator is continuously executing the schedule P_S given by the Optimizer for each plan window. As part of this process, the Orchestrator (actually the

Plan and Host Managers from Section 2.6) has to map the snapshots s_i , $1 \leq i \leq \frac{P_W}{P_I}$, identified by the Optimizer in the plan window to actual snapshots S_j collected by the Snapshot Manager. Notice from Figure 2.2 that the Snapshot Manager is submitting snapshot descriptors every 15 minutes on average, while the snapshot interval P_I in the plan is 30 minutes.

At $Time(s_i)$, $1 \leq i \leq \frac{P_W}{P_I}$, in the plan window, the Orchestrator checks whether a snapshot S is available from the Snapshot Manager; if so, S will be tested. Otherwise, the Orchestrator waits for a slack interval to see whether a new snapshot is submitted. If no new snapshot arrives, then the last submitted snapshot will be tested. If this snapshot has already been tested, then an error notification is generated. The Host Manager uses P_S to find out whether it has to load a submitted snapshot (and if so, which tests it needs to run on the snapshot and how). Notice from Figure 2.2 that every other snapshot submitted is tested. The boxes with notation $S_j T_M$ in Figure 2.2 denote the actual run of test T_M on snapshot S_j submitted by the Snapshot Manager. The width of each box is the actual execution time of the test. These times are in the 18-22 minutes range which matches the estimated execution time of 20 minutes.

Case 2: Multiple Tests and Multiple Objectives

Angel program: We now consider a more complex Angel program with more tests as well as more and stricter objectives. The program specifies an RPO, an SSO, and an SIO for a single volume. The time interval τ_{rpo} in the RPO is reduced to 30 minutes from before. The SSO specifies three tests: two myisamchk medium tests respectively on a 2.4 GB lineitem table and a 1 GB orders table (with no index on either table), and an fsck metadata test. All tests have to be run on Small hosts. The SIO specifies `Snapshot_interval` ≤ 10 minutes. The Snapshot Manager sends snapshot descriptors announcing new snapshots to Amulet every 10 minutes on average.

Testing Plan from the Optimizer: The minimum-cost plan P generated by the Optimizer is more complex than before:

- $P_I = 10$ minutes
- $P_W = 30$ minutes ($\frac{P_W}{P_I} = 3$ snapshots s_1, s_2, s_3 per window)
- P_M has all three tests mapped to all three snapshots in P_W
- P_S assigns one host to run the fsck test (estimated to run in 6 minutes). The two myisamchk tests are run concurrently on a second Small host, with estimated times of 9 and 7 minutes.
- P_R takes the default value as in Case 1

Orchestration Timeline: Figure 2.14 shows the actual execution timeline of the above plan P . Note that $P_I = 10$ minutes causes all submitted snapshots to be tested. T_{ML} denotes the myisamchk test on lineitem, T_{MO} denotes the myisamchk test on orders, and T_F denotes the fsck test. While there is some variance in the actual execution times of the grouped test (1-2 minutes deviation from the estimates), the plan works as the optimizer estimated.

Case 3: Dealing with Unpredictable Lags

Here we run the same Angel program as in Case 2. Thus, the same plan P is picked and run as in Section 2.7.2. However, in this case, we cause a problem on $Host_1$ which causes the T_{ML} test on the host to run almost $2x$ slower than expected. Figure 2.15 shows the actual execution timeline.

Notice that test T_{ML} on snapshot S_4 now takes around 16 minutes to run compared to the estimated time of 9 minutes. The Plan Manager will mark $Host_1$ as a straggler because the two conditions for stragglers from Section 2.6 get satisfied at around 52 minutes in the timeline. $Host_1$ has overshoot the estimated time, and

the next snapshot S_5 on which $Host_1$ has to run tests is ready. (A slack interval of 2 minutes is used.) The Plan Manager will use the reserved cost budget P_R to request a helper host at time 52 minutes. The helper host is available at time 54 minutes, and takes over the running of the T_{ML} and T_{MO} tests on snapshot S_5 from $Host_1$. These tests complete at time 63. At that point, a check reveals that $Host_1$ is no longer a straggler host; thus, the helper host is released back to the resource provider. Intuitively, the helper host was pulled in adaptively to help the plan tide over a transient problem.

Case 4: Corruption Detection and Repair

In this case, we cause a data corruption in the production system that manifests itself in two snapshots. We create a scenario where a corruption happens due to a software bug that does a misdirected write on the production system. We run the Angel program from Case 2 with one change. The modified program associates repair actions with the `myisamchk` tests on the `lineitem` and `orders` tables. The repair actions invoke `myisamchk` with the “-r” option.

Figure 2.16 shows the actual execution timeline in this case. We inject the misdirected write in the interval between 30 and 40 minutes in the timeline, which corrupts the data in the `lineitem` table. Snapshots S_4 and S_5 submitted by the Snapshot Manager contain this corruption. The corruption will be detected by T_{ML} when executed on S_4 (S_4, T_{ML} in Figure 2.16). The corruption is reported to the Plan Manager which uses the reserved cost budget P_R to request a repair host. The repair action is run on the repair host. When the repair finishes around time 60, a snapshot of the repaired data is taken and the Snapshot Manager is notified. Around that time, the Plan manager gets notified of the corruption detected in S_5 . The repair on S_5 is run on the same repair host, and the repaired snapshot is available at time 66. Since no pending repairs exist at this point, the repair host is released.

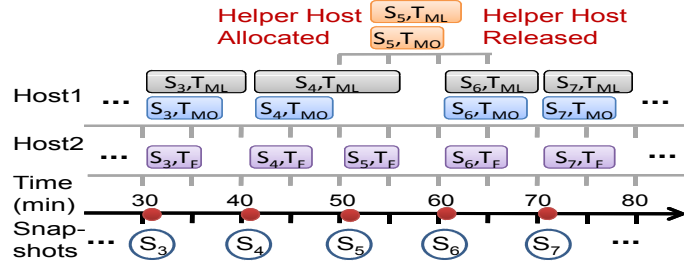


FIGURE 2.15: Actual execution timeline for Case 3

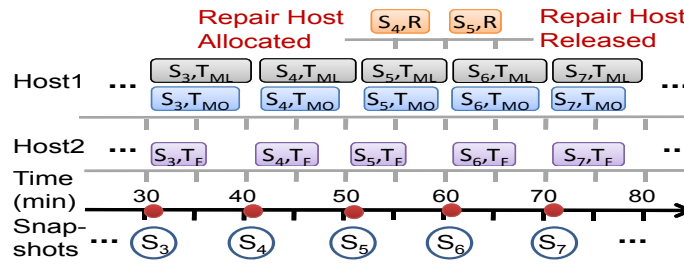


FIGURE 2.16: Actual execution timeline for Case 4

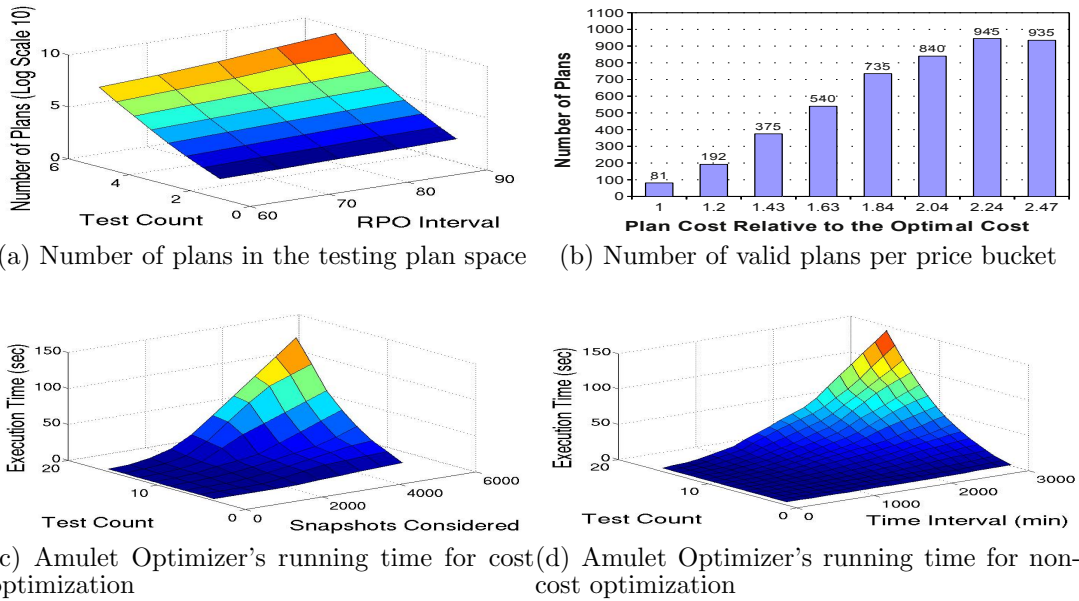


FIGURE 2.17: Characteristics of the testing plan space and the performance of Amulet's Optimizer

2.7.3 Evaluation of Amulet’s Optimizer

To understand the space of testing plans and to evaluate the quality and efficiency of Amulet’s Optimizer, we developed an *Exhaustive Optimizer (EOpt)*. EOpt works by enumerating the (nearly) full space of possible testing plans per volume as follows:

- P_W is set to the maximum among intervals in all objectives.
- The number of uniformly-spaced snapshots in P_W is varied from 1 to $\frac{P_W}{P_I^{min}}$, where we set the minimum snapshot interval P_I^{min} to 5 minutes.
- For each number of snapshots in P_W , EOpt enumerates all possible test scheduling combinations starting from all tests running on a single host (if all tests specify the same host type, otherwise, the minimum distinct host types), and finishing at a separate host per test.

Plan space size: We consider the Angel program with an RPO and an SSO from Case 1 in Section 2.7.2, and add more tests to the SSO. For each distinct number of tests in the SSO, we varied the τ_{rpo} in the RPO. For each unique Angel program generated in this manner, Figure 2.17a shows the total number of plans obtained by running EOpt. Note the \log_{10} scale in Figure 2.17a on the z -axis which shows the total size of the plan space. The plan space increases drastically as the number of tests increase because the size of the space is exponential in the number of tests.

Cost distribution of valid plans: More than 99% of the plans in the space enumerated by EOpt were invalid for most Angel programs, i.e., their test schedules (P_S) violate one of the specified objectives or are unsustainable (see Equation 2.2). Figure 2.17b shows the distribution of valid plans according to their respective total cost for an Angel program with $\tau_{rpo} = 60$ minutes and an SSO with 4 tests. The figure shows that there is more than one optimal (in this case, minimum cost) plan.

In this case—as well as in all cases where we could run EOpt in a reasonable time frame—Amulet’s Optimizer generated one of the optimal plans.

Given the large plan space per volume and the speed at which it grows (Figure 2.17a), we measured the time that each optimizer takes to find a testing plan per volume. We fixed τ_{rpo} to 60 minutes in the RPO, and varied the number of tests in the SSO from 1 to 6. EOpt’s times increased rapidly from 0.5 seconds to 30+ minutes. (With 6 tests, we killed EOpt after 35 minutes.) Amulet’s Optimizer ran in under 0.5 seconds in all these cases. We further increased the number of tests up to 16 (which would be a high-end number for tests on a single volume). Our Optimizer’s running time remained under 0.5 seconds.

Next, we increased the τ_{rpo} interval in the RPO, and specified an SIO with a maximum snapshot interval of 1 minute to force our Optimizer to come up with a plan where a snapshot is tested every minute. Figure 2.17c summarizes the results. Planning for 3000 snapshots on each of which 16 tests should be run on average, gave an Optimizer running time under 1 minute. We can see this result as: If the snapshot interval P_I is 5 minutes, then Amulet’s optimizer needs less than a minute to produce a plan with a window P_W spanning 10 days (3000 snapshots at $P_I=5$ minutes per snapshot). We conclude that Amulet’s Optimizer is efficient for today’s needs.

Finally, Figure 2.17d shows how Amulet’s Optimizer continues to remain efficient even under non-cost optimization objectives. We used the binary-search algorithm from Section 2.5.6 as the linear-search algorithm has a worse average case compared to the binary-search. We defined a TCO and varied the time interval and the number of tests that are part of the TCO. No cost objective was specified. The goal was to maximize the number of tests in the plan. While non-cost optimization is more expensive, the trend in Figure 2.17d is similar to what we observed for the cost optimization objective in Figure 2.17c.

2.8 Conclusions and Future Work

Hardware errors, software bugs, and mistakes by human SAs can corrupt important sources of data. Current approaches to deal with data corruption are ad hoc and labor-intensive. We introduced the Amulet system that gives SAs a declarative language to specify their objectives regarding the detection and repair of data corruption. Amulet automatically finds and orchestrates efficient testing plans that run integrity tests to meet the specified objectives in cost-effective ways. We believe that Amulet provides a general framework for SAs to analyze cost versus risk tradeoffs regarding data protection. Although we prototyped Amulet on a cloud platform, Amulet can be applied to conventional enterprise environments with minor modifications. As future work we plan to explore several directions including adaptive techniques for plan optimization.

Diagnosing with DiaDS

In this chapter, we present our contributions in the Diagnosing category, namely, a tool called DiaDS. The main goal of DiaDS is to diagnose and find the root causes of performance problems in multi-tier systems. This tool was evaluated and tested on a Database System that runs on top of a Storage Area Network (SAN). Existing diagnosis tools in this domain have a level-specific focus. Examples of such tools include database-only diagnosis tools such as Dias et al. (2005) or SAN-only diagnosis tools such as Shen et al. (2005).

DiaDS is a tool that carefully integrates information from several system layers, including database and SAN systems (in our prototype implementation). The integration is not a simple concatenation of the monitoring data, but also includes annotation of the monitoring data as well as attaching the relevant monitoring data to the related system elements. This approach not only increases the accuracy of diagnosis, but also leads to significant improvements in efficiency.

DiaDS uses a novel combination of non-intrusive machine learning techniques (e.g., Kernel Density Estimation) and domain knowledge encoded in a new symptoms database design. The machine learning component provides core techniques for

problem diagnosis from monitoring data, and domain knowledge acts as checks-and-balances to guide the diagnosis in the right direction. This unique system design enables DiaDS to function effectively even in the presence of multiple concurrent problems as well as noisy data prevalent in production environments. We present the details of the implementation as well as the experimental evaluation in this chapter.

3.1 Introduction

“The online transaction processing database myOLTP has a 30% slow down in processing time, compared to performance two weeks back.” This is a typical problem ticket a database administrator would create for the SAN administrator to analyze and fix. Unless there is an obvious failure or degradation in the storage hardware or the connectivity fabric, the response to this problem ticket would be: *“The I/O rate for myOLTP tablespace volumes has increased 40%, with increased sequential reads, but the response time is within normal bounds.”* This to-and-fro may continue for a few weeks, often driving SAN administrators to take drastic steps such as migrating the database volumes to a new isolated storage controller or creating a dedicated SAN silo (the inverse of *consolidation*, explaining in part why large enterprises still continue to have highly under-utilized storage systems). The *myOLTP* problem may be fixed eventually by the database administrator realizing that a change in a table’s properties had made the plan with sequential data scans inefficient; and the I/O path was never an issue.

The above example is a realistic scenario from large enterprises with separate teams of database and SAN administrators, where each team uses tools specific to its own subsystem. With the growing popularity of Software-as-a-Service, this division is even more predominant with application administrators belonging to the customer, while the computing infrastructure is provided and maintained by the ser-

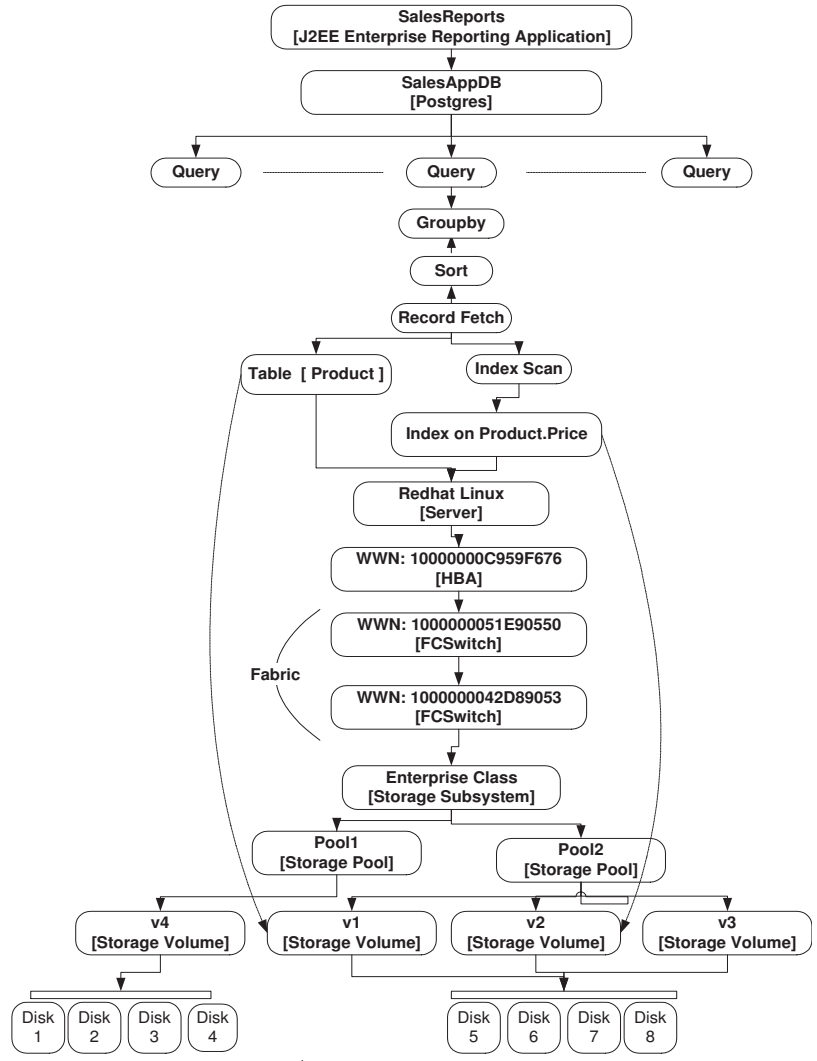


FIGURE 3.1: Example database/SAN deployment. The dotted arrows indicate the effect of noise in real-world data causing mis-classification of the root-cause

vice provider administrators. The result is a lack of end-to-end correlated information across the system stack that makes problem diagnosis hard. Problem resolution in such cases may require either *throwing iron* at the problem and re-creating resource silos, or employing highly-paid consultants who understand both databases and SANs to solve the performance problem tickets.

The goal of this work is to develop an integrated diagnosis tool (called DiaDS) that spans the database and the underlying SAN consisting of end-to-end I/O paths

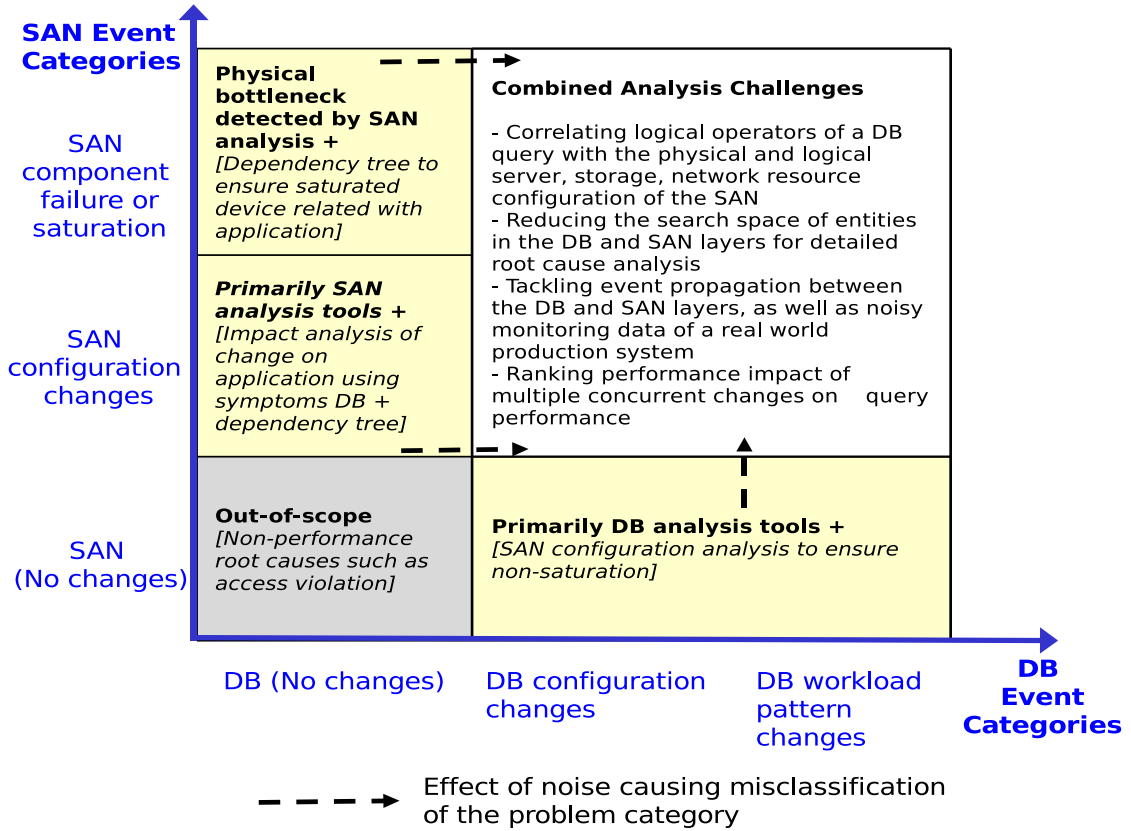


FIGURE 3.2: Taxonomy of scenarios for root-cause analysis

with servers, interconnecting network switches and fabric, and storage controllers. The input to DiaDS is a problem ticket from the SA with respect to a degradation in database query performance. The output is a collection of top-K events from the database and SAN that are candidate root causes for the performance degradation. Internally, DiaDS analyzes thousands of entries in the performance and event logs of the database and individual SAN devices to shortlist an extremely selective subset for further analysis.

3.1.1 Challenges in Integrated Diagnosis

Figure 3.1 shows an integrated database and SAN taxonomy with various logical (e.g., sort and scan operators in a database query plan) and physical components (e.g., server, switch, and storage controller). Diagnosis of problems within the database

or SAN subsystem is an area of ongoing research (described later in Section 3.2). Integrated diagnosis across multiple subsystems is even more challenging:

- **High-dimensional search space:** Integrated analysis involves a large number of entities and their combinations (see Figure 3.1). Pure machine learning techniques that aim to find correlations in the raw monitoring data—which may be effective within a single subsystem with few parameters—can be ineffective in the integrated scenario. Additionally, real-world monitoring data has inaccuracies (i.e., the data is *noisy*). The typical source of noise is the large monitoring interval (5 minutes or higher in production environments) which averages out the instantaneous effects of spikes and other bursty behavior.
- **Event cascading and impact analysis:** The cause and effect of a problem may not be contained within a single subsystem (i.e., *event flooding* may result). Analyzing the impact of an event across multiple subsystems is a nontrivial problem.
- **Deficiencies of rule-based approaches:** Existing diagnosis tools for some commercial databases (e.g., Dias et al., 2005) use a rule-based approach where a root-cause taxonomy is created and then complemented with rules to map observed symptoms to possible root causes. While this approach has the merit of encoding valuable domain knowledge for diagnosis purposes, it may become complex to maintain and customize.

3.1.2 Contributions

The taxonomy of problem determination scenarios handled by DiaDS is shown in Figure 3.2. The events in the SAN subsystem can be broadly classified into configuration changes (such as allocation of new applications, change in interconnectivity, firmware upgrades, etc.) and component failure or saturation events. Similarly,

database events could correspond to changes in the configuration parameters of the database, or a change in the workload characteristics driven by changes in query plans, data properties, etc. The figure represents a matrix of change events, with relatively complex scenarios arising due to combinations of SAN and database events. In real-world systems, the *no change* category is misleading, since there will always be change events recorded in management logs that may not be relevant or may not impact the problem at hand; those events still need to be filtered by the problem determination tool. For completeness, there is another dimension (outside the scope of this work) representing transient effects, e.g., workload contention causing transient saturation of components.

The key contributions of this work are:

- A novel workflow for integrated diagnosis that uses an end-to-end canonical representation of database query operations combined with physical and logical entities from the SAN subsystem (referred to as *dependency paths*). DiaDS generates these paths by analyzing system configuration data, performance metrics, as well as event data generated by the system or by user-defined triggers.
- The workflow is based on an innovative combination of machine learning, domain knowledge of configuration and events, and impact analysis on query performance. This design enables DiaDS to address the integrated diagnosis challenges of high-dimensional space, event propagation, multiple concurrent problems, and noisy data.
- An empirical evaluation of DiaDS on a real-world testbed with a PostgreSQL database running on an enterprise-class storage controller. We describe problem injection scenarios including combinations of events in the database and SAN layers, along with a drill-down into intermediate results given by DiaDS.

3.2 Related Work

We give an overview of relevant database (DB), storage, and systems diagnosis work, some of which is complementary and leveraged by our integrated approach.

3.2.1 Independent DB and Storage Diagnosis

There has been significant prior research in performance diagnosis and problem determination in databases (e.g., Dias et al., 2005; Dageville et al., 2004; Mehta et al., 2008) as well as enterprise storage systems (e.g., Pollack and Uttamchandani, 2006; Shen et al., 2005). Most of these techniques perform diagnosis in an isolated manner attempting to identify root cause(s) of a performance problem in individual database or storage silos. In contrast, DiaDS analyzes and correlates data across the database and storage layers.

DB-only Diagnosis: Oracle’s Automatic Database Diagnostic Monitor (ADDM) (Dageville et al., 2004; Dias et al., 2005) performs fine-grained monitoring to diagnose database performance problems, and to provide tuning recommendations. A similar system (Chaudhuri et al., 2004) has been proposed for Microsoft SQLServer. (Interested readers can refer to Weikum et al. (2002) for a survey on database problem diagnosis and self-tuning.) However, these tools are oblivious to the underlying SAN layer. They cannot detect problems in the SAN, or identify storage-level root causes that propagate to the database subsystem.

Storage-only Diagnosis: Similarly, there has been research in problem determination and diagnosis in enterprise storage systems. Genesis (Pollack and Uttamchandani, 2006) uses machine learning to identify abnormalities in SANs. A disk I/O throughput model and statistical techniques to diagnose performance problems in the storage layer are described in Shen et al. (2005). There has also been work on profiling techniques for local file systems (e.g., Aranya et al., 2004; Zhou et al.,

1985) that help collect data useful in identifying performance bottlenecks as well as in developing models of storage behavior (Joukov et al., 2006; Thereska et al., 2006; Mesnier et al., 2007).

Drawbacks: Independent database and storage analysis can help diagnose problems like deadlocks or disk failures. However, independent analysis may fail to diagnose problems that do not violate conditions in any one layer, rather contribute cumulatively to the overall poor performance. Two additional drawbacks exist. First, it can involve multiple sets of experts and be time consuming. Second, it may lead to spurious corrective actions as problems in one layer will often surface in another layer. For example, slow I/O due to an incorrect storage volume placement may lead a DB administrator to change the query plan. Conversely, a poor query plan that causes a large number of I/Os may lead the storage administrator to provision more storage bandwidth.

Studies measuring the impact of storage systems on database behavior (Reiss and Kanungo, 2003; Qin et al., 2007) indicate a strong interdependence between the two subsystems, highlighting the importance of an integrated diagnosis tool like DiaDS.

3.2.2 System Diagnosis Techniques

Diagnosing performance problems has been a popular research topic in the general systems community in recent years (e.g., Wang et al., 2004; Cohen et al., 2004, 2005; Zhang et al., 2005; Basu et al., 2007; Manji). Broadly, this work can be split into two categories: (a) systems using machine learning techniques, and (b) systems using domain knowledge. As described later, DiaDS uses a novel mix where machine learning provides the core diagnosis techniques while domain knowledge serves as checks-and-balances against spurious correlations.

Diagnosis based on Machine Learning: PeerPressure (Wang et al., 2004) uses statistical techniques to develop models for a healthy machine, and uses these models

to identify *sick* machines. Another proposed method (Basu et al., 2007) builds models from process performance counters in order to identify anomalous processes that cause computer slowdowns. There is also work on diagnosing problems in multi-tier Web applications using machine learning techniques. For example, modified Bayesian network models (Cohen et al., 2004) and ensembles of probabilistic models (Zhang et al., 2005) that capture system behavior under changing conditions have been used. These approaches treat data collected from each subsystem equally, in effect creating a single table of performance metrics that is input to machine learning modules. In contrast, DiaDS adds more structure and semantics to the collected data, e.g., to better understand the impact of database operator performance vs. SAN volume performance. Furthermore, DiaDS complements machine learning techniques with domain knowledge.

Diagnosis based on Domain Knowledge: There are also many systems, especially in the DB community, where domain knowledge is used to create a *symptoms* database that associates performance symptoms with underlying root causes (Yemini et al., 1996; Manji; Perazolo, 2005; Dageville et al., 2004; Dias et al., 2005). Commercial vendors like EMC, IBM, and Oracle use symptom databases for problem diagnosis and correction. While these databases are created manually and require expertise and resources to maintain, recent work attempts to partially automate this process (Cohen et al., 2005; Duan et al., 2009a).

We believe that a suitable mix of machine learning techniques and domain knowledge is required for a diagnosis tool to be useful in practice. Pure machine learning techniques can be misled by spurious correlations in data resulting from noisy data collection or event propagation (where a problem in one component impacts another component). Such effects need to be addressed using appropriate domain knowledge, e.g., component dependencies, symptoms databases, and knowledge of query plan and operator relationships.

It is also important to differentiate DiaDS from tracing-based techniques (e.g., Chen et al., 2004; Aguilera et al., 2003) that trace messages through systems end-to-end to identify performance problems and failures. Such tracing techniques require changes in production system deployments and often add significant overhead in day-to-day operations. In contrast, DiaDS performs a postmortem analysis of monitored performance data collected at industry-standard intervals to identify performance problems.

3.3 Overview of DiaDS

Suppose a query Q that a report-generation application issues periodically to the database system shows a slowdown in performance. One approach to track down the cause is to leverage historic monitoring data collected from the entire system. There are several product offerings (e.g., EMC Control Center; Hewlett Packard Systems Insight Manager; IBM Tivoli Network Manager; IBM TotalStorage Productivity Center; VMWare Virtual Center) in the market that collect and persist monitoring data from IT systems.

DiaDS uses a commercial storage management server – IBM TotalStorage Productivity Center (IBM TotalStorage Productivity Center) – that collects monitoring data from multiple layers of the IT stack including databases, servers, and the SAN. The collected data is transformed into a tabular format, and persisted as time-series data in a relational database.

SAN-level data: The collected data includes: (i) configuration of components (both physical and logical), (ii) connectivity among components, (iii) changes in configuration and connectivity information over time, (iv) performance metrics of components, (v) system-generated events (e.g., disk failure, RAID rebuild) and (vi) events generated by user-defined *triggers* (Garcia-Molina et al., 2001) (e.g., degradation in volume performance, high workload on storage subsystem).

Database-level data: To execute a query, a database system generates a *plan* that consists of operators selected from a small, well-defined family of *operators* (Garcia-Molina et al., 2001). Let us consider an example query Q :

```
SELECT Product.Category, SUM(Product.Sales)
FROM Product
WHERE Product.Price > 1000
GROUP BY Product.Category
```

Q asks for the total sales of products, priced above 1000, grouped per category. Figure 3.1 shows a plan P to execute Q . P consists of four operators: an *Index Scan* of the index on the Price attribute, a *Fetch* to bring matching records from the Product table, a *Sort* to sort these records on Category values, and a *Grouping* to do the grouping and summation. For each execution of P , DiaDS collects some monitoring data per operator O . The relevant data includes: O 's start time, stop time, and *record-count* (number of records returned in O 's output).

DiaDS's Diagnosis Interface: DiaDS presents an interface where a SA can mark a query as having experienced a slowdown. Furthermore, the SA either specifies declaratively or marks directly the runs of the query that were *satisfactory* and those that were *unsatisfactory*. For example, runs with running time below 100 seconds are satisfactory, or all runs between 8 AM and 2 PM were satisfactory, and those between 2 PM and 3 PM were unsatisfactory.

Diagnosis Workflow: DiaDS then invokes the *workflow* shown in Figure 3.3 to diagnose the query slowdown based on the monitoring data collected for satisfactory and unsatisfactory runs. By default, the workflow is run in a batch mode. However, the SA can choose to run the workflow in an interactive mode where only one module is run at a time. After seeing the results of each module, the SA can edit the data or results before feeding them to the next module, bypass or reinvoke modules, or stop the workflow.

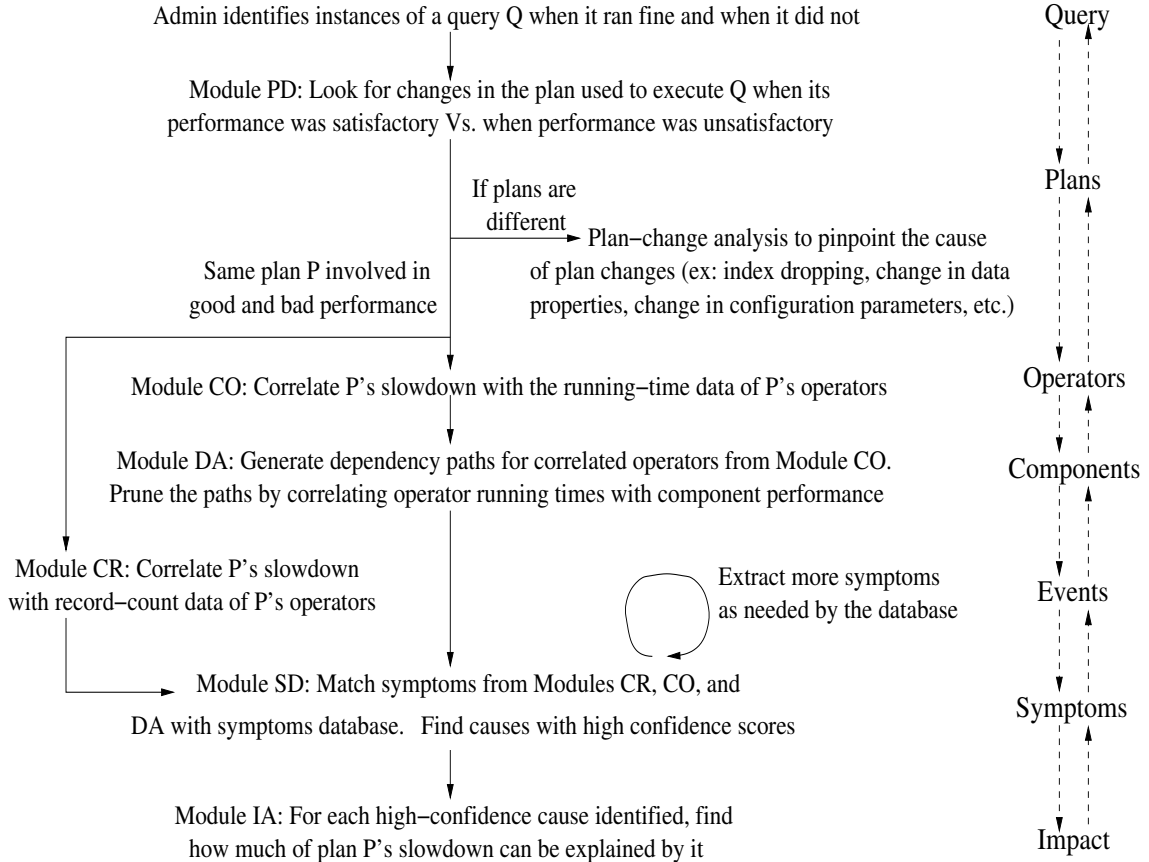


FIGURE 3.3: DiaDS's diagnosis workflow

The first module in the workflow, called Module Plan-Diffing (PD), looks for significant changes between the plans used in satisfactory and unsatisfactory runs. If such changes exist, then DiaDS tries to pinpoint the cause of the plan changes (which includes, e.g., index addition or dropping, changes in data properties, or changes in configuration parameters used during plan selection).

The remaining modules are invoked if DiaDS finds a plan P that is involved in both satisfactory and unsatisfactory runs of the query. We give a brief overview before diving into the details in Section 3.4:

- **Module Correlated Operators (CO):** DiaDS finds the (nonempty) subset of operators in P whose change in performance correlates with the query slowdown. The operators in this subset are called *correlated operators*.

- **Module Dependency Analysis (DA):** Having identified the correlated operators, DiaDS uses a combination of correlation analysis and the configuration and connectivity information collected during monitoring to identify the components in the system whose performance is correlated with the performance of the correlated operators.
- **Module Correlated Record-counts (CR):** Next, DiaDS checks whether the change in P 's performance is correlated with the record-counts of P 's operators. If significant correlations exist, then it means that data properties have changed between satisfactory and unsatisfactory runs of P .
- **Module Symptoms Database (SD):** The correlations identified so far are likely *symptoms* of the root cause(s) of query slowdown. Other symptoms may be present in the stream of system-generated events and trigger-generated (user-defined) semantic events. The combination of these symptoms is used to probe a *symptoms database* that maps symptoms to the underlying root cause(s). The symptoms database improves diagnosis accuracy by dealing with the propagation of faults across components as well as missing symptoms, unexpected symptoms (e.g., spurious correlations), and multiple simultaneous problems.
- **Module Impact Analysis (IA):** The symptoms database computes a *confidence score* for each suspected root cause. For each high-confidence root cause R , DiaDS performs impact analysis to answer the following question: if R is really a cause of the query slowdown, then what fraction of the query slowdown can be attributed to R . To the best of our knowledge, DiaDS is the first automated diagnosis tool to have an impact-analysis module.

Integrated database/SAN diagnosis: Note that the workflow “drills down” progressively from the level of the query to plans and to operators, and then uses dependency analysis and the symptoms database to further drill down to the level of performance metrics and events in components. Finally, impact analysis is a “roll up” to tie potential root causes back to their impact on the query slowdown. The drill down and roll up are based on a careful integration of information from the database and SAN layers; and is not a simple concatenation of database-only and SAN-only modules. Only low overhead monitoring data is used in the entire process.

Machine learning + domain knowledge: DiaDS’s workflow is a novel combination of elements from machine learning with the use of domain knowledge. A number of modules in the workflow use correlation analysis which is implemented using machine learning; the details are in Sections 3.4.1 and 3.4.2. Domain knowledge is incorporated into the workflow in Modules DA, SD, and IA; the details are given respectively in Section 3.4.2–3.4.4. As we will demonstrate, the combination of machine learning and domain knowledge provides built-in checks and balances to deal with the challenges listed in Section 3.1.

3.4 Modules in the Workflow

We now provide details for all modules in DiaDS’s diagnosis workflow. Upfront, we would like to point out that our main goal is to describe an end-to-end instantiation of the workflow. We expect that the specific implementation techniques used for the modules will change with time as we gain more experience with DiaDS.

3.4.1 *Identifying Correlated Operators*

Objective: Given a plan P that is involved in both satisfactory and unsatisfactory runs of the query, DiaDS’s objective in this module is to find the set of correlated operators. Let O_1, O_2, \dots, O_n be the set of all operators in P . The correlated

operators form the subset of O_1, \dots, O_n whose change in running time best explains the change in P 's running time (i.e., P 's slowdown).

Technique: DiaDS identifies the correlated operators by analyzing the monitoring data collected during satisfactory and unsatisfactory runs of P . This data can be seen as records with attributes $A, t(P), t(O_1), t(O_2), \dots, t(O_n)$ for each run of P . Here, attribute $t(P)$ is the total time for one complete run of P , and attribute $t(O_i)$ is the running time of operator O_i for that run. Attribute A is an *annotation* (or *label*) associated with each record that represents whether the corresponding run of P was satisfactory or not. Thus, A takes one of two values: satisfactory (denoted S) or unsatisfactory (denoted U).

Let the values of attribute $t(O_i)$ in records with annotation S be s_1, s_2, \dots, s_k , and those with annotation U be u_1, u_2, \dots, u_l . That is, s_1, \dots, s_k are k observations of the running time of operator O_i when the plan P ran satisfactorily. Similarly, u_1, u_2, \dots, u_l are l observations of the running time of O_i when the running time of P was unsatisfactory. DiaDS pinpoints correlated operators by characterizing how the distribution of s_1, \dots, s_k differs from that of u_1, \dots, u_l . For this purpose, DiaDS uses *Kernel Density Estimation (KDE)* (Parzen, 1962).

KDE is a non-parametric technique to estimate the probability density function of a random variable. Let S_i be the random variable that represents the running time of operator O_i when the overall plan performance is satisfactory. KDE applies a kernel density estimator to the k observations s_1, \dots, s_k of S_i to learn S_i 's probability density function $f_i(S_i)$.

$$f_i(S_i) = \frac{\sum_{j=1}^k K\left(\frac{S_i - s_j}{h}\right)}{kh} \quad (3.1)$$

Here, K is a *kernel function* and h is a smoothing parameter. A typical kernel is

the standard Gaussian function $K(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$. (Intuitively, kernel density estimators are a generalization and improvement over *histograms*.)

Let u be an observation of operator O_i 's running time when the plan performance was unsatisfactory. Consider the probability estimate $prob(S_i \leq u) = \int_{-\infty}^u f_i(S_i) ds_i$. Intuitively, as u becomes higher than the typical range of values of S_i , $prob(S_i \leq u)$ becomes closer to 1. Thus, a high value of $prob(S_i \leq u)$ represents a significant increase in the running time of operator O_i when plan performance was unsatisfactory compared to that when plan performance was satisfactory.

Specifically, DiaDS includes O_i in the set of correlated operators if $prob(S_i \leq \bar{u}) \geq 1 - \alpha$. Here, \bar{u} is the average of u_1, \dots, u_l and α is a small positive constant. $\alpha = 0.1$ by default. For obvious reasons, $prob(S_i \leq \bar{u})$ is called the *anomaly score* of operator O_i .

3.4.2 Dependency Analysis

Objective: This module takes the set of correlated operators as input, and finds the set of system components that show a change in performance correlating with the change in running time of one of more correlated operators.

Technique: DiaDS implements this module using *dependency analysis* which is based on generating and pruning *dependency paths* for the correlated operators. We describe the generation and pruning of dependency paths in turn.

Generating dependency paths: The dependency path of an operator O_i is the set of physical (e.g., server CPU, database buffer cache, disk) and logical (e.g., volume, external workload) components in the system whose performance can have an impact on O_i 's performance. DiaDS generates dependency paths automatically based on the following data:

1. System-wide configuration and connectivity data as well as updates to this data collected during the execution of each operator (recall Section 3.3).
2. Domain knowledge of how each database operator executes. For example, the dependency path of a sort operator that creates temporary tables on disk will be different from one that does not create temporaries.

We distinguish between *inner* and *outer* dependency paths. The performance of components in O_i 's inner dependency path can affect O_i 's performance directly. O_i 's outer dependency path consists of components that affect O_i 's performance indirectly by affecting the performance of components on the inner dependency path. As an example, the inner dependency path for the Index Scan operator in Figure 3.1 includes the server, HBA, FCSwitches, Pool2, Volume $v2$, and Disks 5-8. The outer dependency path will include Volumes $v1$ and $v3$ (because of the shared disks) and other database queries.

Pruning dependency paths: The fact that a component C is in the dependency path of an operator O_i does not necessarily mean that O_i 's performance has been affected by C 's performance. After generating the dependency paths conservatively, DiaDS prunes these paths based on correlation analysis using KDE.

Recall from Section 3.3 that the monitoring data collected by DiaDS contains multiple observations of the running time of operator O_i both when the overall plan ran satisfactorily and when the plan ran unsatisfactorily. For each run of O_i , consider the performance data collected by DiaDS for each component C in O_i 's dependency path; this data is collected in the $[t_b, t_e]$ time interval where t_b and t_e are respectively O_i 's (absolute) start and stop times for that run. Across all runs, this data can be represented as a table with attributes $A, t(O_i), m_1, \dots, m_p$. Here, m_1 - m_p are performance metrics of component C , and the annotation attribute A represents whether O_i 's running time $t(O_i)$ was satisfactory or not in the corresponding run.

It follows from Section 3.4.1 that we can set A 's value in a record to U (denoting unsatisfactory) if $\text{prob}(S_i \leq t(O_i)) \geq 1 - \alpha$; and to S otherwise.

Given the above annotated performance data for an $\langle O_i, C \rangle$ operator-component pairing, we can apply correlation analysis using KDE to identify C 's performance metrics that are correlated with the change in O_i 's performance. The details are similar to that in Section 3.4.1 except for the following: for some performance metrics, observed values lower than the typical range are anomalous. This correlation can be captured using the condition $\text{prob}(M \leq v) \leq \alpha$, where M is the random variable corresponding to the metric, v is a value observed for M , and α is a small positive constant.

In effect, the dependency analysis module will identify the set of components that: (i) are part of O_i 's dependency path, and (ii) have at least one performance metric that is correlated with the running time of a correlated operator O_i . By default, DiaDS will only consider the components in the inner dependency paths of correlated operators. However, components in the outer dependency paths will be considered if required by the symptoms database (Module SD).

Recall Module CR in the diagnosis workflow where DiaDS checks for significant correlation between plan P 's running time and the record counts of P 's operators. DiaDS implements this module using KDE in a manner almost similar to the use of KDE in dependency analysis; hence Module CR is not discussed further.

3.4.3 Symptoms Database

The modules so far in the workflow drilled down from the level of the query to that of physical and logical components in the system; in the process identifying correlated operators and performance metrics. While this information is useful, the detected correlations may only be *symptoms* of the true root cause(s) of the query slowdown. This issue, which can mask the true root cause(s), is generally referred to as the *event*

(*fault*) *propagation* problem in diagnosis. For example, a change in data properties at the database level may, in turn, propagate to the volume level causing volume contention, and to the server level increasing CPU utilization. In addition, some spurious correlations may creep in and manifest themselves as unexpected symptoms in spite of our careful drill down process.

Objective: DiaDS’s Module SD tries to map the observed symptoms to the actual root cause(s), while dealing with missing as well as unexpected symptoms arising from the noisy nature of production systems.

Technique: DiaDS uses a *symptoms database* to do the mapping. This database streamlines the use of domain knowledge in the diagnosis workflow to:

- Generate more accurate diagnosis results by dealing with event propagation.
- Generate diagnosis results that are semantically more meaningful to SAs (for example, reporting lock contention as the root cause instead of reporting some correlated metrics only).

We considered a number of formats proposed previously in the literature to input domain knowledge for aiding diagnosis. Our evaluation criteria were the following:

- I. How easy is the format for SAs to use? Here, usage includes customization, maintenance over time, as well as debugging. When a diagnosis tool pinpoints a particular cause, it is important that the SAs are able to understand and validate the tool’s reasoning. Otherwise, SAs may never trust the tool enough to use it.
- II. Can the format deal with the noisy conditions in production systems, including multiple simultaneous problems, presence of spurious correlations, and missing symptoms.

	symp ₁	symp ₂	symp ₃	symp ₄
R ₁	1	0	0	1
R ₂	1	1	0	0
R ₃	1	0	1	1

FIGURE 3.4: Example codebook

One of the formats from the literature (IBM Tivoli Network Manager) is an *expert knowledge-base* of rules where each rule expresses patterns or relationships that describe symptoms, and can be matched against the monitoring data. Most of the focus in this work has been on exact matches, so this format scores poorly on Criterion II. Representing relationships among symptoms (e.g., event X will cause event Y) using deterministic or probabilistic networks like Bayesian networks (Pearl, 2000) has been gaining currency recently. This format has high expressive power, but remains a black-box for SAs who find it hard to interpret the reasoning process (Criterion I).

Another format, called the *Codebook* (Yemini et al., 1996), is very intuitive as well as implemented in a commercial product. This format assumes a finite set of symptoms such that each distinct root cause R has a unique *signature* in this set. That is, there is a unique subset of symptoms that R gives rise to which differs makes it distinguishable from all other root causes. This information is represented in the Codebook which is a matrix whose columns correspond to the symptoms and rows correspond to the root causes. A cell is mapped to 1 if the corresponding root cause should show the corresponding symptom; and to 0 otherwise. Figure 3.4 shows an example Codebook where there are four hypothetical symptoms $symp_1$ – $symp_4$ and three root causes R_1 – R_3 .

When presented with a vector V of symptoms seen in the system, the Codebook computes the *distance* $d(V, R)$ of V to each row R (i.e., root cause). Any number of different distance metrics can be used, e.g., Euclidean (L_2) distance or Hamming

distance (Yemini et al., 1996). $d(V, R)$ is a measure of the confidence that R is a root cause of the problem. For example, given a symptoms vector $\langle 1, 0, 0, 1 \rangle$ (i.e., only $symp_1$ and $symp_4$ are seen), the Euclidean distances to the three root causes in Figure 3.4 are 0, $\sqrt{2}$, and 1 respectively. Hence, R_1 is the best match.

The Codebook format does well on both our evaluation criteria. Codebooks can handle noisy situations, and SAs can easily validate the reasoning process. However, DiaDS needs to consider complex symptoms such as symptoms with temporal properties. For example, we may need to specify a symptom where a disk failure is seen within X minutes of the first incidence of the query slowdown, where X may vary depending on the installation. Thus, it is almost impossible in our domain to fully enumerate a closed space of relevant symptoms, and to specify for each root cause whether each symptom from this space will be seen or not. These observations led to DiaDS’s new design of the symptoms database:

1. We define a *base set* of symptoms consisting of: (i) operators in the database system that can be included in the correlated set, (ii) performance metrics of components that can be correlated with operator performance, and (iii) system-monitored and user-defined events collected by DiaDS.
2. The language defined by IBM’s *Active Correlation Technology (ACT)* is used to express complex symptoms over the base set of symptoms (Biazetti and Gajda, 2005). The benefit of this language comes from its support for a range of built-in patterns including filter, collection, duplicate, computation, threshold, sequence, and timer. ACT can express symptoms like: (i) the workload on a volume is higher than 200 IOPS, and (ii) event E_1 should follow event E_2 in the 30 minutes preceding the first instance of query slowdown.
3. DiaDS’s symptoms database is a collection of root cause entries each of which has the format $Cond_1 \ \& \ Cond_2 \ \& \ \dots \ \& \ Cond_z$, for some $z > 0$ which can differ

across entries. Each $Cond_i$ is a Boolean condition of the form $\exists symp_j$ (denoting presence of $symp_j$) or $\neg\exists symp_j$ (denoting absence of $symp_j$). Here, $symp_j$ is some base or complex symptom. Each $Cond_i$ is associated with a weight w_i such the sum of the weights for each individual root cause entry is 100%. That is, $\sum_{i=1}^z w_i = 100\%$.

4. Given a vector of base symptoms, DiaDS computes a *confidence score* for each root cause entry R as the sum of the weights of R 's conditions that evaluate to true. Thus, the confidence score for R is a value in $[0\%, 100\%]$ equal to $\sum_{i=1}^z w_i | Cond_i = true$.

DiaDS's symptoms database tries to balance the expressive power of rules with the intuitive structure and robustness of Codebooks. The symptoms database differs from conventional Codebooks in a number of ways. For each root cause entry, DiaDS avoids the "closed-world" assumption for symptoms by mapping symptoms to 0, 1, or "don't care". Conventional Codebooks are constrained to 0 or 1 mappings. DiaDS's symptoms database can contain mappings for *fixes* to problems in addition to root causes. This feature is useful because it may be easier to specify a fix for a query slowdown (e.g., add an index) instead of trying to find the root cause. DiaDS also allows multiple distinct entries for the same root cause.

Generation of the symptoms database: Companies like EMC, IBM, HP, and Oracle are investing significant (currently, mostly manual) effort to create symptoms databases for different subsystems like networking infrastructure, application servers, and databases (Yemini et al., 1996; Manji; Perazolo, 2005; Cohen et al., 2005; Dageville et al., 2004; Dias et al., 2005). Symptoms databases created by some of these efforts are already in commercial use. The creation of these databases can be partially automated, e.g., through a combination of fault injection and machine learning (Cohen et al., 2005; Duan et al., 2009a). In fact, DiaDS's modules like corre-

lation, dependency, and impact analysis can be used to identify important symptoms automatically.

3.4.4 Impact Analysis

Objective: The confidence score computed by the symptoms database module for a potential root cause R captures how well the symptoms seen in the system match the expected symptoms of R . For each root cause R whose confidence score exceeds a threshold, the impact analysis module computes R 's *impact score*. If R is an actual root cause, then R 's impact score represents the fraction of the query slowdown that can be attributed to R individually. DiaDS's novel impact analysis module serves three significant purposes:

- When multiple problems coexist in the system, impact analysis can separate out high-impact causes from the less significant ones; enabling prioritization of SA effort in problem solving.
- As a safeguard against misdiagnoses caused by spurious correlations due to noise.
- As an extra check to find whether we have identified the right cause(s) or all cause(s).

Technique: Interestingly, one approach for impact analysis is to *invert* the process of dependency analysis from Section 3.4.2. Let R be a potential root cause whose impact score needs to be estimated:

1. Identify the set of components, denoted $comp(R)$, that R affects in the inner dependency path of the operators in the query plan. DiaDS gets this information from the symptoms database.

2. For each component $C \in comp(R)$, find the subset of correlated operators, denoted $op(R)$, such that for each operator O in this subset: (i) C is in O 's inner dependency path, and (ii) at least one performance metric of C is correlated with the change in O 's performance. DiaDS has already computed this information in the dependency analysis module.
3. R 's impact score is the percentage of the change in plan running time (query slowdown) that can be attributed to the change in running time of operators in $op(R)$. Here, change in running time is computed as the difference between the average running times when performance is unsatisfactory and that when performance is satisfactory.

The above approach will work as long as for any pair of suspected root causes R_1 and R_2 , $op(R_1) \cap op(R_2) = \emptyset$. However, if there are one or more operators common to $op(R_1)$ and $op(R_2)$ whose running times have changed significantly, then the above approach cannot fully separate out the individual impacts of R_1 and R_2 .

DiaDS addresses the above problem by leveraging *plan cost models* that play a critical role in all database systems. For each query submitted to a database system, the system will consider a number of different plans, use the plan cost model to predict the running time (or some other cost metric) of each plan, and then select the plan with minimum predicted running time to run the query to completion. These cost models have two main components:

- Analytical formula per operator type (e.g., sort, index scan) that estimates the resource usage (e.g., CPU and I/O) of the operator based on the values of input parameters. While the number and types of input parameters depend on the operator type, the main ones are the sizes of the input processed by the operator.

- Mapping parameters that convert resource-usage estimates into running-time estimates. For example, IBM DB2 uses two such parameters to convert the number of estimated I/Os into a running-time estimate: (i) the overhead per I/O operation, and (ii) the transfer rate of the underlying storage device.

The following are two examples of how DiaDS uses plan cost models:

- Since DiaDS collects the old and new record-counts for each operator, it estimates the impact score of a change in data properties by plugging the new record-counts into the plan cost model.
- When volume contention is caused by an external workload, DiaDS estimates the new I/O latency of the volume from actual observations or the use of device performance models. The impact score of the volume contention is computed by plugging this new estimate into the plan cost model.

DiaDS’s use of plan cost models is a general technique for impact analysis, but it is limited by what effects are accounted for in the model. For example, if wait times for locks are not modeled, then the impact score cannot be computed for locking-based problems. Addressing this issue—e.g., by extending plan cost models or by using *planned experiments* at run time—is an interesting avenue for future work.

3.5 Experimental Evaluation

The taxonomy of scenarios considered for diagnosis in the evaluation follows from Figure 3.2. DiaDS was used to diagnose query slowdowns caused by (i) events within the database and the SAN layers, (ii) combinations of events across both layers, as well as (iii) multiple concurrent problems (a capability unique to DiaDS). Due to space limitations, it is not possible to describe all the scenario permutations from Figure 3.2. Instead, we start with a scenario and make it increasingly complex by

combining events across the database and SAN. We consider: (i) volume contention caused by SAN misconfiguration, (ii) database-level problems (change in data properties, contention due to table locking) whose symptoms propagate to the SAN, and (iii) independent and concurrent database-level and SAN-level problems.

We provide insights into how DiaDS diagnoses these problems by drilling down to the intermediate results like anomaly, confidence, and impact scores. While there is no equivalent tool available for comparison with DiaDS, we provide insights on the results that a database-only or SAN-only tool would have generated; these insights are derived from hands-on experience with multiple in-house and commercial tools used by SAs today. Within the context of the scenarios, we also report sensitivity analysis of the anomaly score to the number of historic samples and length of the monitoring interval.

3.5.1 Setup Details

Our experimental testbed is part of a production SAN environment, with the interconnecting fabric and storage controllers being shared by other applications. Our experiments ran during low activity time-periods on the production environment. The testbed runs data-warehousing queries from the popular TPC-H benchmark (tpch, 2009) on a PostgreSQL database server configured to access tables using two Ext3 filesystem volumes created on an enterprise-class IBM DS6000 storage controller. The database server is a 2-way 1.7 GHz IBM xSeries machine running Linux (Redhat 4.0 Server), connected to the storage controller via Fibre Channel (FC) host bus adaptor (HBA). Both the storage volumes are RAID 5 configurations consisting of (4 + 2P) 15K FC disks.

An IBM TotalStorage Productivity Center (IBM TotalStorage Productivity Center) SAN management server runs on a separate machine recording configuration details, statistics, and events from the SAN as well as from PostgreSQL (which was

instrumented to report the data to the management tool). Figure 3.6 shows the key performance metrics collected from the database and SAN. The monitoring data is stored as time-series data in a DB2 database. Each module in DiaDS’s workflow is implemented using a combination of Matlab scripts (for KDE) and Java. DiaDS uses a symptoms database that was developed in-house to diagnose query slowdowns in database over SAN deployments.

Our experimental results focus on the slowdown of the plan shown in Figure 3.5 for Query 2 from TPC-H. Figure 3.5 shows the 25 operators in the plan, denoted O_1 – O_{25} . In database terminology, the operators Index Scan and Sequential Scan are *leaf* operators since they access data directly from the tables; hence the leaf operators are the most sensitive to changes in SAN performance. The plan has 9 leaf operators. The other operators process intermediate results.

3.5.2 Scenario 1: Volume Contention due to SAN Misconfiguration

Problem Setting

In this scenario, a contention is created in volume V1 (from Figure 3.5) causing a slowdown in query performance. The root cause of the contention is another application workload that is configured in the SAN to use a volume V’ that gets mapped to the same physical disks as V1. For an accurate diagnosis result, DiaDS needs to pinpoint the combination of SAN configuration events generated on: (i) creation of the new volume V’, and (ii) creation of a new zoning and mapping relationship of the server running the workload that accesses V’.

Module CO

DiaDS analyzes the historic monitoring samples collected for each of the 25 query operators. The monitoring samples for an operator are labeled as satisfactory or unsatisfactory based on past problem reports from the SA. Using the operator run-

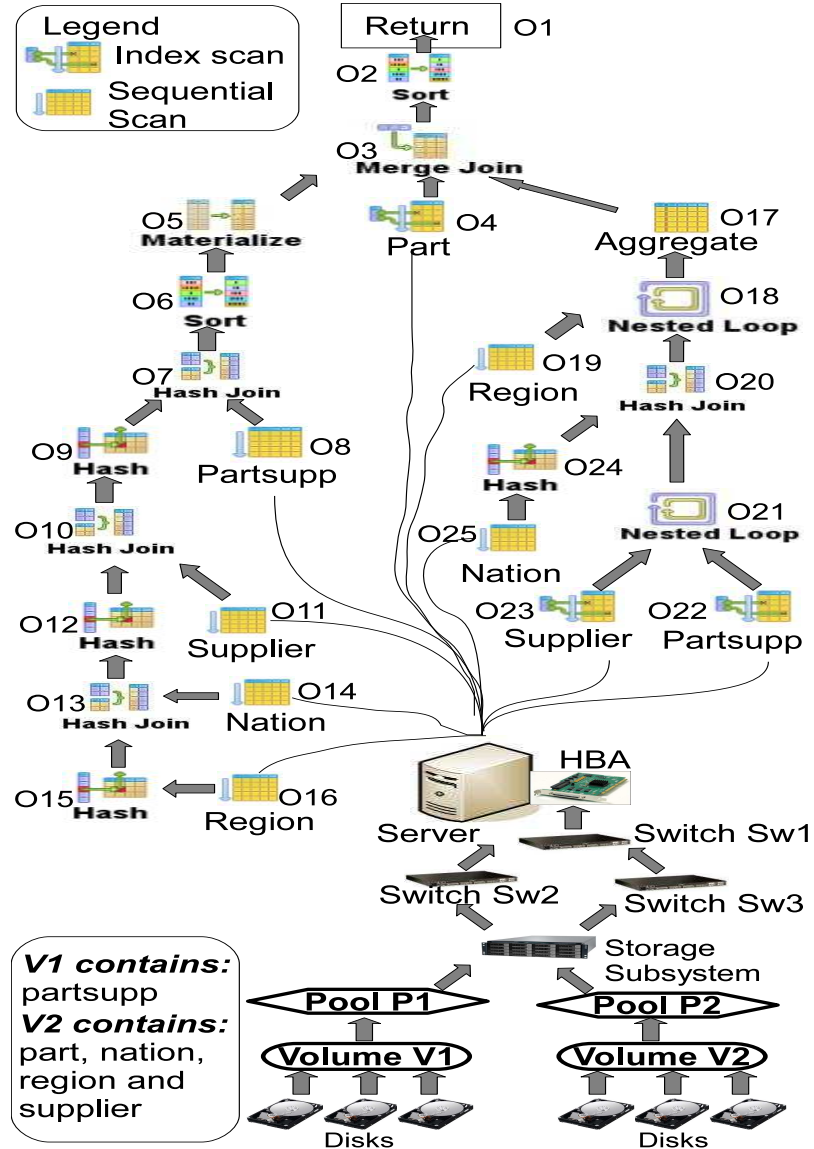


FIGURE 3.5: Query plan, operators, and dependency paths for the experimental results

ning times in these labeled samples, Module CO in the workflow uses KDE to compute anomaly scores for the operators (recall Section 3.4.1). Table 3.1 shows the anomaly scores of the operators identified as the correlated operators; these operators have anomaly scores ≥ 0.8 (the significance of the anomaly scores is covered in Section 3.4.1). The following observations can be made from Table 3.1:

Database Metrics	Server Metrics	Network Metrics	Storage Metrics
Operator Start Stop Times	CPU Usage (%ge)	Bytes Transmitted	Bytes Read
Record-counts	CPU Usage (Mhz)	Bytes Received	Bytes Written
Plan Start Stop Times	Handles	Packets Transmitted	Contaminating Writes
Locks outstanding and held	Threads	Packets Received	PhysicalStorageRead Operations
Lock wait times	Processes	LIP Count	Physical Storage Read Time
Space Usage	Heap Memory Usage(KB)	NOS Count	PhysicalStorageWriteOperations
Blocks Read	Physical Memory Usage (%)	Error Frames	Physical Storage Write Time
Buffer Hits	Kernel Memory(KB)	Dumped Frames	Sequential Read Requests
Index Scans	Memory Being Swapped(KB)	Link Failures	Sequential Write Requests
Index Reads	Reserved Memory	CRC Errors	Total IOs
Index Fetches	Capacity(KB)	Address Errors	
Sequential Scans	Wait I/O		
	Network Bandwidth (HBA)		

FIGURE 3.6: Important performance metrics collected by DiaDS

- Leaf operators O_8 and O_{22} were correctly identified as correlated. These two are the only leaf operators that access data on the Volume V1 under contention.
- Eight intermediate operators were ranked highly as well. This ranking can be explained by event propagation where the running times of these operators are affected by the running times of the “upstream” operators in the plan (in this case O_8 and O_{22}).
- A false positive for leaf operator O_4 which operates on tables in Volume V2. This could be a result of noisy monitoring data associated with the operator.

In summary, Module CO’s KDE analysis has zero false negatives and one false positive from the total set of 9 leaf operators. The false positive gets filtered out later in the symptoms database and impact analysis modules.

To further understand the anomaly scores, we conducted a series of sensitivity tests. Figure 3.7 shows the sensitivity of the anomaly scores of three representative operators to the number of samples available from the satisfactory runs. O_{22} ’s score converges quickly to 1 because O_{22} ’s running time under volume contention is almost 5X the normal. However, the scores for leaf operator O_{11} and intermediate operator O_1 take around 20 samples to converge. With fewer than these many samples, O_{11} could have become a false positive. In all our results, the anomaly scores of all

Table 3.1: Anomaly scores for query operators from Figure 3.5 in Scenario 1

Operator	Operator Type	Anomaly Score
O_2	Non-leaf	1.00
O_3	Non-leaf	1.00
O_6	Non-leaf	1.00
O_7	Non-leaf	1.00
O_8	Leaf (sequential scan)	1.00
O_{18}	Non-leaf	1.00
O_{20}	Non-leaf	1.00
O_{21}	Non-leaf	1.00
O_{22}	Leaf (index scan)	1.00
O_{17}	Non-leaf	0.969
O_4	Leaf (index scan)	0.965

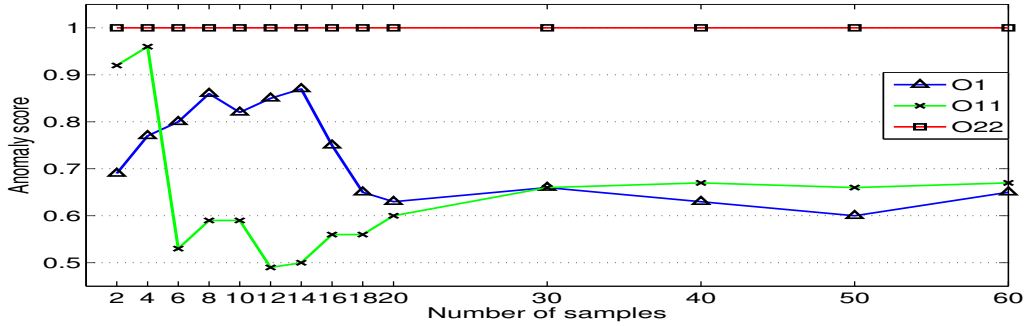


FIGURE 3.7: Sensitivity of anomaly scores to the number of satisfactory samples. While O_{22} shows highly anomalous behavior, scores for O_1 and O_{11} should be low

25 operators converge within 20 samples. While more samples may be required in environments with higher noise levels, the relative simplicity of KDE (compared to models like Bayesian networks) keeps this number low.

Figure 3.8 shows the sensitivity of O_{22} 's anomaly score to the length of the monitoring interval during a 4-hour period. Intuitively, larger monitoring intervals suppress the effect of spikes and bursty access patterns. In our experiments, the query running time was around 4 minutes under satisfactory conditions. Thus, monitoring intervals of 10 minutes and larger in Figure 3.8 cause the anomaly score to deviate more and more from the true value.

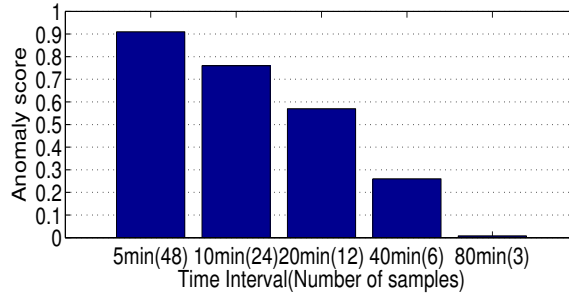


FIGURE 3.8: Sensitivity of anomaly scores to noise in the monitoring data

Table 3.2: Anomaly scores computed during dependency analysis for performance metrics from Volumes V1, V2

Volume, Perf. Metric	Anomaly Score (no contention in V2)	Anomaly Score (contention in V2)
V1, writeIO	0.894	0.894
V1, writeTime	0.823	0.823
V2, writeIO	0.063	0.512
V2, writeTime	0.479	0.879

Module DA

This module generates and prunes dependency paths for correlated operators in order to relate operator performance to database and SAN component performance. For ease of presentation, we will focus on the leaf operators in Figure 3.5 since they are the most sensitive to SAN performance. Given the configuration of our experimental testbed in Figure 3.5, the primary difference between the dependency paths of various operators is in the volumes they access: V1 is in the dependency path of O_8 and O_{22} , and V2 is in the paths of O_4 , O_{11} , O_{14} , O_{16} , O_{19} , O_{23} , and O_{25} .

The set of correlated operators from Module CO are O_4 , O_8 , and O_{22} . Thus, DiaDS will compute anomaly scores for the performance metrics of both V1 and V2. Table 3.2’s second column shows the anomaly scores for two representative metrics each from V1 and V2. (Table 3.2’s third column is described later in this section.) As expected, none of V2’s metrics are identified as correlated because V2 has no contention; while those of V1 are.

Module CR

Anomaly scores are low in this module because data properties do not change.

Module SD

The symptoms identified up to this stage are:

- High anomaly scores for operators dependent on V1.
- High anomaly scores for V1's performance metrics.
- High anomaly score for only one V2-dependent operator (out of seven such operators).

These symptoms are strong evidence that V1's performance is a cause of the query slowdown, and V2's performance is not. Thus, even when a symptoms database is not available, DiaDS correctly narrows down the search space a SA has to consider during diagnosis. An impact analysis will further point out that the false positive symptom due to O_4 has little impact on the query slowdown.

However, without a symptoms database or further diagnosis effort from the SA, the *root cause* of V1's change of performance is still unknown among possible candidates like: (i) change of performance of an external workload, (ii) a runaway query in the database, or (iii) a RAID rebuild. We will now report results from the use of a symptoms database that was developed in-house. DiaDS uses this database as described in Section 3.4.3 except that instead of reporting numeric confidence scores to SAs, DiaDS reports confidence as one of High ($score \geq 80\%$), Medium ($80\% > score \geq 50\%$), or Low ($50\% > score \geq 0\%$). The summary of Module SD's output in the current scenario is:

- All root causes with contention-related symptoms for V2 have Low confidence (few symptoms are found).

- RAID rebuild gets Low confidence because no RAID rebuild start or end events are found.
- V1 contention due to changes in data properties gets Low confidence because symptoms are missing.
- V1 contention due to change in external workload gets Low confidence because no external workload was on the outer dependency path of a correlated operator when performance was satisfactory.
- V1 contention due to change in database workload gets Medium confidence because of a weak correlation between the performance of some correlated operators and the rest of the database workload.
- V1 contention due to the SAN misconfiguration problem gets High confidence because all specified symptoms are found including: (i) creation of a new volume (parametrized with the physical disk information), and (ii) creation of new masking and zoning information for the volume.

The symptoms database had an entry for the actual root cause because this problem is common. Hence, DiaDS was able to diagnose the root cause for this scenario. Note that DiaDS had to consider more than 900 events (system generated as well as user-defined) for the database and SAN generated during the course of the satisfactory and unsatisfactory runs for this experiment.

Module IA

Impact analysis done using the inverse dependency analysis technique gives an impact score of 99.8% for the high-confidence root cause found. This score is high because the slowdown is caused entirely by the contention in V1.

In keeping with our experimental methodology, we complicated the problem scenario to test DiaDS’s robustness. Everything was kept the same except that we created extra I/O load on Volume V2 in a bursty manner such that this extra load had little impact on the query beyond the original impact of V1’s contention. Without intrusive tracing, it would not be possible to rule out the extra load on V2 as a potential cause of the slowdown.

Interestingly, DiaDS’s integrated approach is still able to give the right answer. Compared to the previous scenario, there will now be some extra symptoms due to higher anomaly scores for V2’s performance metrics (as shown in the third column in Table 3.2). However, root causes with contention-related symptoms for V2 will still have Low confidence because most of the leaf operators depending on V2 will have low anomaly scores as before. Also, impact scores will be low for these causes.

Unlike DiaDS, a SAN-only diagnosis tool may spot higher I/O loads in both V1 and V2, and attribute both of these as potential root causes. Even worse, the tool may give more importance to V2 because most of the data is on V2. A database-only tool can pinpoint the slowdown in the operators. However, this tool cannot track the root cause down to the SAN level because it has no visibility into SAN configuration or performance. From our experience, database-only tools may give several false positives in this context, e.g., suboptimal bufferpool setting or a suboptimal choice of execution plan.

3.5.3 Scenario 2: Database-Layer Problem Propagating to the SAN-Layer

In this scenario we cause a query slowdown by changing the properties of the data, causing extra I/O on Volume V2. The change is done by an update statement that modifies the value of an attribute in some records of the *part* table. The overall size of all tables, including *part*, are unchanged. There are no external causes of contention on the volumes.

Modules CO, DA, and CR behave as expected. In particular, module CR correctly identifies all the operators whose record-counts show a correlation with plan performance: operators O_1 , O_2 , O_3 , and O_4 show increased record-counts, while operators O_5 and O_6 show reduced record-counts. The root-cause entry for changes in data properties gets High confidence in Module SD because all needed symptoms match. All other root-cause entries get Low confidence, including contention due to changes in external workload and database workload because no correlations are detected on the outer dependency paths of correlated operators (as expected).

The impact analysis module gives the final confirmation that the change in data properties is the root cause, and rules out the presence of high-impact external causes of volume contention. As described in Section 3.4.4, we can use the plan cost model from the database to estimate the individual impact of any change in data properties. In this case, the impact score for the change in data properties is 88.31%. Hence, DiaDS could have diagnosed the root cause of this problem even if the symptoms database was unavailable or incomplete.

3.5.4 Scenario 3: Concurrent Database-Layer and SAN-Layer Problems

We complicate Scenario 2 by injecting contention on Volume V2 due to SAN misconfiguration along with the change in data properties. Both these problems individually cause contention in V2. The SAN misconfiguration is the higher-impact cause in our testbed. This key scenario represents the occurrence of multiple, possibly related, events at the database and SAN layers, complicating the diagnosis process. The expected result from DiaDS is the ability to pinpoint both these events as causes, and giving the relative impact of each cause on query performance.

The CO, DA, and CR Modules behave in a fashion similar to Scenario 2, and drill down to the contention in Volume V2. We considered DiaDS's performance in two cases: with and without the symptoms database. When the symptoms database

is unavailable or incomplete, DiaDS cannot distinguish between Scenarios 2 and 3. However, DiaDS's impact analysis module computes the impact score for the change in data properties, which comes to 0.56%. (This low score is representative because the SAN misconfiguration has more than 10X higher impact on the query performance than the change in data properties.) Hence, DiaDS final answer in this case is as follows: (i) a change in data properties is a high-confidence but low-impact cause of the problem, and (ii) there are one or more other causes that impact V2 which could not be diagnosed.

When the symptoms database is present, both the actual root causes are given High confidence by Module SD because the needed symptoms are seen in both cases. Thus, DiaDS will pinpoint both the causes. Furthermore, impact analysis will confirm that the full impact on the query performance can be explained by these two causes.

A database-only diagnosis tool would have successfully diagnosed the change in data properties in both Scenarios 2 and 3. However, the tool may have difficulty distinguishing between these two scenarios or pinpointing causes at the SAN layer. A SAN-only diagnosis tool will pinpoint the volume overload. However, it will not be able to separate out the impacts of the two causes. Since the sizes of the tables do not change, we also suspect that such a tool may even rule out the possibility of a change in data properties being a cause.

3.5.5 Discussion

The scenarios described in the experimental evaluation were carefully chosen to be simple, but not simplistic. They are representative of event categories occurring within the DB and SAN layers as shown in Figure 3.2. We have additionally experimented with different events within those categories such as CPU and memory contention in the SAN in addition to disk-level saturation, different types of database

misconfiguration, and locking-based database problems. Locking-based problems are hard to diagnose because they can cause different types of symptoms in the SAN layer, including contention as well as underutilization. We have also considered concurrent occurrence of three or more problems, e.g., change in data properties, SAN misconfiguration, and locking-based problems. The insights from these experiments are similar to those seen already, and further confirm the utility of an integrated tool. However:

- High levels of noise in the monitoring data can reduce DiaDS’s effectiveness.
- While DiaDS would still be effective when the symptoms database is incomplete, more manual effort will be needed to pinpoint actual root causes.
- Incomplete or inaccurate plan cost models reduce the accuracy of impact analysis.

3.6 Conclusions and Future Work

This chapter presented an integrated database and storage diagnosis tool called DiaDS. Using a novel combination of machine learning techniques with database and storage expert domain-knowledge, DiaDS accurately identifies the root cause(s) of problems in query performance; irrespective of whether the problem occurs in the database or the storage layer. This integration enables a more accurate and efficient diagnosis tool for SAs. Through a detailed experimental evaluation, we also demonstrated the robustness of our approach: with its ability to deal with multiple concurrent problems as well as the presence of noisy data.

In future, we are interested in exploring two directions of research. First, we are investigating approaches that further strengthen the analysis done as part of DiaDS modules, e.g., techniques that complement database query plan models using planned

run-time experiments. Second, we aim to generalize our diagnosis techniques to support applications other than databases in conjunction with enterprise storage.

4

Testing with Flex

In this chapter we present our contributions in the Testing category, namely, a system called Flex. Flex is motivated by the routine need from the SAs to perform testing of production systems with production-like workloads (W), configurations (C), data (D), and resources (R). 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 degradations observed after the production system 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 systems that a SA has to manage is growing rapidly.

Flex is a platform for trustworthy testing and tuning of production systems. To make the presentation concrete, we present the integration of Flex with production database system instances. Flex gives *DataBase Administrators* (DBAs) a declarative 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. In this chapter, we present the implementation as well as 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.

4.1 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 (Digital Universe, 2011). Meanwhile, skilled information technology (IT) staff to manage the growing number of servers and data will increase less than 1.5 times (Digital Universe, 2011).

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., Elmore et al., 2011; Oracle Online Index Rebuild). However, more challenging administrative tasks to meet database availabil-

ity 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 (Peter Zaitsev, 2007), Oracle (Oracle Upgrade Regression), PostgreSQL (PostgreSQL TPC-H Bug), 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 (CouchDB Data Loss, 2010).

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 (Borisov et al., 2011). 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

media (bit rot), bit flips in CPU or RAM due to alpha particles, bugs in software or firmware, as well as mistakes by human administrators (Subramanian et al., 2010). 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 (Magnolia, 2009).

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.

Table 4.1: 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

A/B testing	
Description of database administration task in English	How would the current database workload have performed if index I_1 had (hypothetically) been part of the production database’s configuration?

Continued on next page

Task specification using w,c,r,d representation	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 $R(t_{cur})$
Things to note	Tries a configuration different from the production database's current one
Tuning surface creation (Figure 4.10)	
Description of database administration task in English	How does the throughput obtained for my website's OLTP workload due to my MySQL database change as its <i>query_cache_size</i> and <i>key_buffer_size</i> parameters are varied in the ranges [0,10GB] and [0,8GB] respectively?
Task specification using w,c,r,d representation	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 <i>query_cache_size</i> is set to q_i and <i>key_buffer_size</i> is set to k_j
Things to note	Gives potential to run multiple independent experiments in parallel
Upgrade planning	
Description of database administration task in English	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 <i>last</i> week as well as each day of the <i>coming</i> week?
Task specification using w,c,r,d representation	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
Things to note	Tries a different software image on the production database's past and future states
Data integrity testing	
Description of database administration task in English	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)
Task specification using w,c,r,d representation	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
Things to note	Runs a custom workload on an indefinite number of future states
Stress testing	
Description of database administration task in English	If the workload on my production database goes up by 10x in the coming holiday season, then how will it perform?
Task specification using w,c,r,d representation	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
Things to note	Uses a scaled version of workload ¹

End of Table 4.1

4.1.1 Flex

The Flex platform enables efficient experimentation for trustworthy testing and tuning of production database instances. Figure 4.1 gives a high-level illustration of the core concepts that Flex is based on. These concepts will be defined precisely later in the chapter.

Flex treats the production database instance as an entity that can evolve over time. As illustrated in Figure 4.1, 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 .

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 4.1, 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 4.1 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.

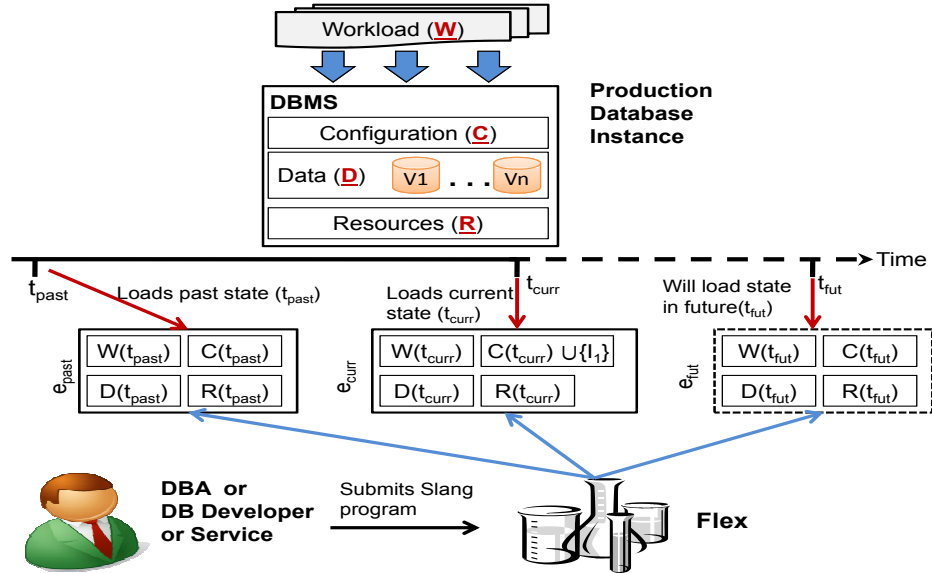


FIGURE 4.1: Overview of how Flex enables trustworthy experimentation with state (workload, configuration, and data) derived from the production database instance

4.1.2 Contributions and Challenges

Slang declarative 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 illustrated in Table 4.1. We have developed the *Slang*² declarative 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. Section 4.2 and Section 4.4 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:

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

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

- 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 Section 4.5 and Section 4.6.

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 4.7.

Section 4.8 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.

4.2 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—with the possibility of 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 4.1.

Host: In this work, 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 4.10.) 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.

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

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 thirteen different host types currently (Amazon Host Types). 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, file-system, 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 give the details of one in Section 4.6—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.

```

Action A1 {
  execute: scripts/myisamchk1.sh
  datum: /volume/lineitem.MY1
  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.MY1
  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 4.2: Example slang program #1 that references past snapshots

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 4.6.

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 4.1, the following are some commonly-used actions in experiments that Flex runs:

- Replay the workload as captured in the snapshot s (Galanis et al., 2008).
- 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 .

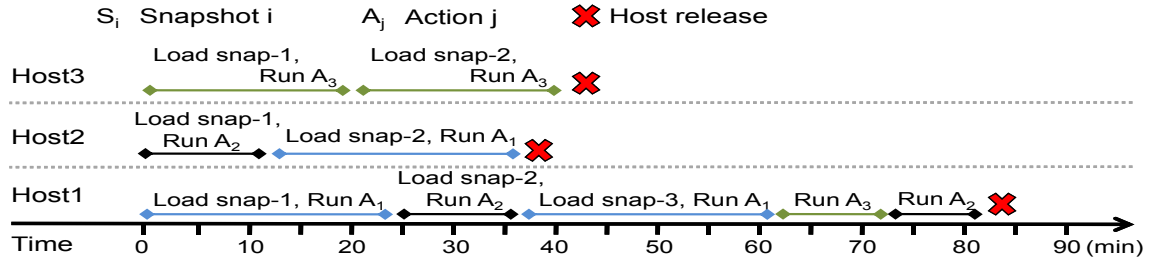


FIGURE 4.3: Execution of example slang program #1 from Figure 4.2 (Section 4.8.2 gives a detailed description)

4.3 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 developer. (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 4.2 shows how the user will express this task in the Slang declarative 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 4.3 shows the complete execution timeline of the example Slang program #1 from Figure 4.2. Notice from Figure 4.3 that the Orchestrator has to make choices regarding when to allocate and release hosts, which experiments to schedule on which hosts

Table 4.2: Summary of statements in the Slang language

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

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 4.3. 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 4.1 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 chapter.

4.4 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, 2011)? 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 4.1, 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 4.2 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 4.2, 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 4.2, 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 4.2 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 4.2, 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 4.7.) 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 4.2.

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 4.1 have this property.) In this scenario, the `Snapshot` statement is provided to specify the arrival frequency and type of future snapshots. Section 4.6 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 4.2 and Table 4.2. 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 definitions in a Slang program are provided by the `Action`, `Host`, `Snapshot`, and `Credentials` statements. 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 4.1 and 4.2 that cloud platforms are the typical resource providers for Flex.

In our example program in Figure 4.2, 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

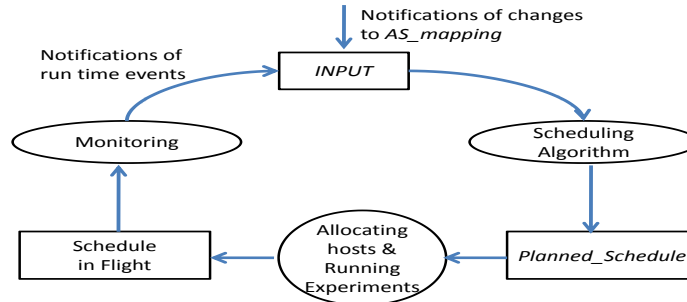


FIGURE 4.4: Steps in Flex's orchestration of experiments

```

Action m {
  execute: scripts/myisamchk.sh
  datum: /volume/*.MYI
  readOnly: true
  reboot: false
  before: tests/scripts/before_myisam.sh
  after: tests/scripts/after_myisam.sh
}

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

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

Credentials loginDetails { ... }

Snapshot exp {
  device: /dev/sdf
  mountDir: /volume
  interval: 10
  waiting: true
  consistency: DATABASE
  snapshotType: Full_Lazy
  production: ec2-user@12.21.amazon.com
}

Plan P {
  mapping: ( [m,exp-1, h1] [f,exp-1, h2] [m,exp-2, h1]
  [f,exp-2, h2] [m,exp-3,h1] [f,exp-3,h2])
  hosts: ( [h1, testHost] [h2, testHost] )
  deadline: 30
  budget: 3
  repeat: true
}
  
```

FIGURE 4.5: Example slang program #2 that references future snapshots

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. The example Slang program in Figure 4.5 has this nature.

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.

4.5 Orchestration of Experiments

Figure 4.4 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. Scheduling and running experiments on hosts based on the *Planned_Schedule*, and 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.

Table 4.3: *INPUT*: Inputs to the orchestration process

Name	Description
<i>AS_Mapping</i>	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
<i>status</i> ($\langle a_i, s_j \rangle$)	One of “ <i>WAITING</i> ” or “ <i>RUNNING</i> on host h_k ” based on current scheduling status of $\langle a_i, s_j \rangle \in AS_Mapping$
h_1, \dots, h_n	Ordered list of currently-allocated hosts (initially empty)
<i>end</i> (h_k)	The later of current time (NOW) and the time when host h_k will finish its last scheduled action
<i>Act_Time</i> ($a, parameters$)	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

Next, we describe the *INPUT* and Flex’s scheduling algorithm. Details of runtime execution and monitoring are given in Section 4.7.

4.5.1 Inputs for Orchestration (*INPUT*)

Table 4.3 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 4.3 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 ways during orchestration, each of which will trigger a rerun of the scheduling algorithm from Section 4.5.2:

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.

4.5.2 Scheduling Algorithm

Given the current *INPUT*, the scheduling algorithm generates the current *Planned_Schedule* (recall Figure 4.4). 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, \dots, 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 4.6 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 4.6, 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.

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

Scheduling Algorithm (Input=*INPUT* (Table 4.3), Output=*Planned_Schedule*)

1. /* Total Ordering step: prioritizes earlier snapshots & longer run times */
2. Order $\langle a_i, s_j \rangle$ entries in *AS_Mapping* so that $\langle a, s \rangle$ precedes $\langle a', s' \rangle$ if:
 - (a) s is an earlier snapshot than s' , OR
 - (b) $s=s'$ AND $Act_Time(a, \{s\}) \geq Act_Time(a', \{s'\})$
3. /* No Preemption step: running actions are never preempted */
4. For each $\langle a_i, s_j \rangle \in AS_Mapping$ from first to last {
5. if ($status(\langle a_i, s_j \rangle) = \text{"RUNNING on host } h_k\text{"}$) {
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 $\langle a_i, s_j \rangle \in AS_Mapping$ from first to last {
9. if ($status(\langle a_i, s_j \rangle) = \text{"WAITING"}$ AND $Act_Time(a_i, \{s_j\}) = \infty$ AND
10. $\nexists s: \langle a_i, s \rangle \in AS_Mapping$ with $status(\langle a_i, s \rangle) = \text{"RUNNING on host } h\text{"}$) {
11. Let h_k be the first free host in $h_1 \dots h_n$ 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 $\langle a_i, s_j \rangle \in AS_Mapping$ in the order from first to last with $status(\langle a_i, s_j \rangle) = \text{"WAITING"}$ {
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) $end(h_k) + Act_Time(a_i, \{s_j, h_k\}) \leq \text{Deadline}$;
 - /* when there is no estimate yet of a_i 's running time */
 - (b) $Act_Time(a_i, \{s_j, h_k\}) = \infty$ AND $end(h_k) < \text{Deadline}$;
- }} /* end of loop for host h_k */
19. if $\langle a_i, s_j \rangle$ 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;

Continued on the next page

FIGURE 4.6: Flex's scheduling algorithm

```

Function Add_To_Planned_Schedule (action  $a$ , snapshot  $s$ , host  $h$ ) {
23. if ( $status(\langle a,s \rangle) = \text{“RUNNING on host } h\text{”}$ ) {
24.   Add  $\langle a,s \rangle$  as the head of  $h$ 's list in  $Planned\_Schedule$ ; }
   /*  $status(\langle a,s \rangle)$  is currently “WAITING” */
25. else if (host  $h$  is currently free) {
26.   Add  $\langle a,s \rangle$  as the head of  $h$ 's list in  $Planned\_Schedule$ ;
27.   Set  $status(\langle a,s \rangle)$  to “RUNNING on host  $h$ ”;
28.    $end(h) = NOW + Act\_Time(a, \{s, h\})$ ; }
29. else {
30.   Add  $\langle a,s \rangle$  to the end of  $h$ 's list in  $Planned\_Schedule$ ;
31.    $end(h) = end(h) + Act\_Time(a, \{s, h\})$ ; }
}

```

FIGURE 4.6: Flex's scheduling algorithm

No-Preemption Step: This step is presented in Lines 3-6 in Figure 4.6. 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 4.6. 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) into 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 4.6 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 . A new host will be added, subject to the Budget, if no such host can be found. 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 across these actions; which can reduce experiment running times significantly as we will see in Section 4.8.

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.

4.6 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 4.7.

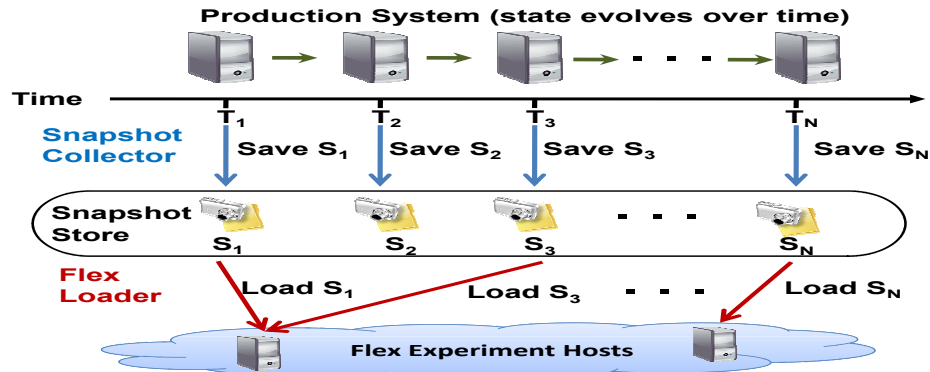


FIGURE 4.7: Snapshot management and Flex loading

4.6.1 Snapshot Collection on the Production DB

A snapshot is collected as follows (Running MySQL on Amazon EC2 with EBS):

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.

4.6.2 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. 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.

4.7 Implementation of Flex

Figure 4.8 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.

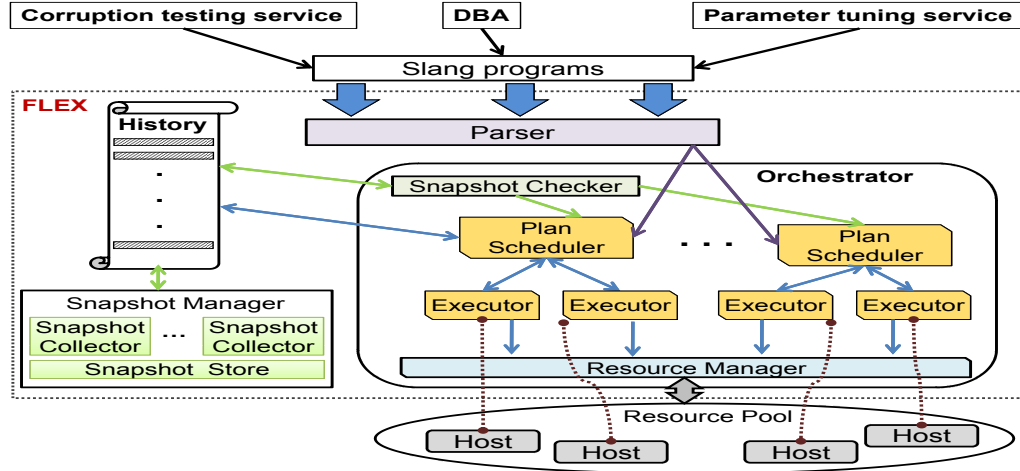


FIGURE 4.8: Architecture of the overall Flex platform

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 4.5.2 is run to generate the *Planned_Schedule*. The Scheduler will retrigger the scheduling algorithm if any of the events described in Section 4.5.1 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 4.6 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 4.3) 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 4.3).

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. In future, we plan to port the catalog to a distributed database or NoSQL engine.

Figure 4.8 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 4.6 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., Bodik et al., 2009; Borisov et al., 2011; Chaudhuri et al., 2009; Duan et al., 2009b; Haftmann et al., 2005; Yagoub et al., 2008; Zheng et al., 2009). Flex can benefit each tool—if the tool runs on Flex as a Flex Service—in multiple ways:

- Providing a common declarative 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 4.8.4 gives a case study based on one such tool called *Amulet* (see Chapter 2 and Borisov et al. (2011)) that we have ported to run as a Flex Service.

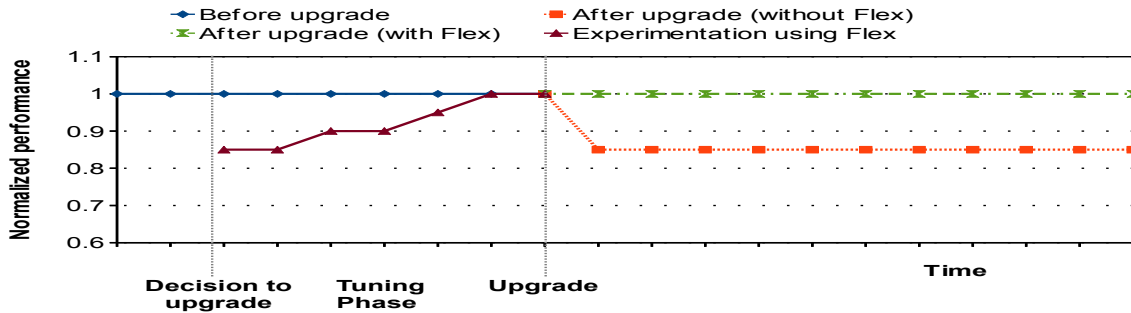


FIGURE 4.9: A performance degradation experienced after upgrading the production MySQL database instance, and how Flex helps to avoid the problem

4.8 Evaluation

This section presents a comprehensive evaluation of Flex. This evaluation uses the Amazon Web Services (AWS) cloud platform 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.

4.8.1 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 4.9 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 4.1) 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 4.1). 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 (Duan et al., 2009b). 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 4.10. The z-axis represents the throughput that the MySQL database can

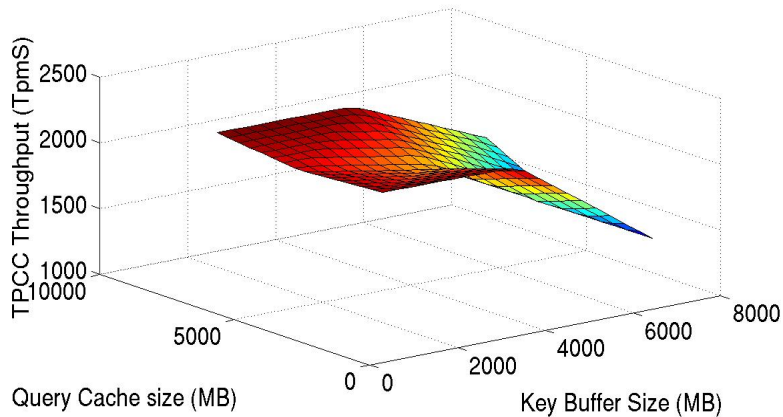


FIGURE 4.10: TPC-C throughput for various configurations

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.

4.8.2 Impact of Scheduling Algorithm

Recall the DBA’s task presented in Section 4.3 and its Slang program from Figure 4.2. 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 4.2).

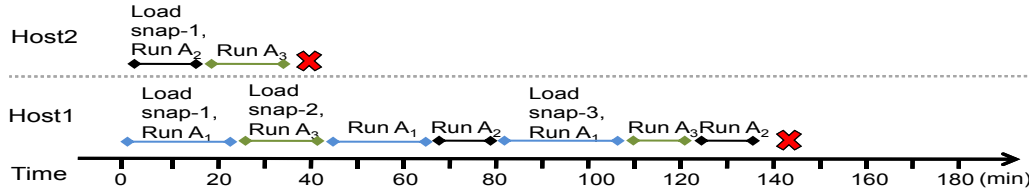
Schedule and Execution: Figure 4.3 presents the actual execution of the Slang program. (All execution timelines are drawn to scale to represent the actual execution of the actions.) None of the actions have been run before by Flex, so there is no history information in the History catalog. Thus, the 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.

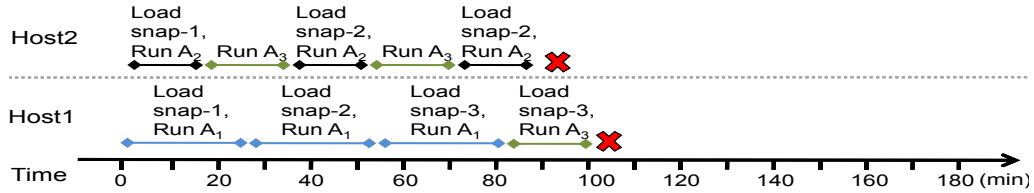
Action A_2 on snap-1 completes first, and gives a better model for action A_2 . This event triggers the scheduling algorithm. The scheduling algorithm identifies that there are no actions with unknown models and in the “*WAITING*” state; thus, it proceeds with the Greedy Packing step. Note that actions 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 4.3 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.

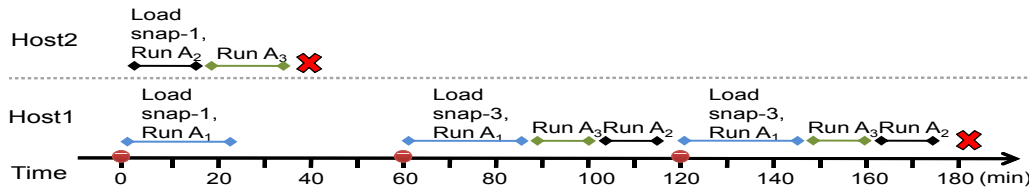
We further explore the impact of the scheduling algorithm by tightening the objectives and increasing the amount of data that needs to be verified from the first DBA task. In this way, we force the decisions of the scheduling algorithm to have a higher impact on the overall execution. The setup for this scenario (we refer to it as Scenario1) is the same as before except for: Lineitem is now 11GB and Order is



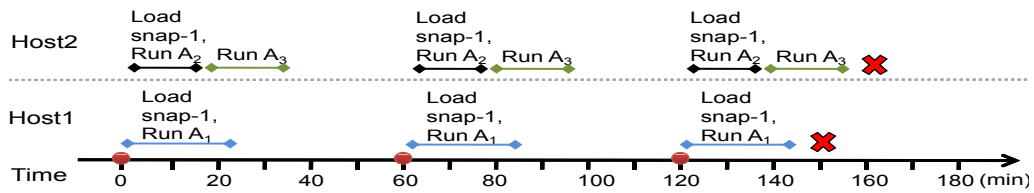
(a) Scenario1 execution with Flex scheduling



(b) Scenario1 execution with FCFS scheduling



(c) Scenario2 execution with Flex scheduling



(d) Scenario2 execution with FCFS scheduling

FIGURE 4.11: Evaluation of the scheduling algorithms for Scenario1 and Scenario2

7.5GB. The Budget was changed to 2 hosts of EC2 m1.large type, and Deadline is now 3 hours (180 minutes).

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 4.5 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,

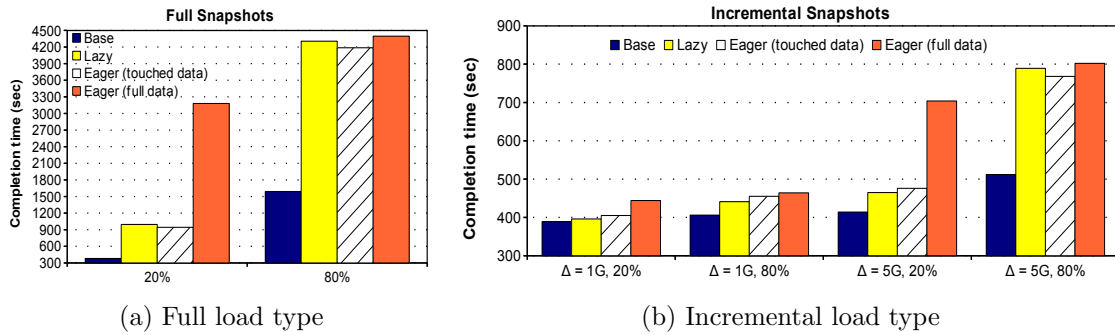


FIGURE 4.12: Running basic action a on snapshots with incremental and full load types

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 4.11a and Figure 4.11b respectively.

As seen in Figure 4.11a and Figure 4.11b, 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 ad-

vantage because it exploits the short execution times of actions that have already preloaded data; see action A_3 on snap-3 in Figure 4.11a.

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. Figure 4.11c and Figure 4.11d 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.

4.8.3 Impact of Snapshot Loading Techniques

In Section 4.6, 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 4.12a 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) and 80% (40GB) as represented on the x-axis. The bar “Lazy” 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 4.12b 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 $\Delta = 5$ GB, 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.

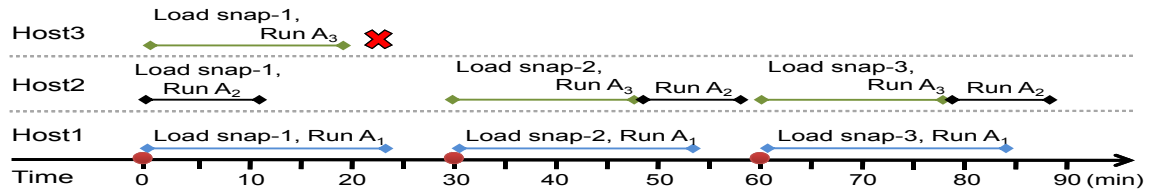
In summary, incremental + lazy is a robust strategy that outperforms or is equal to the other options, followed by incremental + eager, full + lazy, and last full + eager. However, recall from Section 4.6 that incremental + lazy cannot be used for all actions. If an action needs trustworthy measurements of running time as what would have been observed on the production database instance, then the eager mechanism is mandatory.

4.8.4 How Flex Reduces Development Effort

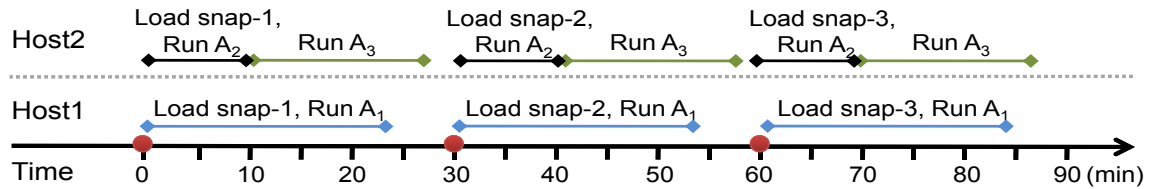
To study the utility of Flex as a platform for rapid development of testing and tuning applications, we ported the *Amulet* testing tool (see Chapter 2) to run as a Flex Service. Amulet aims to verify the integrity of stored data proactively and continuously.

Table 4.4: Lines of code in standalone Amulet

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



(a) Amulet as a Flex service



(b) Standalone Amulet

FIGURE 4.13: Amulet running as standalone and as a Flex service

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 4.4 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 4.13 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 with less than 100 lines of code is all it takes to obtain the monitoring data needed for Figure 4.10. Adding this program to a library makes it reusable by other Flex users.

⁵ An order of actions can be enforced using the $\langle a_i, s_j, h_k \rangle$ triples form of mapping; see Section 4.4.

4.8.5 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.

4.9 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 (Kohavi et al., 2010). Facebook has a related, but architecturally different, initiative called *Dark Launch* (Facebook Dark Launch). 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 targeting experiments on live applications or users, leading to a very different architecture.

Salesforce’s *Sandbox* (Salesforce Sandbox) initiative underscores the need for platforms like Flex. Sandbox provides a framework that developers can use to create replicas of the production data for testing purposes. The *Dexterity* automated testing framework (Farrar, 2010) verifies database correctness and performance by performing regression or ad-hoc testing based on the database schema. *JustRunIt* (Zheng et al., 2009) is an automated framework for experimentation. JustRunIt uses vir-

tual 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 (Oracle Online Index Rebuild), database migration (Elmore et al., 2011), 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 4.8.4 described our experiences in porting the Amulet testing tool to run as a Flex Service. We are currently porting the HTPar (Haftmann et al., 2005) regression testing tool to Flex. iTuned (Duan et al., 2009b) is a tuning tool for database configuration parameters. To find a good configuration from an unknown tuning surface like Figure 4.10, iTuned

proceeds iteratively: each iteration issues a set of experiments with different server configurations to collect performance data.

4.10 Future Work

This chapter presented Flex, a platform that enables efficient experimentation for trustworthy testing and tuning of production database instances. Flex gives DBAs a declarative 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 (SLA) violations. Based on the workload observed, the service will use tuning tools like index wizards (Agrawal et al., 2001; Introduction to Oracle Index Tuning Wizard) and iTuned (Duan et al., 2009b) to come up with potential fixes. The service will perform experiments to find and validate the best fix, which will then be applied to the production database to complete the loop.
- *Extending Flex to support parallel databases:* A promising direction is the integration of Flex with *Mesos* (Hindman et al., 2011) which is a cluster manager that provides efficient resource isolation and sharing across distributed applications. 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 (e.g., Galanis et al., 2008) or the *UpSizeR* that enables data sizes to be scaled up or down in a trustworthy manner (UpSizeR). 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.

Conclusions and Future Work

This dissertation outlined our contributions towards addressing the growing gap between the growth of systems versus the SAs who will manage these systems. We embraced the approach of reducing the burden of the SAs by automation of their tasks (recall from Section 1, Approach II). In particular, we presented tools that automate SA's tasks in the following three categories:

- Monitoring: **Amulet**, a tool that continuously monitors and proactively detects problems on the production system.
- Diagnosing: **DiaDS**, a tool that finds root causes of problems in a multi-tier system.
- Testing: **Flex**, a system that provides easy and flexible running of experiments in production like environments.

As outlined in Chapter 1, a complete solution for Approach II involves creating a self-managing system. A self-managing system needs to automate and integrate all the phases in the system managing cycle (outlined in Figure 5.1a). Furthermore,

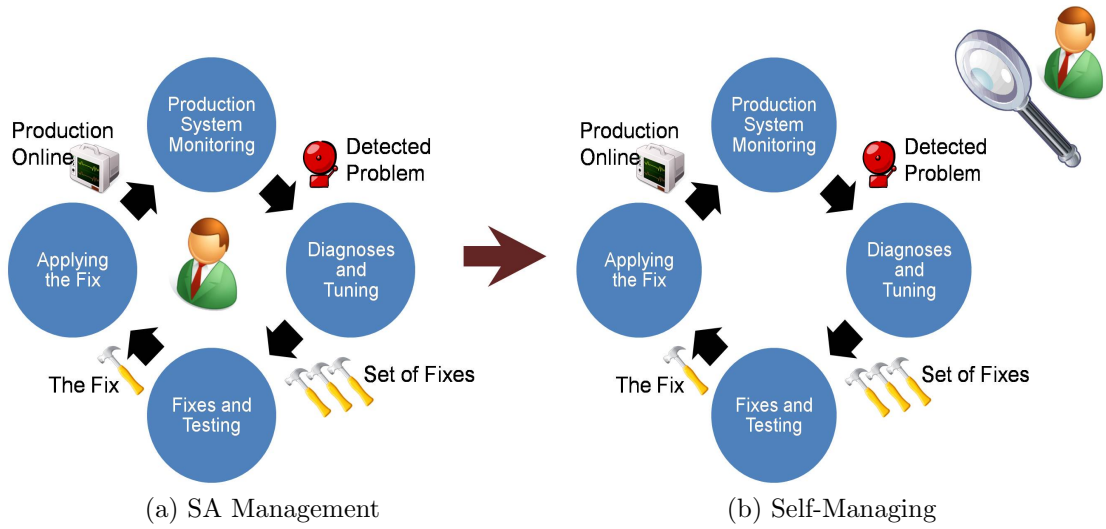


FIGURE 5.1: Overview of the system management cycle.

a self-managing system needs to initiate the transition between phases as well as provide the appropriate information from one phase to the next.

The first phase in the self-managing cycle is the Production Monitoring phase. This phase keeps track of the health of the system and collects monitoring data. When a problem occurs, i.e., the system is not able to meet its objectives, then the system should move to the next phase, namely Diagnoses and Tuning. The Diagnosing and Tuning phase takes as input monitoring data as well as the problem. This phase analyzes the problem and identifies a set of possible fixes (*fix* is any action that will resolve the problem on the production system - e.g., change of a configuration parameter, index creation, repair tool invocation, or system restart). As a next step, the system should go to the Fixes and Testing phase. This phase takes as an input the set of fixes identified in the previous phase, and selects the the fix that: (i) will fix the problem on the production system, and (ii) has the smallest amount of side effects. The fix selection process applies the set of fixes on a production-like environment, and examines the results to identify the best candidate. After the completion of this phase, the selected fix is sent to the next phase, namely

Applying the Fix. This phase applies the fix on the production system and ensures that all the SA's objectives are met. Once the fix is applied, the system transfers back to the Production Monitoring phase which completes the self-managing cycle. A production system that is self-managing will shift the role of the SAs from the main driving force to a supervisor position (depicted in Figure 5.1b).

Our contributions are in the automation of the Production Monitoring, Diagnosing, and Testing phases. Our tools described in Chapters 2 to 4 automate the tasks in these phases and function without SA intervention. However, more challenges need to be addressed to complete the self-managing cycle. Important future work includes: (i) extensions to these phases, (ii) addressing the challenges involved in the Applying the Fix phase, and (iii) addressing the challenges of integrating all the phases together.

Bibliography

- Admin's Choice (2009), *Checking and Repairing Unix File system with fsck*, <http://adminschoice.com/repairing-unix-file-system-fsck>.
- Agrawal, S., Chaudhuri, S., Kollar, L., and Narasayya, V. (2001), *Index Tuning Wizard SQL Server 2000*, <http://technet.microsoft.com/library/Cc966541>.
- Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. (2003), "Performance Debugging for Distributed Systems of Black Boxes", in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.
- Amazon Host Types, <http://aws.amazon.com/ec2/#instance>.
- Amazon Web Services, <http://aws.amazon.com>.
- Aranya, A., Wright, C. P., and Zadok, E. (2004), "Tracefs: A File System to Trace Them All", in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
- Babcock, B. and Chaudhuri, S. (2005), "Towards a Robust Query Optimizer: A Principled and Practical Approach", in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.
- Bairavasundaram, L. N., Goodson, G. R., Schroeder, B., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2008), "An Analysis of Data Corruption in the Storage Stack", in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
- Basu, S., Dunagan, J., and Smith, G. (2007), "Why Did My PC Suddenly Slow Down?" in *Proceedings of the USENIX workshop on Tackling computer systems problems with machine learning techniques (SYSML)*.
- Biazetti, A. and Gajda, K., "Achieving Complex Event Processing with Active Correlation Technology", <http://www.ibm.com/developerworks/library/ac-acact/index.html>.
- Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M. I., and Patterson, D. A. (2009), "Automatic Exploration of Datacenter Performance Regimes", in *Automated Control for Datacenters and Clouds*.

- Borisov, N., Babu, S., Mandagere, N., and Uttamchandani, S. (2011), “Warding off the Dangers of Data Corruption with Amulet”, in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.
- BPEL (2011), *Business Process Execution Language*, https://en.wikipedia.org/wiki/Business_Process_Execution_Language.
- Brunette, G. (2008), *Data corruption in Amazon S3*, <http://glennbrunette.sys-con.com/node/991097/mobile>.
- Chaudhuri, S., Konig, A., and Narasayya, V. (2004), “SQLCM: a Continuous Monitoring Framework for Relational Database Engines”, in *Proceedings of the International Conference on Data Engineering (ICDE)*.
- Chaudhuri, S., Narasayya, V. R., and Ramamurthy, R. (2009), “Exact Cardinality Query Optimization for Optimizer Testing”, *PVLDB*.
- Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. (2004), “Path-based Failure and Evolution Management”, in *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*.
- Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J. (2004), “Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control”, in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. (2005), “Capturing, Indexing, Clustering, and Retrieving System History”, in *ACM symposium on Operating systems principles (SOSP)*.
- Costa, D., Rilho, T., and Madeira, H. (2000), “Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection”, in *Proceedings of the International Conference on Dependable Systems and Networks*.
- CouchDB Data Loss (2010), *Data corruption in CouchDB*, <https://news.ycombinator.com/item?id=1586486>.
- Cromar, S. (2011), *ZFS Management*, <http://www.princeton.edu/~unix/Solaris/troubleshoot/zfs.html>.
- Dageville, B., Das, D., Dias, K., Yagoub, K., Zait, M., and Ziauddin, M. (2004), “Automatic SQL Tuning in Oracle 10g.” in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.
- DB2 Check Utilities, “Checking data consistency with the CHECK utilities”, http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc.admin/db2z_usecheck2checkconsistent.htm.

DB2 Inspect Command, “DB2 Inspect command”, <http://publib.boulder.ibm.com/infocenter/db2luw/v8//topic/com.ibm.db2.udb.doc/core/r0008633.htm>.

db2ckbkp Tool, “DB2ckbkp tool”, http://webdocs.caspur.it/ibm/db2/8.1/doc/htmlcd/en_US/core/r0002585.htm.

db2dart Tool, “DB2Dart tool”, <http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/r0003477.htm>.

db2inspect Tool, “DB2Inspect tool”, <http://publib.boulder.ibm.com/infocenter/db2luw/v8//topic/com.ibm.db2.udb.doc/admin/r0008552.htm>.

DBMS Repair, “Using DBMS_REPAIR to Repair Data Block Corruption”, http://download.oracle.com/docs/cd/B19306_01/server.102/b14231/repair.htm.

DBVerify, “DBVerify tool for detecting block corruptions”, <http://itcareershift.com/blog1/2011/02/08/dbverify-tool-for-detecting-block-corruption-complete-reference-for-the-new-oracle-dba/>.

DBVerify Usage, “How to check Corruption in Database Using DBVerify”, <http://onlineappsdba.com/index.php/2008/07/03/how-to-check-corruption-in-database-using-dbverify/>.

Dias, K., Ramacher, M., Shaft, U., Venkataramani, V., and Wood, G. (2005), “Automatic Performance Diagnosis and Tuning in Oracle.” in *Proceedings of Conference on Innovative Data Systems Research (CIDR)*.

Digital Universe, “The Exploding Digital Universe”, www.emc.com/leadership/programs/digital-universe.htm.

Duan, S., Babu, S., and Munagala, K. (2009a), “Fa: A System for Automating Failure Diagnosis”, in *Proceedings of the International Conference on Data Engineering (ICDE)*.

Duan, S., Thummala, V., and Babu, S. (2009b), “Tuning Database Configuration Parameters with iTuned”, in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

Elmore, A. J., Das, S., Agrawal, D., and Abbadi, A. E. (2011), “Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms”, in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

EMC Control Center, <http://www.emc.com/data-center-management/controlcenter.htm>.

- Eseutil Tutorial, http://www.msexchange.org/articles_tutorials/exchange-server-2010/management-administration/eseutil-part1.html.
- Facebook Dark Launch, <http://on.fb.me/RWs00>.
- Farrar, D. J. (2010), "Schema-driven Experiment Management: Declarative Testing with Dexterity", in *Proceedings of the International Workshop on Testing Database Systems (DBTest)*.
- Fixing XFS, "Fixing XFS", http://mirror.linux.org.au/pub/linux.conf.au/2008/slides/135-fixing_xfs_faster.pdf.
- Galanis, L., Buranawanachoke, S., Colle, R., Dageville, B., Dias, K., Klein, J., Papadomanolakis, S., Tan, L. L., Venkataramani, V., Wang, Y., and Wood, G. (2008), "Oracle Database Replay", in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.
- Ganglia, "Ganglia Monitoring System", <http://ganglia.sourceforge.net/>.
- Garcia-Molina, H., Ullman, J., and Widom, J. (2001), *Database Systems: The Complete Book*, Prentice Hall, Upper Saddle River, New Jersey.
- Graefe, G. and Stonecipher, R. (2009), "Efficient Verification of B-tree Integrity", in *Proceedings of the Database Systems for Business, Technology, and Web (BTW)*.
- Gunawi, H. S., Rajimwale, A., Arpaci-Dusseau, A. C., and Arpaci-Susseau, R. H. (2008), "SQCK: A Declarative File System Checker", in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- Guo, H., Jones, D., Beckmann, J. L., and Seshadri, P. (2009), "Declarative Database Management in SQLServer", *PVLDB*.
- Hablador, P., "Should ZFS Have a fsck Tool?" http://www.osnews.com/story/22423/Should_ZFS_Have_a_fsck_Tool_.
- Haftmann, F., Kossmann, D., and Lo, E. (2005), "Parallel Execution of Test Runs for Database Application Systems", in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.
- Hewlett Packard Systems Insight Manager, <http://h18002.www1.hp.com/products/servers/management/hpsim/index.html>.
- Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., and Stoica, I. (2011), "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", in *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*.
- Hyperic, <http://www.hyperic.com/>.

- IBM Tivoli Network Manager, <http://www-01.ibm.com/software/tivoli/products/netcool-precision-ip>.
- IBM TotalStorage Productivity Center, <http://www-306.ibm.com/software/tivoli/products/totalstorage-data/>.
- Inspect vs db2dart, “Comparison of INSPECT and db2dart”, <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.trb.doc/doc/c0020763.html>.
- Introduction to Oracle Index Tuning Wizard, http://docs.oracle.com/cd/A97630_01/em.920/a88749/indxtun.htm.
- Isinteg, “Description of the Isinteg utility”, <http://support.microsoft.com/kb/182081>.
- Joukov, N., Traeger, A., Iyer, R., Wright, C. P., and Zadok, E. (2006), “Operating System Profiling via Latency Analysis”, in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- King Solomon’s Amulets, “King Solomon’s Amulets”, <http://www.kabbalah-corner.com/king-solomons-amulets.asp>.
- Kohavi, R., Crook, T., and Longbotham, R. (2010), “Online Experimentation at Microsoft”, in *Workshop on Data Mining Case Studies and Practice Prize*.
- Krioukov, A., Bairavasundaram, L. N., Goodson, G. R., Srinivasan, K., Thelen, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2008), “Parity Lost and Parity Regained”, in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
- Laurie, V., “How to Use Chkdsk”, <http://best-windows.vlaurie.com/chkdsk.html>.
- Magnolia (2009), *Data corruption at Ma.gnolia.com*, en.wikipedia.org/wiki/Gnolia.
- Manji, A., “Creating Symptom Databases to Rervice J2EE Applications in WebSphere Studio”, https://www.ibm.com/developerworks/websphere/library/techarticles/0407_manji/0407_manji.html.
- Mehta, A., Gupta, C., Wang, S., and Dayal, U. (2008), “Automatic Workload Management for Enterprise Data Warehouses”, *IEEE Data Engineering Bulletin*, 31.
- Mesnier, M. P., Wachs, M., Sambasivan, R. R., Zheng, A. X., and Ganger, G. R. (2007), “Modeling the Relative Fitness of Storage”, *SIGMETRICS Performance Evaluation*, 35.

myisamchk Command, “MySQL myisamchk command”, <http://dev.mysql.com/doc/refman/5.0/en/myisamchk.html>.

mysqlcheck Command, “MySQL mysqlcheck command”, <http://dev.mysql.com/doc/refman/5.0/en/mysqlcheck.html>.

Oracle Corrupted Backups (2007), *Oracle Corrupted Backups*, <http://forums.oracle.com/forums/thread.jspa?threadID=1102491\&tstart=45>.

Oracle HARD, “Oracle HARD Initiative”, http://www.dba-oracle.com/real_application_clusters_rac_grid/hard.html.

Oracle Health Monitor, “Running Health Checks with Health Monitor”, http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/diag007.htm.

Oracle Online Index Rebuild, <http://www.oracle-base.com/articles/9i/HighAvailabilityEnhancements9i.php#Online>.

Oracle Recover, “Detecting and Recovering from Database Corruption”, <http://mnsinger.wordpress.com/2006/08/19/chapter-30-detecting-and-recovering-from-database-corruption/>.

Oracle Repair, “Detecting and Repairing Data Block Corruption”, <http://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10739/repair.htm#i1006139>.

Oracle Upgrade Regression, <http://dbaforums.org/oracle/index.php?showtopic=8613>.

Parchive, “Parchive”, <http://parchive.sourceforge.net/>.

Parzen, E. (1962), “On Estimation of a Probability Density Function and Mode”, *Annals of Mathematical Statistics*, 33.

Pearl, J. (2000), *Causality: Models, Reasoning, and Inference*, Cambridge University Press, Cambridge, UK.

Perazolo, M., “The Autonomic Computing Symptoms Format”, <https://www.ibm.com/developerworks/autonomic/library/ac-symptom1/>.

Peter Zaitsev (2007), *MySQL upgrade from version 4 to 5*, <http://www.mysqlperformanceblog.com/2007/06/06/mysql-4-to-mysql-5-upgrade-performance-regressions/>.

Pollack, K. T. and Uttamchandani, S. (2006), “Genesis: A Scalable Self-Evolving Performance Management Framework for Storage Systems”, in *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

- PostgreSQL TPC-H Bug, http://blogs.oracle.com/jkshah/entry/postgresql_east_2008_talk_postgresql.
- Qin, Y., Salem, K., and Goel, A. K. (2007), “Towards Adaptive Costing of Database Access Methods”, in *Proceedings of the International Workshop on Self-Managing Database Systems (SMDB)*.
- Quirke, C. (2002), *Understanding ScanDisk*, <http://cquirke.mvps.org/9x/scandisk.htm>.
- Reiss, F. and Kanungo, T. (2003), “A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters”, in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.
- Running MySQL on Amazon EC2 with EBS, <http://aws.amazon.com/articles/1663>.
- Salesforce Sandbox, http://wiki.developerforce.com/index.php/An_Introduction_to_Environments.
- Schroeder, B., Pinheiro, E., and Weber, W. D. (2009), “DRAM Errors in the Wild: A Large-Scale Field Study”, in *SIGMETRICS*.
- Schwarz, T., Xin, Q., Miller, E., Long, D., Hospodor, A., and Ng, S. (2004), “Disk Scrubbing in Large Archival Storage Systems”, in *MASCOTS*.
- Shen, K., Zhong, M., and Li, C. (2005), “I/O System Performance Debugging Using Model-driven Anomaly Characterization”, in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST)*.
- SQL Server checkDB, “Microsoft SQL Server checkDB command”, <http://msdn.microsoft.com/en-us/library/aa258278%28v=sql.80%29.aspx>.
- SQL Server Restore Command, “Microsoft SQL Server restore command”, <http://msdn.microsoft.com/en-us/library/ms186858.aspx>.
- Stellar Phoenix Linux Data Recovery, “Stellar Phoenix Linux Data Recovery”, <http://www.data-recovery-linux.com/linux-ext2-ext3-reiserfs-recovery.php>.
- Subramanian, S., Zhang, Y., Vaidyanathan, R., Gunawi, H. S., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Naughton, J. F. (2010), “Impact of Disk Corruption on Open-Source DBMS”, in *Proceedings of the International Conference on Data Engineering (ICDE)*.
- Sun Microsystems ZFS (2006), *Solaris ZFS Administration Guide*, <http://docs.huihoo.com/opensolaris/solaris-zfs-administration-guide/html/index.html>.

- Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. (2006), “Stardust: Tracking Activity in a Distributed Storage System”, *SIGMETRICS Performance Evaluation*, 34.
- tpch (2009), *TPC Benchmark H Standard Specification*, TPC, <http://www.tpc.org/tpch/spec/tpch2.9.0.pdf>.
- Treynor, B., “Storage software update causes 0.02% of Gmail users to lose their emails”, <http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html>.
- Tripwire (2008), *Tripwire Tutorial*, <http://www.thegeekstuff.com/2008/12/tripwire-tutorial-linux-host-based-intrusion-detection-system/>.
- UpSizeR, <http://upsizer.comp.nus.edu.sg/upsizer/>.
- Vargas, A. (2008), *RMAN Hands on*, <http://static7.userland.com/oracle/gems/alejandrovargas/RmanHandsOn.pdf>.
- VMWare Virtual Center, <http://www.vmware.com/products/vi/vc/>.
- Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. (2004), “Automatic Misconfiguration Troubleshooting with PeerPressure”, in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
- Weikum, G., Moenkeberg, A., Hasse, C., and Zabback, P. (2002), “Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering”, in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.
- Wood, T., Cecchet, E., Ramakrishnan, K., Shenoy, P., van der Merwe, J., and Venkataramani, A. (2010), “Disaster Recovery as a Cloud Service: Economic Benefits and Deployment Challenges”, in *HotCloud*.
- Yagoub, K., Belknap, P., Dageville, B., Dias, K., Joshi, S., and Yu, H. (2008), “Oracle’s SQL Performance Analyzer”, *DEB*, 31.
- Yemini, S. A., Kliger, S., Mozes, E., Yemini, Y., and Ohsie, D. (1996), “High Speed and Robust Event Correlation”, *IEEE Communications Magazine*, 34.
- Zhang, S., Cohen, I., Symons, J., and Fox, A. (2005), “Ensembles of Models for Automated Diagnosis of System Performance Problems”, in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*.
- Zhang, Y., Rajimwale, A., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2010), “End-to-end Data Integrity for File Systems: A ZFS Case Study”, in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

Zheng, W., Bianchini, R., Janakiraman, G. J., Santos, J. R., and Turner, Y. (2009), “JustRunIt: Experiment-Based Management of Virtualized Data Centers”, in *USENIX*.

Zhou, S., Costa, H. D., and Smith, A. J. (1985), “A File System Tracing Package for Berkeley UNIX”, Tech. rep.

Biography

Nedyalko Krasimirov Borisov was born and raised in Bulgaria. He graduated in 2002 from the Math High School “PMG Nancho Popovich” in Shoumen. During his high-school years he participated in Mathematics and Computer Science competitions on a regional and national level.

After High school, Nedyalko attended Sofia University in Sofia, Bulgaria where he received his bachelor degree in Computer Science. While he was an undergraduate, he worked for Sciant Ltd as a full-time Software Engineer. He started as a Trainee Software Engineer and was promoted to Senior Software Engineer withing 2.5 years.

In August 2007, he enrolled as a graduate student in the Computer Science Department at Duke University in Durham, North Carolina. In 2009, he was one of the finalist for the IBM PhD Fellowship. In 2010, he received the Outstanding Research Initiation Project Award from the Computer Science Department. Nedyalko obtained his Master of Science in May 2010, focusing on the integrated management of the persistent-storage and data-processing layers in data-intensive computing systems. He continued this work with his advisor Shivnath Babu for his doctoral research. After graduation, he will pursue an industrial career at Facebook Inc.