Warding off the Dangers of Data Corruption with Amulet

Nedyalko Borisov
Duke University
nedyalko@cs.duke.edu

Shivnath Babu*
Duke University
shivnath@cs.duke.edu

Nagapramod Mandagere
IBM Almaden Research
nmandage@us.ibm.com

Sandeep Uttamchandani
IBM Almaden Research
sandeepu@us.ibm.com

ABSTRACT

Occasional corruption of stored data is an unfortunate byproduct of the complexity of modern systems. Hardware errors, software bugs, and mistakes by human administrators can corrupt important sources of data. 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 introduce the *Amulet* system that addresses the problem of verifying the correctness of stored data proactively and continuously. To our knowledge, Amulet is the first system that: (i) gives administrators a declarative language to specify their objectives regarding the detection and repair of data corruption; (ii) contains optimization and execution algorithms to ensure that the administrator's objectives are met robustly and with least cost, e.g., using pay-as-you cloud resources; and (iii) provides timely notification when corruption is detected, allowing proactive repair of corruption before it impacts users and applications. We describe the implementation and a comprehensive evaluation of Amulet for a database software stack deployed on an infrastructure-as-a-service cloud provider.

Categories and Subject Descriptors

H.2.0 [**Datbase Management**]: Data Protection

General Terms

Management, Reliability, Verification

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 administrators have to deal with occasionally; often when they are least prepared [6, 11, 14, 17, 25, 30]. 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 administrators 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* [14]. Bugs in storage software [11], OS device drivers, and higher-level layers like load balancers [21] and database software [6] 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 [6]. A recent bug triggered by a storage software update caused 0.02% of Gmail users to lose their email data (which had to be restored from tape) [11].
- 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 [3, 22].
- 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 [21].

Data loss can have serious consequences. 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 social-bookmarking site Ma.gnolia.com out of business [15].

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 [18]. 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 [14] as well as plenty of anecdotal evidence show that problems due to corruption happen, and more frequently than expected [3, 22]. A particularly dangerous scenario that the authors as well as others (e.g., [15, 17]) have come across involves the propagation of corruption from the production system to critical backups.

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. Table 1 lists popular tests for detecting and repairing corruption that occur in different system layers 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 [25].
- Barring few exceptions, tests have been developed to be run offline when the system is not serving a workload. If a workload

*Supported by NSF grants 0964560 and 0917062

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

SIGMOD'11, June 12–16, 2011, Athens, Greece.

Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

Test	System	Description	Does Repair	Runs Online
Application Level				
Eseutil, Isinteg	Exchange Server	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 [9]	Yes	No
Tripwire	Any file	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 [26]	No	No
par, par2	Any file	Uses an error correcting code to create parity data. Checks whether the given file's content matches its stored parity data [20]	Yes	No
Database Level				
myisamchk	MySQL	Uses checksums and structure rules (flags, table metadata) to check whether data pages and records are correct and indexes point to correct records. This suite contains 5 distinct tests that do increasingly rigorous and time-consuming checks [16, 25]	Yes	No
DBVerify	Oracle	Uses checksums and structure rules (head & tail info) to check the database content and data pages. Can check backups [19]	No	No
db2dart	DB2	Uses checksums and structure rules (page headers, properties of records) to check the database, data pages, and records [7]	Yes	No
Checkdb, Checktable	SQL Server	Uses structure rules (page headers, properties of indexes) to check whether data pages are stored correctly and index entries point to the correct records [24]	Yes	No
File-System Level				
xfs_check, xfs_repair	XFS	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, and inode maps [28]	Yes	No
fsck	ext3, ext4	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 [10]	Yes	No
ScanDisk, Chkdsk	FAT*, NTFS	Uses file structures to check the disk headers, file content, file attributes, lost disk space, and file crosslinks [5]	Yes	No
zpool scrub	ZFS	Uses built-in block checksums and block replication mechanisms in the ZFS file-system to check file content for errors [29, 30]	Yes	Yes
Storage Level				
Scrubbing	Disks	Uses checksums stored at the level of disk media blocks to verify that each block's content matches its checksum [23]	No	No
Scrubbing	RAID	Uses stored parity information or replicas at the RAID level (instead of per disk) to verify the content of each data block [14]	Yes	Yes

Table 1: Data integrity tests to detect and possibly repair data corruption at different levels of the database software stack

	Description of Example Objectives in English
1	If the *myisamchk* test detects corruption in the *lineitem* 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 *fsck* 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?

Table 2: Examples of objectives that an administrator may have regarding timely detection and repair of data corruption

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

Administrators usually have specific objectives in mind for proactive detection and repair of data corruption. Table 2 gives examples of such objectives. To our knowledge, no system today helps administrators specify objectives like these easily, and automates the 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.

1.1 Amulet: Challenges and Overview

A typical *database software stack* that production systems use is shown in Figure 1. Different levels of the software stack maintain

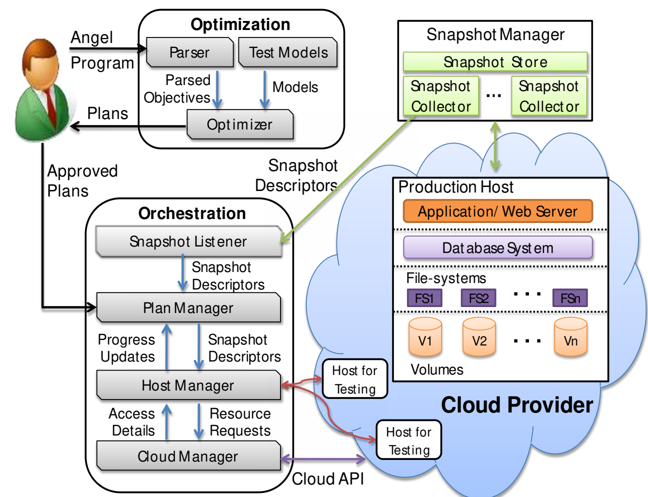


Figure 1: Overview of Amulet's optimization and orchestration

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-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. Table 1 lists commonly-used tests at each level of the database software stack. For example, MySQL's *myisamchk* suite contains five different tests invoked through distinct invoca-

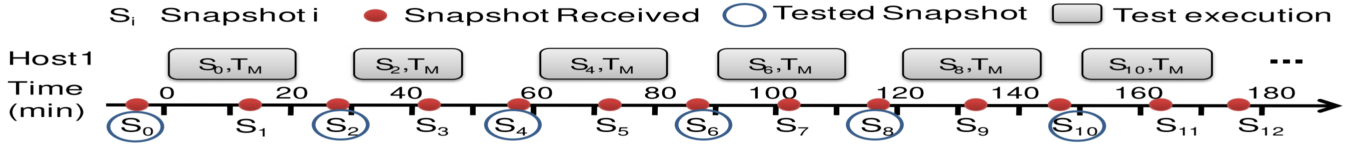


Figure 2: Actual execution timeline, in minutes, of a testing plan in Amulet for Example 1 from Table 2. Each box denotes a run of the myisamchk (T_M) test on the respective snapshot S_i. The horizontal width of each box corresponds to the test execution time.

tion options: fast, check-only-changed, check (default), medium-check, and extended-check. These tests apply checks of increasing 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 Table 1 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 paper, 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 1, testing hosts are different from the production host to avoid performance problems.
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 poses a nontrivial language design problem. A dissection of the examples in Table 2 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 4 and Table 4.

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 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 shows an actual execution timeline of a testing plan P for an Angel program corresponding to Example 1 from Table 2. Plan P uses one testing host that runs the myisamchk 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 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 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 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, cur-

[1] Production deployments that need near-real-time disaster recovery take snapshots regularly and store them on cloud storage [27].

rently, 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 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 7 will present comprehensive experimental results from a prototype of Amulet running on the Amazon cloud.

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.

2.1 The Dangers of Data Corruption

The authors of [3] 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.

The authors of [22] 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 [25]. 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 [30]. 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 Dealing with Data Corruption

The techniques categorized as the first line of defense in Section 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 [3]. For example, the authors of [14] 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. 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 Table 1 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.

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 in [12].

Amulet's goal of early detection and repair of data corruption forms a crucial part of disaster recovery planning. The authors of [27] 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.

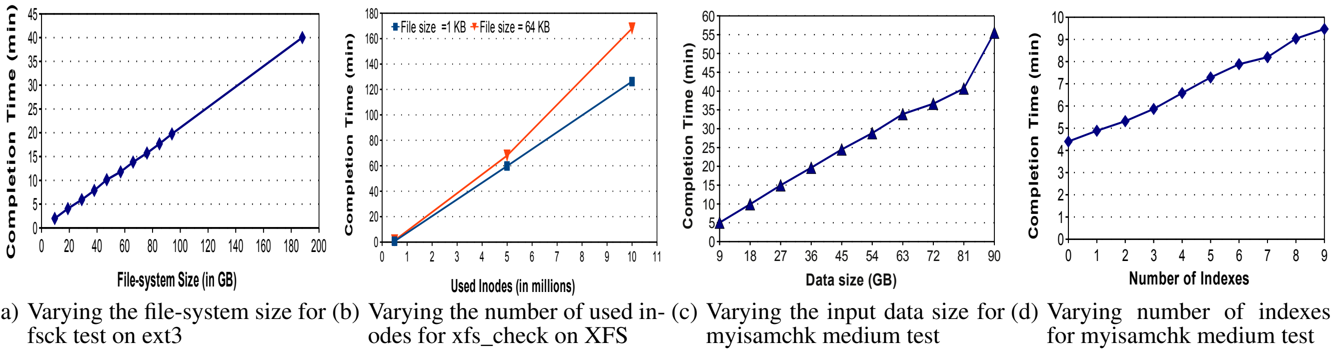
The concept of declarative system management is gaining currency. Chef, Puppet, and Microsoft SQL Server's policy-driven manager are now popular tools that take declarative specifications as input, and then configure and maintain systems automatically [13]. However, unlike Amulet, these tools do not support objectives that are specified declaratively and optimized automatically.

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 xfs_check tests from Table 1. 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 involves verifying the superblock, inodes, and free block list, while the data test does a full scan to find all bad blocks.

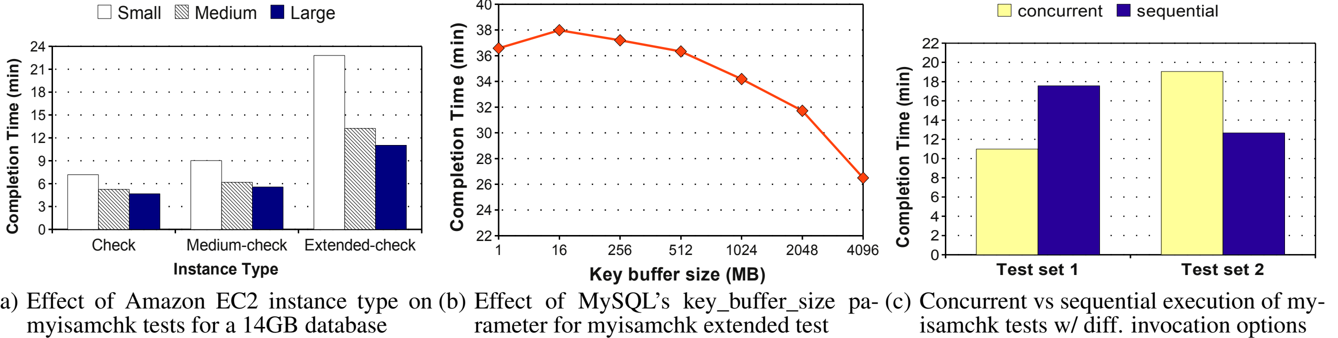
3.1 Data-dependent Attributes

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



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



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

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 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 3(a), 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 3(b), and is independent of the total file-system size.

For the myisamchk tests, the total input data size to the test is the primary data-dependent attribute. Figure 3(c) 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 3(d).

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.

Figure 4(a) 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.

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 4(b) shows the variation in the execution time of the myisamchk extended test for input data of size 12 GB.

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.

Name	Description
O_1-O_n	SSO, RPO, SIO, TCO, or CO objectives specified per volume in an Angel program (Table 4 gives a summary)
O_{opt}	Optimization objective for a volume in an Angel program
t, s, h, τ, 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 4)
d_{co}	The maximum cost budget in a CO (see Table 4)
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
$\mathit{ExecTime}(t)$	Execution time of test t
$\mathit{Time}(s_i)$	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
$\mathit{start}(h)$	Time when host h is first used in the plan window
$\mathit{end}(h)$	Time when host h will finish its last scheduled test for the plan window ($\mathit{end}(h)$ can be $> P_W$)
$\mathit{cost}(P_S)$, $\mathit{cost}(h)$	Cost incurred for the plan schedule P_S or a host h for one plan window

Table 3: Notation used in the paper

Concurrent execution of tests on the same host: Concurrent execution makes test execution times difficult to estimate. Figure 4(c) illustrates this complexity. Test sets 1 and 2 are runs of two myisamchk 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,[2] 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.

4. ANGEL DECLARATIVE LANGUAGE

Recall from Section 1.1 that an Angel program specifies tests and the objectives to be met while running the tests. We now discuss the main statements in Angel. Table 4 provides a summary.

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). 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 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 administrator 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 will always run t on testing hosts of that type.

[2]From our experience, most administrators prefer to use standard tests that come with each system, rather than writing new tests.

Name	Simplified Specification Syntax
Test	`Test(Data:` *data*, `Host:` *host*, exec scripts, ...)
Input data for test	`Data(Volume:` V, type, properties, ...)
Host to run test	`Host`(type, setup scripts, ...)
Volume	`Volume(Host:` production host where V is located, path on host, volume id, properties, ...)
Repair action	`Repair(Test:` t, t's return code, exec scripts, ...)
SSO: Safe Snapshot Objective	List of tests $\{t_1, t_2, \ldots, t_k\}$, for volume V
RPO: Recovery Point Objective	`Recovery_point` $\leq \tau$, for volume V
SIO: Snapshot Interval Objective	`Snapshot_interval` *Op* τ, for volume V
TCO: Test Count Objective	`Test_count`(t) *Op* x, in time interval τ, for volume V
CO: Cost Objective	`Cost` $\leq d$, 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

Table 4: 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

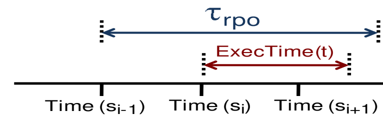


Figure 5: Illustration of RPO

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 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 the full list of tests $t_1,\ldots,t_k$ to be run on a given snapshot s for V before s can be labeled *corruption-free*. If none of the tests $t_1,\ldots,t_k$ find corruption in s, then s is labeled corruption-free. If a test t_i detects a corruption in s, then s is labeled *corrupted* and any repair action associated with t_i will be run on s.

Recovery Point Objective (RPO): When corruption is detected in the data in a volume V, it is useful to have immediate access to a tested recovery point for V from the recent past. A recovery point is a corruption-free snapshot of V from which the database software stack can be brought back online quickly. Recovery points are an essential part of disaster recovery planning strategies [27].

Figure 5 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 administrator 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 5). Using notation from Table 3, the RPO mandates:

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

An RPO together with an SSO with a list of tests $t_1,\ldots,t_k$ is a powerful combination to express recovery points. Now Amulet should ensure that all of $t_1,\ldots,t_k$ will finish on s_i in time $\leq \tau_{rpo}$

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 Fig. 7 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 5.3);
3. Use Fig. 8 to pick the test-to-snapshot mapping P_M for O_1-O_n,P_W,P_I;
4. Pick test execution schedule P_S for P_W,P_I,P_M (details given in [4]);
5. P_R is available from the CO in O_1-O_n, or a default is used;

Figure 6: Finding the least-cost plan for given objectives O_1-O_n

− (*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 administrator 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 here.

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. The CO applies to the entire testing plan that the Optimizer generates for V. 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 7 describes the pricing model that we use in our evaluation.

The CO also 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 estimates of test execution times by the models used in the Optimizer. Section 6 will discuss how the Orchestrator uses the reserved cost budget to provision hosts on demand to handle these unpredictable events. By default, the reservation is set to the cost of using one testing host.

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.

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}.*

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 5.3–5.5 by describing the stages in which the cost-optimal planning algorithm works. As illustrated in Figure 6, 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 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. This algorithm, discussed in Section 5.6, repeatedly invokes the cost-optimal planning algorithm with a series of increasingly stricter objectives until no valid plan can be found. The optimal plan can be identified at that point.

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

Figure 7 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.

5.3 Selecting the Plan Window

Recall from Section 4 and Table 4 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) *ExecTime*(t) for each test t specified in an SSO or TCO objective. Picking $P_W >$ *ExecTime*(t) for all tests (i.e., Case (b) above) is needed to ensure the *sustainability* of schedules as we will explain in Section 5.5.

Once P_W has been determined, the corresponding parameters in

Selecting the Plan Snapshot Interval P_I in a Testing Plan
Inputs: Objectives O_1-O_n (with syntax from Table 4)
1. P_I^{min} = Snapshot interval from the Snapshot Manager;
2. $P_I^{max} = \infty$; $\tau_{rpo} = \infty$;
3. if ($O_1, \ldots, O_n$ contains RPO: `Recovery_point` $\leq \tau$) {
4. τ_{rpo}=τ; $P_I^{max} = \frac{\tau_{rpo}}{2}$; /* test ≥2 snapshots in τ_{rpo} to meet RPO */}
5. for (every Objective O in $O_1, \ldots, O_n$) {
6. if (O is SSO: List of tests $\{t_1, \ldots, t_k\}$) {
7. for (Test t in $t_1, \ldots, t_k$)
8. P_I^{max} = Min[P_I^{max}, τ_{rpo} - *ExecTime*(t)]; } /* Equation 1 */
9. if (O is TCO: `Test_count`(t) $\geq x$, in time interval τ)
10. P_I^{max} = Min[P_I^{max}, τ_{rpo} - *ExecTime*(t), $\frac{\tau}{x}$]; /* Equation 1 */
11. if (O is SIO: `Snapshot_interval` $\geq \tau$)
12. P_I^{min} = Max[P_I^{min}, τ];
13. if (O is SIO: `Snapshot_interval` $\leq \tau$)
14. P_I^{max} = Min[P_I^{max}, τ];
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 7: Selection of P_I (notation used from Tables 3 and 4)

Selecting the Test-to-Snapshot Mapping P_M in a Testing Plan
Inputs: 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 $O_1, \ldots, O_n$) {
2. $COUNT_t^{min}$ = 0; $COUNT_t^{max} = \frac{P_W}{P_I}$; }
3. for (every Objective O in $O_1, \ldots, O_n$) {
4. if (O is SSO: List of tests $\{t_1, \ldots, t_k\}$) {
5. for (Test t in $t_1, \ldots, t_k$) /* t has to run on all $\frac{P_W}{P_I}$ snapshots */
6. $COUNT_t^{min}$ = Max[$COUNT_t^{min}$, $\frac{P_W}{P_I}$]; }
7. if (O is TCO: `Test_count`(t) $\geq x$, in time interval P_W)
8. $COUNT_t^{min}$ = Max[$COUNT_t^{min}$, x];
9. if (O is TCO: `Test_count`(t) $\leq x$, in time interval P_W)
10. $COUNT_t^{max}$ = Min[$COUNT_t^{max}$, x];
11. }
12. $P_M = \emptyset$;
13. for (every test t referenced in an SSO or TCO in $O_1, \ldots, O_n$) {
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 8: Selection of P_M (notation used from Tables 3 and 4)

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.

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 8 shows the steps involved. For each test t specified in an SSO or TCO, the algorithm in Figure 8 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 8).

5.5 Selecting the Schedule of Test Execution

After the test-to-snapshot mapping P_M has been generated, the Optimizer selects the test execution schedule as well as the minimum number of hosts needed for running these tests. This stage 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 allocation of testing hosts and running of the tests on the resource provider—only after the selected testing plan is submitted to the Orchestrator.

The complete details of the algorithm used in this stage are given in the online technical report [4]. This greedy 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 three ways as discussed next.

The first way 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. Recall that Amulet strives to generate a robust testing plan, i.e., a plan whose chances of performing worse than estimated is low [2]. If there is no model to estimate how the concurrent execution of a set of tests will perform, then the Optimizer will take the low-risk route of avoiding such executions.

While selecting the test execution schedule, the algorithm also addresses the important issue of *schedule sustainability* which ensures that the plan generated for one window can be run continuously for any number of windows that come one after the other. Using notation from Table 3, let *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:

$$P_W + start(h) > end(h) \quad (2)$$

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 5.3 to select the window size P_W ensures that no test run will span more than two consecutive windows.

The second way to schedule test t_{ij} 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 and sustainability, the algorithm also checks whether t_{ij} can be started over s_i on h_k before 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 testing host that is already allocated in the plan, then the third way is to schedule t_{ij} on a new testing host added to the plan. This step will use the resource provider's pricing model to ensure that the addition of a new testing host will not overshoot the cost budget specified in the Angel program. Note that the allocation of a new testing host to run t_{ij} will not violate schedule sustainability because *ExecTime*(t_{ij}) $\leq P_W$ (from Section 5.3).

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. The complete details of two algorithms that we have developed for non-cost optimization are given in the online technical report [4]. 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 has this optimization objective.) It emerges

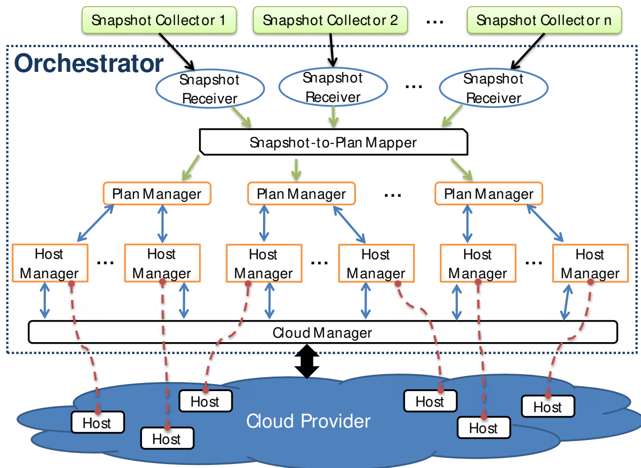


Figure 9: Amulet's Orchestrator

from Equation 1 that the way to achieve lower values of τ_{rpo} is by reducing the snapshot interval $P_I = Time(s_i) - Time(s_{i-1})$. ($ExecTime(t)$ 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 adding 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.

6. ORCHESTRATOR

Recall from Section 1 and Figure 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 9 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 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 9). 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 overshot 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 administrator.

7. EXPERIMENTAL EVALUATION

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 [1] as the infrastructure-as-a-service cloud provider. Section 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 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) [8]. For each layer of the stack, we chose a representative test from Table 1: myisamchk for MySQL database integrity checking, and fsck and xfs_check for file-system integrity checking. Execution-time estimation models for these tests were trained and validated on the Amazon cloud (see Section 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 [8]. 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 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 resource usage is charged in a pay-as-you-go fashion on the Amazon cloud [1]. Table 5 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.

7.2 End-to-end Processing of Angel Programs

7.2.1 Case 1: Maintaining a Tested Recovery Point

Angel program: We first submit to Amulet the Angel program corresponding to Example 1 from Table 2. 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 6 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

Resource Used	Pay-as-you-go Pricing Method
Testing hosts	Hosts of the Small type (see Figure 4(a)) 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

Table 5: Pricing model used in our evaluation

- Since no CO is specified, P_R takes the default value which in this case is the cost of one Small host (see Section 4.1)

Orchestration Timeline: Figure 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).

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 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 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 that every other snapshot submitted is tested. The boxes with notation S_jT_M in Figure 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.

7.2.2 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` $\leq$ 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 10 shows the actual execution timeline of the above plan P. Note that P_I = 10 minutes causes

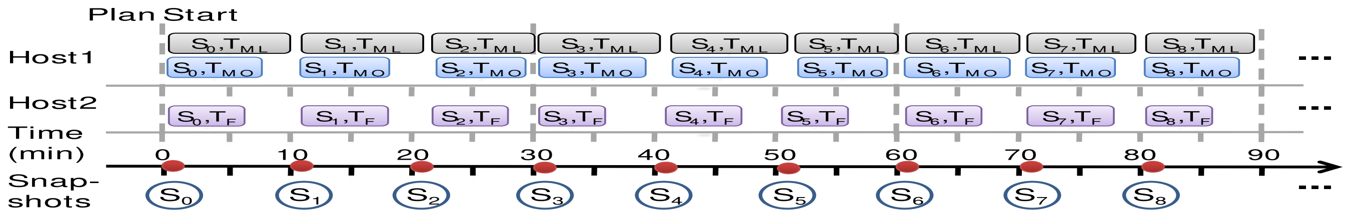


Figure 10: Actual execution timeline on the Amazon Cloud for Case 2 in our evaluation

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.

7.2.3 *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 7.2.2. 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 11 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 6 get satisfied at around 52 minutes in the timeline. *Host*$_1$ has overshot 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.

7.2.4 *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 12 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 12). 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.

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:

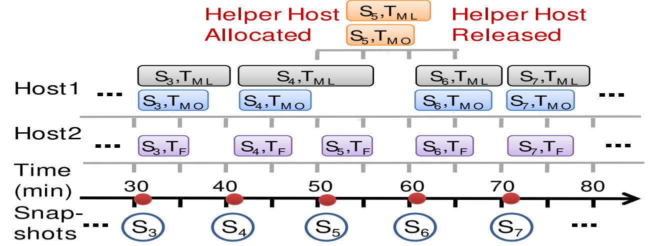


Figure 11: Actual execution timeline for Case 3

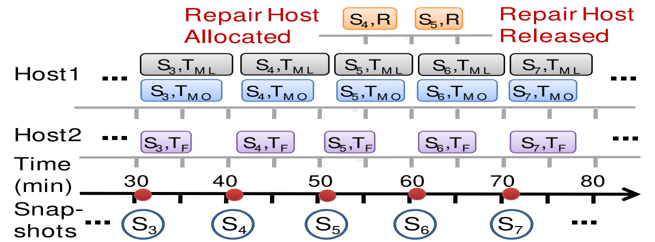


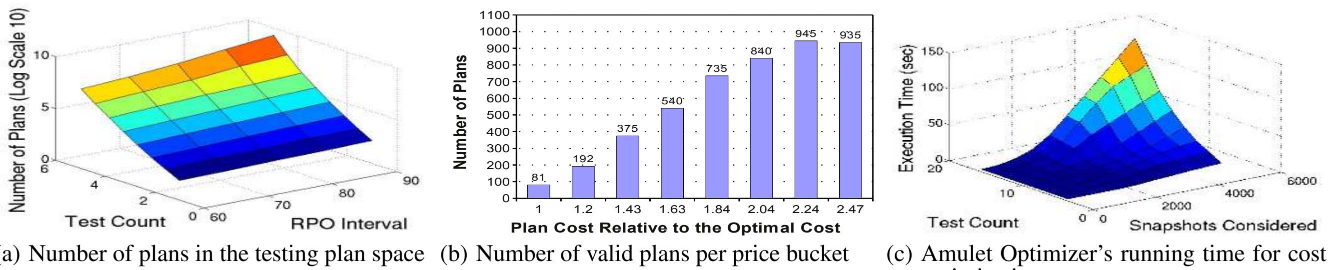
Figure 12: Actual execution timeline for Case 4

- 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 7.2.1, 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 13(a) shows the total number of plans obtained by running EOpt. Note the $\log_{10}$ scale in Figure 13(a) 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). Figure 13(b) 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.

Scalability: Given the large plan space per volume and the speed at which it grows (Figure 13(a)), we measured the time that each optimizer takes to find a testing plan per volume. We fixed τ_{rpo}



(a) Number of plans in the testing plan space (b) Number of valid plans per price bucket (c) Amulet Optimizer's running time for cost optimization

Figure 13: Characteristics of the testing plan space and the performance of Amulet's Optimizer

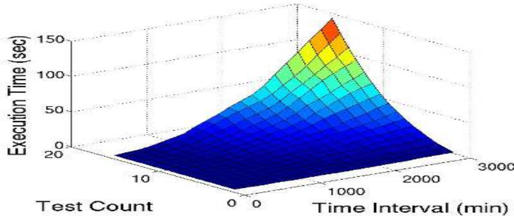


Figure 14: Amulet Optimizer's running time for non-cost optimization

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 13(c) 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 14 shows how Amulet's Optimizer continues to remain efficient even under non-cost optimization objectives. The binary-search algorithm from Section 5.6 was used. 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 14 is similar to what we observed for the cost optimization objective in Figure 13(c).

8. CONCLUSIONS AND FUTURE WORK

Hardware errors, software bugs, and mistakes by human administrators 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 administrators 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 administrators 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. We intend to build on the current framework in future, and explore several directions including adaptive techniques for plan optimization.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd for helping us improve the paper in many ways.

10. REFERENCES

[1] *Amazon Web Services*. aws.amazon.com.
[2] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *SIGMOD*, 2005.
[3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST*, 2008.
[4] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Warding off the Dangers of Data Corruption with Amulet. Technical report, 2011.
[5] How to Use Chkdsk. http://bit.ly/vyyZL.
[6] *Data corruption in CouchDB*. couchdb.apache.org/notice/1.0.1.html.
[7] Comparison of INSPECT and db2dart. http://bit.ly/hPYpq5.
[8] *Running MySQL on Amazon EC2 with EBS*. http://bit.ly/b7SWwg.
[9] Eseutil tutorial. http://bit.ly/f0iPY1.
[10] *Checking and Repairing Unix File system with fsck*. http://adminschoice.com/repairing-unix-file-system-fsck.
[11] Storage software update causes 0.02% of Gmail users to lose their emails. http://bit.ly/e5Xn18.
[12] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Susseau. SQCK: A Declarative File System Checker. In *OSDI*, 2008.
[13] H. Guo, D. Jones, J. L. Beckmann, and P. Seshadri. Declarative Database Management in SQLServer. *PVLDB*, 2(2), 2009.
[14] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST*, 2008.
[15] *Data corruption at Ma.gnolia.com*. en.wikipedia.org/wiki/Gnolia.
[16] MySQL myisamchk command. http://bit.ly/gGFRUP.
[17] *Corrupted backups*. http://bit.ly/iffTC8.
[18] Oracle HARD Initiative. http://bit.ly/eTxiqa.
[19] Detecting and Recovering from Database Corruption. http://bit.ly/eKXMAi.
[20] Parchive. http://parchive.sourceforge.net/.
[21] *Data corruption in Amazon S3*. http://bit.ly/foWIul.
[22] B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.
[23] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS*, 2004.
[24] Microsoft SQL Server checkDB command. http://bit.ly/gawaHF.
[25] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *ICDE*, 2010.
[26] *Tripwire Tutorial*. http://bit.ly/csCvmr.
[27] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster Recovery as a Cloud Service: Economic Benefits and Deployment Challenges. In *HotCloud*, 2010.
[28] Fixing XFS. http://bit.ly/1XRuDy.
[29] *Solaris ZFS Administration Guide*. http://bit.ly/cw1EZY.
[30] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *FAST*, 2010.