Proactive Detection and Repair of Data Corruption: Towards a Hassle-free Declarative Approach with Amulet

Nedyalko Borisov
Duke University
nedyalko@cs.duke.edu

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

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, apart from being tedious and error-prone, provides no understanding of the potential system unavailability and data loss if a corruption were to occur. We have developed the *Amulet* system that can verify the correctness of stored data proactively and continuously. This demonstration focuses on the uses of Amulet and its technical innovations: (i) a declarative language for administrators to specify their objectives regarding the detection and repair of data corruption; (ii) optimization and execution algorithms to meet the administrator's objectives robustly and with least cost using pay-as-you-go cloud resources; and (iii) timely notification when corruption is detected, allowing proactive repair of corruption before it impacts users and applications.

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 [2, 3, 5, 7, 11]. 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* [5]. Bugs in storage software [3], OS device drivers, and higher-level layers like load balancers [9] and database software [2] 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 [2]. 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) [3].
- 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 [10].
- 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 [9].
- Increasing inter-dependencies and complexity in systems. Corruption of four files in an Oracle database at JPMorgan Chase recently caused a severe outage that left customers stranded, and blocked about $132 Million in financial transactions [4].

Data corruption can have severe 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 [6].

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 [8]. 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 [5] as well as plenty of anecdotal evidence show that problems due to corruption happen, and more frequently than expected [10]. A particularly dangerous scenario that the authors as well as others (e.g., [4, 6, 7]) have come across involves the propagation of corruption from the production system to critical backups; increasing the chances of data loss if a failure occurs in the production system.

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. A detailed list of tests can be found in [1]. 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 [11].
- 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, then 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.

*Supported by NSF grants 0644106 and 0964560

Proceedings of the VLDB Endowment, Vol. 4, No. 12
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

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

- 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 1 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.

This demonstration will present the *Amulet* system (Figure 1) that we have developed to detect and repair data corruption proactively and continuously [1]. Section 2 gives an overview of the uses of Amulet and its technical innovations. Section 3 presents our demonstration plan. We refer the reader to [1] for a detailed description and experimental evaluation of Amulet.

2. AMULET

A typical *database software stack* on a production host is shown in Figure 1. Different levels of the software stack maintain different sources of data. All these data sources have to be kept corruption-free in order 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. For example, MySQL's *myisamchk* 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 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.

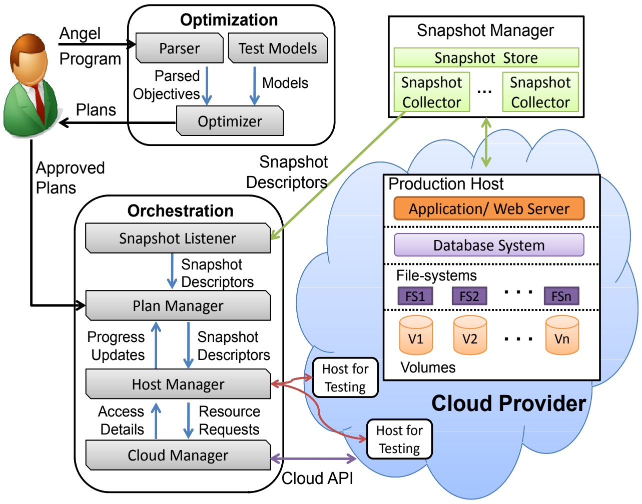

Figure 1: Overview of Amulet's optimization and orchestration.

The first challenge that Amulet faces is how to run a test automatically. Most of the popular tests cannot be run concurrently with the regular workload on the production host 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 demonstration, we focus on volume-level snapshots because they capture the data needed for any test in the software stack.[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 to the production host.

2.1 Amulet's Angel Declarative Language

Easy and intuitive declaration of objectives like those in Table 1 poses a language design problem. Amulet's declarative language, called *Angel*, is designed to express such objectives. Example Angel programs can be found in [1]. The cases in Table 1 reveal the important features that are built into the language, discussed next.

Tests, Data, and Resources: Angel contains a `Test` statement that 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 `Data` statement defines 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. Angel's `Host` statement defines a host type (from a set of supported types) for a test t so that Amulet will always run t on testing hosts of that type.

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

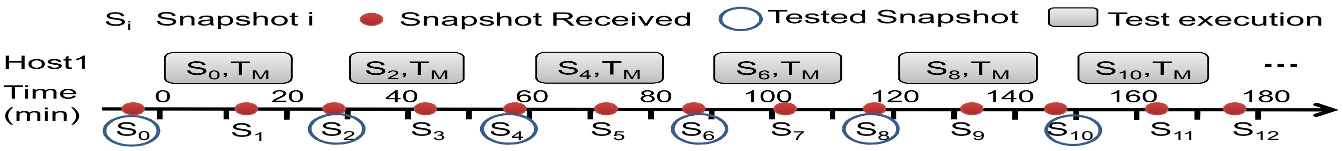

Figure 2: Actual execution timeline, in minutes, of a testing plan in Amulet for Example 1 from Table 1. Each box denotes a run of the myisamchk (T_M) test on the respective snapshot S_i. The horizontal width of each box corresponds to the test execution time.

Tested Recovery Points: 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. Angel's `Recovery Point Objective (RPO)` statement defines a recovery point objective with its respective time interval.

Other Objectives: Angel contains a `Test Count Objective (TCO)` statement that defines, for a test t, the minimum or maximum number of test runs per specific time window. The `Cost Objective (CO)` statement can be used to specify a budget for provisioning pay-as-you-go cloud resources to run tests. The `Snapshot Interval Objective (SIO)` statement specifies constraints on the rate at which snapshots are tested.

Optimization Objective: The `Optimize Objective (OO)` statement allows administrators to declare any one objective as subject to maximization or minimization during test execution.

2.2 Amulet's Optimizer

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 on the Amazon cloud for a testing plan P for an Angel program corresponding to Example 1 from Table 1. 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. 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 that Amulet addresses.

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. 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.

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.

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 have developed 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. The complete details of the algorithm used to generate testing plans are given in [1].

2.3 Amulet's Orchestrator

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:

- *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.
- *Repairs:* It is impossible to predict when a corruption will be detected and a repair action needs to be taken.
- *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. 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.

3. DEMONSTRATION PLAN

Amulet has been deployed and evaluated as a service running on the Amazon Elastic Compute Cloud [1]. We will use this deployment (illustrated in Figure 1) as the setting for the demon-

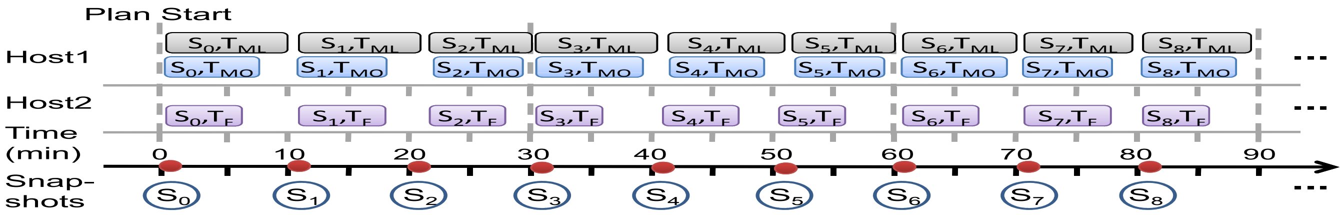

Figure 3: Actual execution timeline on the Amazon Cloud for the testing plan from the base scenario in our demonstration.

stration.[2] The demonstration itself will present three aspects of Amulet: declarative use, regular optimized execution of testing plans, and dealing with unpredictable events during plan execution.

3.1 Ease of (Declarative) Use

This part of the demonstration will focus on the Angel language by showing how the use cases from Table 1 can be expressed in the Angel declarative language. We will start with the Angel program P for Example 1. This program will illustrate statements for Test, Data, and Resources, in addition to the important `Recovery Point Objective (RPO)` statement. Amulet's Visualizer will be used to show the testing plan chosen for P by the Optimizer.

We will then invite viewer participation to either show more use cases from Table 1 or to complicate these use cases in two directions: (i) adding more tests to the program, and (ii) including more objectives. This process will enable viewers to appreciate the ease of use and intricacies of Amulet. Viewers will also gain an understanding of the Optimizer's algorithm used to satisfy objectives and the guarantees provided by the chosen testing plans. For example, viewers can see how the chosen testing plan changes as the budget for provisioning pay-as-you-go cloud resources is varied through a `Cost Objective (CO)`. Users with a deep interest in Amulet can also browse the cost models for tests using the Visualizer.

3.2 Automatically-Optimized Execution

The demonstration session will continue by showing viewers how the optimized testing plans are executed by the Orchestrator. Our base scenario—whose actual execution timeline is shown in Figure 3—will consists of three tests for a MySQL database running on the Amazon cloud: two myisamchk-medium tests respectively on a lineitem table (T_{ML}) and an orders table (T_{MO}), and an fsck metadata test (T_F). The desired objectives will be to test at least one snapshot every 10 minutes, and to ensure that a tested recovery point exists within the most recent 30 minutes. Amulet's Visualizer supports visualization of live testing plan execution as well as that of monitoring information collected by the Orchestrator from past plan executions.

3.3 Dealing with Unpredictable Events

Finally, based on viewer interest, we will present how Amulet's Orchestrator handles the following two types of unpredictable events:

1. *Straggler hosts:* During the execution of the Orchestrator's base scenario from Figure 3, we will inject a problem on $Host_1$ that causes the T_{ML} test on the host to run slower than expected. Figure 4 shows the actual execution timeline. Because test T_{ML} on snapshot S_4 overshoots its estimated time and poses the danger of violating plan guarantees, the Orchestrator will mark $Host_1$ as a straggler, and allocate a helper host (around time 52 in Figure 4) to take over some of the workload on $Host_1$. Around time 63, $Host_1$ is no longer a straggler host; thus the helper host is released. Intuitively, the helper host was allocated dynamically to help the plan tide over a transient problem.

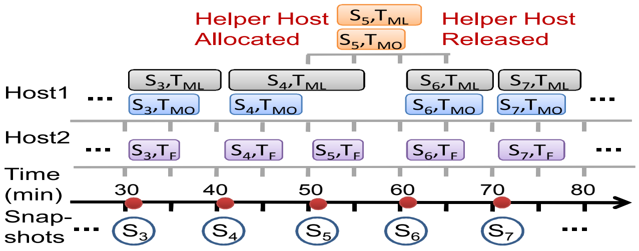

Figure 4: Actual execution timeline with a straggler host.

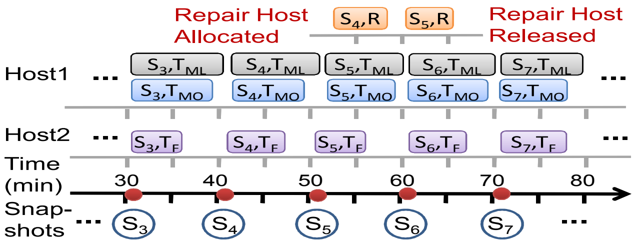

Figure 5: Actual execution timeline with a need for repair.

2. *Repairs:* During the execution of the Orchestrator's base scenario, we will cause a data corruption in the production host (lineitem table). In Figure 5, this corruption manifests itself in two snapshots: S_4 and S_5. The corruption is detected by T_{ML} when run on S_4 (S_4, T_{ML} in Figure 5), and reported to the Orchestrator. The Orchestrator requests a repair host that will execute the repair action specified by the Angel program. The repaired snapshot can be loaded or the repair action applied directly on the production host as desired by the administrator.

4. REFERENCES

[1] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Warding off the Dangers of Data Corruption with Amulet. In *SIGMOD*, pages 277–288, 2011. `http://www.cs.duke.edu/~nedyalko/amulet.html`.

[2] *Data corruption in CouchDB*. couchdb.apache.org/notice/1.0.1.html.

[3] Storage software update causes 0.02% of Gmail users to lose their emails. http://bit.ly/e5Xn18.

[4] Outage at JPMorgan Chase due to Oracle Database Corruption. dbms2.com/2010/09/17/jp-morgan-chase-oracle-database-outage.

[5] 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*, pages 127–141, 2008.

[6] *Data corruption at Ma.gnolia.com*. en.wikipedia.org/wiki/Gnolia.

[7] *Corrupted backups*. http://bit.ly/iffTC8.

[8] Oracle HARD Initiative. http://bit.ly/eTxiqa.

[9] *Data corruption in Amazon S3*. http://bit.ly/foWIul.

[10] B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, pages 193–204, 2009.

[11] 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*, pages 509–520, 2010.

[2] In case internet connectivity is unavailable at the demonstration site, we will fall back on the Amulet Visualizer's replay mode that can visualize past execution of testing plans.