Understanding Real World Data Corruptions in Cloud Systems

Peipei Wang, Daniel J. Dean, Xiaohui Gu
Department of Computer Science
North Carolina State University
Raleigh, North Carolina
{pwang7,djdean2}@ncsu.edu, gu@csc.ncsu.edu

Abstract—**Big data processing is one of the killer applications for cloud systems. MapReduce systems such as Hadoop are the most popular big data processing platforms used in the cloud system. Data corruption is one of the most critical problems in cloud data processing, which not only has serious impact on the integrity of individual application results but also affects the performance and availability of the whole data processing system. In this paper, we present a comprehensive study on 138 real world data corruption incidents reported in Hadoop bug repositories. We characterize those data corruption problems in four aspects: 1) what impact can data corruption have on the application and system? 2) how is data corruption detected? 3) what are the causes of the data corruption? and 4) what problems can occur while attempting to handle data corruption?**

Our study has made the following findings: 1) the impact of data corruption is not limited to data integrity; 2) existing data corruption detection schemes are quite insufficient: only 25% of data corruption problems are correctly reported, 42% are silent data corruption without any error message, and 21% receive imprecise error report. We also found the detection system raised 12% false alarms; 3) there are various causes of data corruption such as improper runtime checking, race conditions, inconsistent block states, improper network failure handling, and improper node crash handling; and 4) existing data corruption handling mechanisms (i.e., data replication, replica deletion, simple re-execution) make frequent mistakes including replicating corrupted data blocks, deleting uncorrupted data blocks, or causing undesirable resource hogging.

I. INTRODUCTION

Cloud computing [1], [2] has become increasingly popular recently. Massive data processing systems such as Hadoop [3] are most commonly used cloud applications. However, cloud data corruption [4] has become a big concern for the user, which can not only cause data loss [5] but also system crashes [6] and reduced availability [7]. Data corruption issues often incur severe financial penalty ranging from millions in fines [8] to company collapse [9].

Data corruption can be caused by both hardware faults (e.g, memory errors, disk bit rot) and software faults (e.g., software bugs [4], [10], [11]). As the capacity of hard disks and memory grows, the likelihood of hardware failures also increases. For example, Facebook [5] temporarily lost over 10% of photos because of a hard drive failure. Similarly, as cloud systems scale out, data corruptions caused by software bugs also become prevalent. For example, Amazon Simple Storage Service (S3) [1] encountered a data corruption problem caused by a load balancer bug [12]. Although previous studies [13]–[15] have shown that disk errors (e.g., latent sector errors [16]) and DRAM errors in large-scale production systems are large enough to require attention, little research has been done to understand software-induced data corruption problems.

In this paper, we present a comprehensive study on the characteristics of the real world data corruption problems caused by software bugs in cloud systems. We examined 138 data corruption incidents reported in the bug repositories of four Hadoop projects (i.e., Hadoop-common, HDFS, MapReduce, YARN [17]). Although Hadoop provides fault tolerance, our study has shown that data corruptions still seriously affect the integrity, performance, and availability of the Hadoop systems. Particularly, we characterize the 138 data corruption incidents in our study based on the following four aspects: 1) what impact can data corruption have on the application and system? 2) how is data corruption detected? 3) what is the causes of the data corruption? and 4) what problems can occur while attempting to handle data corruption?

Our study has made the following findings:

- **Data corruption impact:** Data corruption impact is not limited to data integrity. If corrupted blocks are not handled correctly, the data they store could be completely lost. Corrupted metadata prevents data from being accessed, which might cause wrong data deletion in some cases. Essential HDFS system files which keep track of HDFS file system information (e.g., permissions) can cause other components of the system to crash when corrupted. Furthermore, if these system files are not recoverable, all data stored in HDFS can be lost. Data corruption can also cause job failures, infinite loops, and failed client requests for Hadoop services.
- **Data corruption detection:** Although Hadoop provides a data corruption detection mechanism, it is far from sufficient to detect all data corruptions. Our study shows that Hadoop fails to detect many data corruption problems caused by software bugs. Moreover, the existing Hadoop file system integrity checking scheme (i.e., `fsck`[1]) can generate false alarms which leads to unnecessary debugging cost. Hadoop data corruption detection mechanisms can also generate imprecise messages. For example, genstamp[2] mismatches are always reported as block corruption. In cases of unhandled exceptions caused by data corruption, exception

[1]fsck is a Hadoop command for checking filesystem and reporting problems with files.

[2]Genstamp is short for GenerationStamp, an identity number indicating block version. It is assigned to each created block and gets increased each time the block is updated.

names and exception messages are often unclear, providing little useful information about the underlying data corruption. Those unclear error messages make it difficult for developers to correctly identify and fix the problem.

- **Data corruption causes:** The cause of data corruption can be from a variety of factors including both external and internal to HDFS. When processing uploaded files, external factors such as disk failure and bit-flip in memory generate exceptions which are difficult to handle properly. Internal factors also exist and mainly result from either Hadoop's inability to maintain data consistency in the face of node crashes or from Hadoop's inability to synchronize data information in the face of concurrent operations on the same blocks.
- **Data corruption handling:** Hadoop provides data corruption handling mechanisms. However, problems can occur during the handling of corrupted data. For example, a corrupted block can be mistakenly replicated, which causes Hadoop to delete it because it is corrupted, which triggers replication again. Worse yet, this replication and deletion process can continue infinitely.

The remainder of the paper is organized as follows: Section II describes our data corruption problem sources, the statistics of different data corruption types, and how we categorize the data corruption problems. Section III describes different impact the data corruption problems have on Hadoop. Section IV describes how Hadoop detects data corruptions along with the problems that can cause Hadoop to miss data corruptions. Section V discusses different causes of data corruptions. Section VI describes how Hadoop handles detected data corruptions along with the problems that can occur during data corruption handling. Section VII discusses the work related to our study. Finally, Section VIII concludes the paper.

II. Preliminaries

A. Hadoop Background

Hadoop has two broad classes of versions, namely, 1.x and 2.x. Hadoop 1.x consists of Hadoop common, HDFS, and MapReduce. Hadoop 2.x introduces HDFS federation and a resource management framework, YARN. Table I summarizes our notations. We now describe them as follows.

Hadoop uses the Hadoop Distributed File System (HDFS) to store files among DataNodes (DNs) under the management of the NameNode (NN) which is the master node where Hadoop maintains the file system namespace of HDFS. DN is the slave node which is used for storing files and responds to commands from NN. The SecondaryNameNode (2NN) periodically creates checkpoints of the file system maintained in NN.

Yet Another Resource Negotiator (YARN) is responsible for resource management when executing MapReduce programs in Hadoop. The ResourceManager (RM) is a scheduler to optimize cluster utilization. For each machine in a Hadoop cluster, there is a NodeManager (NM) which monitors resource usage and reports it to RM. For each MapReduce job, there is an ApplicationManager (AM) which requests resources from RM and reports job status and progress to RM.

TABLE I. NOTATIONS.

Notation	Meaning
HDFS	Hadoop Distributed File System
YARN	Yet Another Resource Negotiator
DN	DataNode
NN	NameNode
2NN	SecondaryNameNode
RM	ResourceManager
NM	NodeManager
AM	ApplicationManager

TABLE II. STATISTICS OF DATA CORRUPTION INCIDENTS.

System name	System file corruption	Metadata corruption	Block corruption	Misreported corruption
Hadoop 1.x	15	11	46	4
Hadoop 2.x(YARN)	1	0	7	0
HDFS 1.x	17	7	23	7
HDFS 2.x	8	0	22	10

There are three different types of data stored in HDFS: 1) *system files* that include files needed by HDFS (e.g., `edits.log`, `fsimage`, `VERSION`), job history logs, and task description files, and are stored on the NN; 2) *data blocks* that are chunks of user data and replicated among several DNs; and 3) *block metadata* that are created along with the data blocks on the same DN and include block information such as block's checksum and genstamp.

B. Data Corruption Sample Collection

We collected samples of real world data corruption incidents from the Hadoop bug repository [18]. We selected the data corruption samples using these criteria: 1) project: HADOOP, HDFS, MAPREDUCE, YARN; 2) issue type: Bug; 3) status: RESOLVED, CLOSED, PATCH AVAILABLE; and 4) keyword: corruption. We then manually examined each incident to ensure it is indeed related to data corruption and is resolved (e.g., patch provided). We found 138 distinct data corruption samples for our study, shown by Table II. We classify data corruption problems into different groups based on what types of data are affected by the problem.

To emphasize data corruption incidents in HDFS, we separate HDFS from Hadoop 1.x and from Hadoop 2.x. Some selected incidents could affect both Hadoop 1.x and Hadoop 2.x. For the 12 incidents without specific version specified, we use the fix version to infer the proper version. For the five incidents specifying neither an affected version nor a fix version, we identify the Hadoop version by searching for version-related information in the bug descriptions, comments, uploaded logs, and attached patches. While some incidents could affect more than one type of data, some others do not actually affect any type of data either because they happen in memory or because they are misreported data corruption.

C. Characteristic Categories

We categorize the data corruption problems in four aspects: *impact*, *detection*, *causes*, and *handling*. Each aspect is subdivided into several types and the percentage of the data corruption samples in each type is also reported.

Data corruption impact: We measure the data corruption impact in three dimensions: 1) *integrity* which means data corruption has occurred to data stored in HDFS; 2) *availability* which means data corruption causes Hadoop node crashes, restarting failures, Hadoop service failures, or MapReduce job failures; and 3) *performance* which means MapReduce jobs or Hadoop services either take a longer time to finish or is unable to serve more client requests. Several data corruption problems impact more than one dimension.

Data corruption detection: For data corruption detection, we divide bug detection into four types: 1) *correct data corruption detection* which means once data corruption occurs, it can be detected immediately or automatically by Hadoop, and the reported messages about data corruption are accurate enough to let users know what data is corrupted; 2) *imprecise data corruption detection* which means although the data corruption is detected immediately or automatically by Hadoop, the generated message does not indicate what data are corrupted; 3) *silent data corruption* which means the data corruption is not detected; and 4) *misreported data corruption* which means one or more blocks are reported as corrupted while actually these blocks are intact and uncorrupted.

Data corruption causes: For data corruption causes, we use the following categories: 1) *improper runtime checking* which means data corruptions of this type can be fixed by checking inputs to Hadoop or by checking resources usages of Hadoop; 2) *race condition* which means multiple threads are changing the same variable value or are writing to the same file; 3) *inconsistent state* which means NN keeps block information which is not consistent with the blocks stored in the DNs, or the DNs have inconsistent block metadata on the same block; 4) *improper network failure handling* which means the data corruption can occur when data or messages are not transmitted successfully, or only part of the data or message is transmitted; 5) *improper node crash handling* which means the data corruption can be caused by the NN or the DN process termination; 6) *incorrect name/value* which means the data corruption is caused by using incorrect variable name or value; 7) *lib/command errors* which means libraries or commands errors can cause data corruption; 8) *compression-related errors* which means the data corruption occurs during the compression process; and 9) *incorrect data movement* which means moving invalid or incomplete data into data directory.

These categories are not disjoint. For example, data corruption could be caused by a race condition which also leads to an inconsistent block state. Since we are only interested in *real* data corruption, we exclude any misreported data corruption incidents. We also exclude those incidents with unclear or undocumented data corruption causes. Specifically, 17 data corruption incidents are of misreported data corruption and there are 40 incidents of which we cannot find the causes.

Data corruption handling: When examining current data corruption handling schemes, we found problems could occur during 1) *data replication* where Hadoop cannot correctly identify the blocks which need replication; 2) *replica deletion* where Hadoop cannot correctly delete the corrupted replica; and 3) *simple re-execution* where Hadoop simply rerun the job or command to recover from failures. We found 32 incidents that describe the problems during data corruption handling.

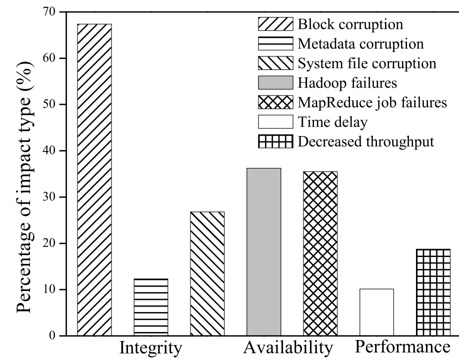

Fig. 1. Different impact of data corruption.

III. Data Corruption Impact

In this section we discuss the impact that data corruption has on Hadoop. Specifically, data corruption has the potential to impact Hadoop in three different ways. First, the integrity of the data may be affected, causing clients to be unable to successfully retrieve data, namely, block corruption, metadata corruption, and system file corruption. Second, the availability of Hadoop may be affected, namely, Hadoop failures, and MapReduce job failures. Third, the performance of the system can be affected, namely, time delay, and decreased throughput. Several data corruption problems impact on more than one type. The detailed percentage distribution of the three types is shown in Figure 1. The dominant data corruption impact is on integrity, and over 65% bug incidents are related to block corruption. With the unexpected high percentage (i.e., over 10%, 25% respectively), metadata corruption and system file corruption are the corruptions people should pay more attention to.

A. Integrity

There are several ways in which data corruption can affect the integrity of Hadoop data. Specifically, the result of a MapReduce job, data already stored in HDFS, block metadata files on the DN, or HDFS system files on the NN can all be corrupted. Furthermore, data corruption can occur in several different files at the same time.

Block corruption: Data blocks are one type of data vulnerable to corruption. Data block corruption can affect data that is already stored or will be stored in HDFS (e.g., MapReduce job results). For example, HADOOP-1014[3] explains how map task output data is corrupted, causing missing key/value pairs.

Metadata corruption: Block metadata is another frequently mentioned type of data in the studied data corruption incidents. In general, the DN creates a metadata file for each block stored on it. The metadata file records a lot of block information (e.g.,

[3] 1014 denotes the bug number in the bug repository.

identifier, size, genstamp, length, and checksum value). Since a block usually has three replicas stored on different DNs, updating data in HDFS could possibly put the replicas into an inconsistent state, causing misreported corruption messages or misleading corruption messages to clients and developers. For example, in HDFS-1232, metadata corruption occurs because of the DN crashes between storing a data block on it and writing the block checksum to the block metadata file.

System file corruption: HDFS system files are files on the NN containing HDFS file system information. `edits.log`, `fsimage` and `VERSION` are the most commonly seen corrupted system files in our case study. These files can be corrupted as a result of unhandled software bugs which cause failures (e.g., improper node crash handling, HDFS synchronization failures). These files need to be loaded into the memory of the NN when starting HDFS. Once corrupted, they will make HDFS unable to start. If not recovered, all files stored in HDFS will be inaccessible. In the worst case, all data can be lost. For example, HDFS-1594 shows the corruption of `edits.log` that is caused by the disk being full, which makes it impossible to start the NN.

B. Availability

The availability of Hadoop can be compromised by either data corruption-based Hadoop failures or data corruption-based MapReduce job failures. Applications built on top of HDFS also suffer from data corruption-based availability issues.

Hadoop failures: Hadoop node crashes, node recovery failures, and node restarting are commonly reported issues in the surveyed incidents. One closely related factor for those failures is corruption of the `fsimage` and `edit.log` files. These two files are necessary for starting the NN and when corrupted, can irrecoverably crash the NN. For example, most users store multiple copies of the `fsimage` file to prevent potential data loss. HDFS-3277 shows that when the first loaded `fsimage` is corrupted, the NN is unable to load a different correct `fsimage`.

The other two types of corruption, namely block and metadata, can cause part of Hadoop services to be unavailable to users. For every access of data, HDFS starts data corruption detection before transmitting data to client. If HDFS detects a data block is corrupted, it will refuse to transmit the data. Specifically, HDFS does checksum checking on a requested block by matching checksum values kept in metadata with the one calculated from the block. As we have previously mentioned, metadata keeps track of block checksum. Either corrupted block or corrupted block results in checksum mismatches, thereafter causes failure of Hadoop services which need to access data block in HDFS. In HDFS-679, the block receiver throws exception about a mismatched checksum because the checksum for the last packet is mistakenly set to zero.

MapReduce job failures: Hadoop prevents clients from reading blocks which are marked as corrupted. As a consequence of this, jobs or tasks can fail. For example, HADOOP-86 is a bug in which corrupted map outputs cause reducers to get stuck fetching data forever. Another case is a failure to read a non-existent block. MAPREDUCE-3931 shows such a failure involving GridMix [19], which is a benchmark for Hadoop MapReduce tasks. In Hadoop 2.x, submitted jobs are divided into a couple of tasks, and every task is executed in an independent resource container. Before launching resource container, YARN needs to localize the file/library which contains the source code of the task, and then downloads the file/library from HDFS. However, a corrupted time stamp causes Hadoop to attempt to download a non-existent block, failing the job.

When data corruption causes HDFS to return an unexpected result to an application built on top of it (e.g.,HBASE [20], HIVE [21], and PIG [22]), availability is impacted. For example, HDFS-3222 is one case preventing HBASE from completely replaying unsaved updates by providing an incorrect length of related file. HBASE stores data updates in memory and saves them to disk in batches. To prevent data loss, HBASE stores unsaved updates in a file which is also stored in HDFS so that it can recover data if one of its server fails. Because HDFS provides a wrong file length value, HBASE ends up only partially reading the data updates and thus could not recover all data completely.

C. Performance

The performance of Hadoop can be degraded in two ways, a time delay from the prolonged job execution or a throughput decrease.

Time delay: Once a job or task fails, Hadoop needs to re-execute it, causing the job or task to take a longer time to finish than usual. A time delay can also occur when a task tries to locally access a block replica which in fact has already been deleted from its local directory. In this case, the overhead is transferring data from a remote DN to the DN where the task is performed. As mentioned in Section II, HDFS file system information and block information are stored separately. In order to collect block information on the NN, every DN sends block reports to the NN which contains all block information on the DN itself. Therefore, the deleted block replica is not known to the NN before a block report is sent from the DN. Because of this, a new task with the same input will request the block again since the first time it is requesting a deleted block replica. This problem could cause substantial unnecessary overhead in iterative and incremental computations. For example, YARN-897 discusses a resource starvation problem. Because YARN uses a counter named "UsedCapacity" as the criteria to assign resources to jobs, releasing resources without updating this counter value can cause a job to have a less chance of getting resources again. Waiting a longer time for resources will elongate the total job execution time.

Client requests can also be delayed because of a temporarily unresponsive NN or DN. This can occur when the NN runs into an infinite loop or a loop which is only exited when the job times out. Unresponsiveness can occur as the result of a large number of block deletions, as seen with bug HDFS-1393. In this case, the NN is unresponsive for as long as 20 seconds because no other operations can occur while deleting blocks.

Decreased throughput: Decreased throughput typically occurs when Hadoop does intensive unnecessary work, such as retrying or repeating a failing command. For example, HADOOP-1911 shows how using the `dfs -cat` command to read a corrupted file causes an exception. Hadoop attempts

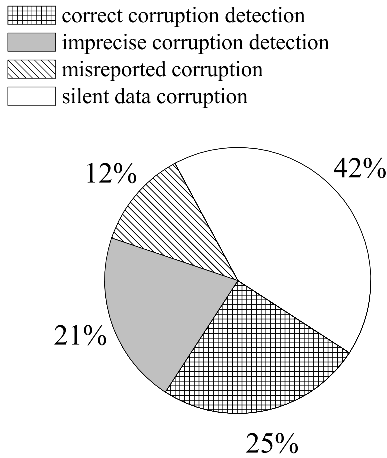

Fig. 2. Different types of data corruption detection.

to handle this problem by repeating the command, causing the process to repeat infinitely. Another example is HADOOP-4692. The only replica existing in HDFS is corrupted. Since the system is under-replicated, the NN replicates the replica. It then finds that the newly created replica is corrupt and should be deleted. This replication and deletion process then repeats infinitely.

IV. Data Corruption Detection

In order to detect data corruption, HDFS currently provides two main methods, checksum verification and block scanning. Checksum verification occurs in several steps. First, a block metadata file is created and stored in HDFS. A checksum value is then calculated for this block and written to the metadata file. Whenever the block is accessed, HDFS calculates that block's checksum value again and compares that value with the checksum value stored in the metadata file. Whenever a block is updated, the checksum value in the metadata file is also updated. If data corruption affects stored data which has some operations performed on it (e.g, write), checksum verification will find it immediately. The other method is periodically running a block scanner. The block scanner provides a good way to automatically find existing data corruption without operations on it.

However, from our study shown in Figure 2, we can see that only 25% of data corruption incidents are detected and reported correctly. 42% of data corruption incidents are silent data corruption. 12% of incidents are reported as false alarms. 21% of incidents have imprecise report messages. We now discuss the three types of errors during data corruption detection in detail.

A. Silent Data Corruption

It is difficult to detect data corruption in HDFS system files. HDFS assumes system files are correct and does not provide any methods to check them. Restarting the NN is an indirect way to check the existence of data corruption in system files. Because HDFS system files are stored on a single NN in Hadoop 1.x, any HDFS system file (e.g., `fsimage`, `edits.log`) corruption will cause the NN to fail to start. HDFS-2305 discusses failures in synchronizing `fsimage` in an HDFS system with multiple 2NNs. Instead of getting an up-to-date file, they get an outdated file. This causes the operations logged in `edits.log` to fail when applied later. HDFS-3626 shows an example of a corrupted `edits.log` as the result of creating file with invalid path. This corruption is unknown to the NN until it is restarted. When restarted, all the operations logged in this file are performed again, triggering an exception. Hadoop 2.x is able to better handle system file corruption because the introduction of multiple NNs allows HDFS system file corruption to be detected and corrected. This leads to a reduction in the system file corruption cases as shown in Table II.

Another reason for silent data corruption is inconsistency. The block information kept in the NN could be inconsistent with the blocks stored on DN. Block replica and block metadata can also be different from each other. This kind of inconsistency allows corrupted blocks to occasionally avoid interval block scanning and only be detected when `fsck` is executed manually. HADOOP-2905 discusses failures of `hadoop fsck / -move`. Specifically, this command mistakenly moves some corrupted files which causes a client exception at a later time. The functionality of this command itself can also be affected by inconsistency. HDFS-2376 shows how this command mistakenly reports a block as uncorrupted even when all replicas are corrupted.

Some data corruption is not detected because the data is already corrupt before being stored in HDFS. Hadoop assumes the data stored in HDFS is correct, causing any corrupted data to be found only when manually inspected or when used as job input. MAPREDUCE-4832 is an instance in which two AMs write different data into the same output file due to a failure. This corrupted data is then stored in HDFS. Similarly, HADOOP-5326 describes how data compression causes corruption in the intermediate stage of a MapReduce task. This results in a corrupted output file being stored in HDFS.

Silent data corruption can also be generated in memory and later propagated outside. For example, YARN-897 discusses a resource starvation problem caused by the "UsedCapacity" counter. This is a type of counters used to represent the resources assigned to a job. In memory, the counter values for different jobs are kept sorted in order to ensure fair resource allocation. However, in this bug, "UsedCapacity" is not always correctly updated in memory, causing resource starvation for large jobs.

B. Misreported Data Corruption

Misreported data corruption can be the result of block verification occurring before it should. Soon after it enters the active state, the NN starts block verification, performing block replication according to the block verification results. However, the states of other nodes may not allow it to do so. For example, in HDFS-3921, when the NN transitions into the active state, no block reports are sent from any DNs. This causes the NN to incorrectly consider all blocks as missing. Another example is the NN recovery in the scenario of multiple NNs described in HDFS-3605. When the NN restarts after crashing, all block operations in `edits.log` have to be

performed again to ensure the block information in the NN is consistent with the block information in DNs. However, as the DN is still actively performing block operations, block metadata inconsistencies can occur resulting in the NN incorrectly marking blocks as corrupt.

Misreported corruption can be caused by failures in block creation. Failures happen in block allocation, allocating DNs for blocks, and uploading block content. For example, HADOOP-2540 shows how `fsck` can falsely report zero sized files as corrupted. For blocks whose length is zero, Hadoop cannot separate newly created blocks without content from missing blocks. The never-created block could occur if the NN crashes after allocating a block for a file but before allocating any DNs to that block. Similarly, Hadoop cannot distinguish never-created blocks from missing blocks, as shown in HDFS-4212 and HADOOP-2275.

Misreported corruption could also be due to inaccurate block information. The block scanner collects block information periodically. When block scanning is coincidentally scheduled during the execution of block operations, block metadata information can change after the block scanner collects it but before the block scanner does block verification. Therefore the block scanner mistakenly marks the block as corrupted. HDFS-763 and HDFS-2798 are misreported corruptions because of inaccurate block information caused by deleting blocks and appending to blocks, respectively. Blocks under construction are also very susceptible to inaccurate block information. The NN blindly treats a new replica as corrupted if the new replica's length is not the same as the one recorded in the NN. This is discussed in HDFS-145 and HDFS-5438.

To summarize, the root cause of misreported corruption typically is that the NN blindly marks corrupted blocks based on block reports of unguaranteed accuracy. The NN does not have further verification methods to confirm the correctness of block reports with DNs. According to the bug description in HDFS-1371, an incorrect block report may cause all files in a DN to be marked as corrupted and may later delete them according to Hadoop's mechanism of handling corruption.

C. Imprecise Data Corruption Detection

Imprecise messages can come in many forms. For example, in HDFS-3874, a DN reports a "ChecksumError" to the NN and provides the NN with the DN where checksum error occurs. However, the DN provided does not exist, making it impossible for the NN to mark the corrupted block correctly.

In several cases, an imprecise message is generated as part of an exception. For example, in bug HDFS-1232 and HADOOP-1262, a "ChecksumException" is generated due to data corruption. While this tells the user corruption has occurred, it does not let the user know if the corruption is in the block or metadata file. Data corruption in either of them will cause a checksum mismatch and trigger HDFS to throw this exception. In HADOOP-2073, although the file `VERSION` on the DN is corrupted by an unexpected crash, the exception still reports an exception which indicates that the data directory is in an inconsistent state. However, in this case the exception has nothing to do with the data directory, but is a result from reading the corrupt `VERSION` file.

TABLE III. STATISTICS OF DATA CORRUPTION CAUSES.

Data corruption cause	**Number of incidents**
Improper runtime checking	25
Race condition	26
Inconsistent state	16
Improper network failure handling	5
Improper node crash handling	10
Incorrect name/value	5
Lib/command errors	4
Compression-related errors	4
Incorrect data movement	2

In cases where the exception is not a "ChecksumException", the exception message is either difficult to relate to data corruption or is unclear about the data corruption location. The latter is typically because the point where the exception is caught is far away from the point where corruption occurred. For example, in bug HDFS-3067, exception handling causes a block object to be set to null. This causes a "NullPointerException" to be thrown at a later time, making it unclear that data corruption has occurred. In bug HADOOP-2890, an "IOException" is thrown, informing the HDFS client that the actual size of a block is inconsistent with the block size information kept in the memory of the NN. However, the block size on the side of the NN shown in the exception message is not correct. The value shown, 134217728 (128K), is actually the size of a block object used in a RPC, not the value kept in the memory of the NN. Another confusing error message is shown in bug HADOOP-1955. This bug describes different messages given by two nodes during block replication. A corrupted block replica is sent from one DN to another. The DN which receives the data reports a checksum error and rejects the block. However, the DN sending the corrupted data only reports a "SocketException", hiding the data corruption. As a result, the NN keeps retrying replication without generating any messages related to data corruption.

V. DATA CORRUPTION CAUSES

In this section, we discuss the causes of the real data corruption problems we studied. Of the 138 data corruption incidents, 17 of studied incidents are misreported data corruptions and 40 of studied incidents have unclear or undocumented causes. Thus, we do not consider the causes of those data corruption incidents. The causes for the remaining 81 data corruption incidents can be broadly classified as having improper runtime checking, race condition, inconsistent state, improper network failure handling, improper node crash handling, incorrect name/value, lib/command errors, compression-related errors, and incorrect data movement. The statistics of different data corruption causes is shown in Table III. As shown, improper runtime checking, race condition, and inconsistent state are the three dominant types of data corruption causes. We now discuss each category in detail along with examples.

Improper runtime checking: Runtime input should be checked because improper input can cause data corruption. However, Hadoop does not strictly check Hadoop job inputs or Hadoop command parameters. There are various examples

in which input path, file name, quota configuration, or file ownership causes data corruption.

HDFS-3626 describes how creating files by executing the Hadoop `fs` command with an invalid path containing a double slash as the root directory can corrupt the `edits.log`. HDFS-1258 shows how `fsimage` is corrupted by changing the limit of the number of file and directory names allowed in the root directory.

Several data corruption incidents are a result of failing to check disk being full. For example, in bugs HDFS-1594 and HADOOP-6774, the `fsimage` and `edits.log` files are corrupted when the NN disk is full. Even when disk space is made available, restarting the NN is impossible.

Abnormal file names can cause corruption as well. In Hadoop-6017, renaming an open file with another name containing Java regex special characters could cause an unrecognized file name to be written to `edits.log`. This causes the NN and the 2NN to fail to load this file.

Unverified file ownership can cause data corruption. For example, in HDFS-1821, calling the method `createSymlink` of class FileContext with the kerberos enable option creates an irremovable symlink with an invalid user.

Race condition: Distributed systems are prone to race conditions and Hadoop is no exception. For example, several client may write to the same file at the same time, multiple threads may operate on the same block in a multi-threading Hadoop task.

From our observation, there are three different types of race conditions that Hadoop is prone to. The race can be between threads owned by Hadoop itself, between threads generated by launching MapReduce tasks, or between threads owned by Hadoop and threads generated to run tasks.

A race condition between threads owned by Hadoop itself usually occurs when concurrent threads are writing to the same variable or to the same file. For example, in HDFS-988, when the NN enters safemode, there may be pending `logSyc()` operations that occur in other threads. These threads call this method to flush save all modifications done by themselves to `edits.log`. If the administrator issues the `saveNameSpace` command at this time, partial writes will cause this log file to become corrupted. The threads are not limited to only being NN threads. In HDFS-4811, there are two such threads in the NN. Thread 1 is doing checkpointing in the NN itself by writing to `fsimage`, and thread 2 is fetching this file from another NN. After getting this file as a temporary file, thread 2 tries to rename the temporary file with the file name fsimage while thread 1 still holds open for this file. This results in `fsimage` being deleted or being incomplete. Race conditions can also occur because developers are neglect to concurrency risks. Java long and double are primitive types whose assignment operation is not atomic in 32-bit JVM. In the previously described bug MAPREDUCE-3931, the corrupted timestamp turns out to be the consequence of non-atomic change on the time stamp value among MapReduce threads.

A race condition between threads running the same task usually occurs when operations (e.g., write) occur on the same file. HADOOP-4397 describes how the `fuse-dfs` command causes data corruption when used by a multi-threaded process. In MAPREDUCE-4832, the RM could possibly assign two AMs to run the same job at the same time which causing concurrent writing operations to the same output file.

A race condition between threads owned by Hadoop and threads to run tasks is usually a result of conflicts when they try to update the same file. In HDFS-4178, the file descriptor “2” can be reused by threads from both sides. Since jobs can usually use file descriptor “2” as the channel for writing error messages, closing standard error stream for hadoop subprocesses with `2>&-` leaves potential risk that those subprocesses will re-use file descriptor “2” for opening file. Two threads using same file descriptor but for different purpose will corrupt the opened file.

Inconsistent state: Race conditions can cause data corruption in the form of an inconsistent block state. Failed write operations can also cause block state inconsistency. Hadoop typically keeps several replicas of each block. When updating a block, several file operations need to be done successfully and in a specific order to ensure a consistent update. If any of these operations fail or are done out of order, inconsistency can occur between the actual blocks stored on disk and their corresponding block state in memory. For example, HADOOP-752 shows an inconsistent state of a map in memory, which is responsible for keeping track of block locations. This inconsistent state is caused by some methods changing the map without acquiring its global lock, because of which the NN could not realize some blocks are non-existent on disk. HADOOP-3649 is a bug report that a data corruption in a list of corrupted blocks causes an inconsistency between the list of corrupted blocks and a list of corrupted block locations. HDFS-1260 shows how a failed block metadata update causes an inconsistency between genstamps.

Improper network failure handling: In Hadoop, nodes communicate with messages piggybacked on heartbeat messages. If these messages are not received in a timely manner, data corruption can occur. For example, in MAPREDUCE-4832 the RM assigns two AMs to run the same job at the same time because it does not receive heartbeat message from the first AM. This leads to data corruption when the duplicate jobs finish and try to write to the same output file.

Another case of improper network failure handling is transmission failures. For example, HDFS-1527 discusses a data transmission failure on 32-bit JVM. When the block size is bigger than 2GB, which is the limit on 32-bit JVM, the data transmission fails causing data corruption in the form of a partial file on the disk.

Improper node crash handling: We observed that data corruption can also be caused by improper node crashes handling. In process of creating files in HDFS, the client’s file creation request will cause the NN to create a new file in HDFS filesystem. A pipeline is then set up between the client, NN, and DN to write data to the file in HDFS. As the data is written, any node crash can possibly cause data corruption. For example, in HADOOP-2275, the data corruption occurs when the NN crashes before forming pipeline. In HDFS-3222, suddenly restarting cluster after forming pipeline but before all the data are written to the DNs causes partial blocks stored on DNs. After writing successfully all data to the DN, the block

```
try{
    ...
S1:
TransferFsImage.getFileServer(response.getOutputStream().nn.getFsImageName());
    ...
}catch(IOException e){
    ...
S5: response.sendError(...);
    ...
}

org.apache.hadoop.dfs.GetImageServlet.java
```

```
Void
getFileServer(OutputStream
outstream, File localfile)
throws IOException{
    ...
    try{
    S2: infile=new
FileInputStream(localfile);
        ...
    S3:
outstream.write(buf,0,num);
        ...
    }finally{
    S4: outstream.close();
        ...
    }
}

org.apache.hadoop.dfs.TransferFsImage.java
```

Fig. 3. An example of data corruption analysis.

pipeline will also calculate the checksum for the block, and write the checksum information into the block metadata file. As described in HDFS-1232, DN crashes before checksum is written to the metadata file but after the data has been stored on the DN makes a healthy block but corrupted checksum information.

Incorrect name/value: Data corruption can occur when some variables are used or updated incorrectly. For example, in YARN-897, releasing resources without updating the list of available resources disorders this list, corrupting the data structure representing this list in memory.

Lib/command errors: Data corruptions can occur when using some libraries or commands. For example, in HDFS-727, passing the block size argument to a method in `libhdfs` without making a appropriate type cast first causes the block size argument to be corrupted in memory.

Compression-related errors: Data corruptions can occur in the compression process. For example, in HADOOP-6799, the compression of HDFS data into several gzip [23] files leaves only the first gzip file readable. All remaining gzip files are corrupted because gzip header and 32-bit Cyclic Redundancy Check are not reset.

Incorrect data movement: Data corruptions can be caused by moving invalid or incomplete data into data directory. For example, HDFS-142 reports an incident of moving invalid files from temporary directory to normal data directory. Since these files fail block verification, they are reported as data corruptions.

Figure 3 illustrates how a corrupted `fsimage` is generated without being detected by the NN. This issue is reported in HADOOP-3069. In `GetImageServlet.java` the line labelled `S1` means the 2NN is trying to pull a new `fsimage` from the NN. It invokes the method `getFileServer` in `TransferFsImage` on the NN side. Note that the OutputStream of object `reponse` is passed to `getFileServer` as the parameter `outstream`, and used as the data channel between the NN and the 2NN. Code `S2` and `S3` try to read the request file, and send the data back to the 2NN. If the data is transferred successfully, code `S4` will close `outstream`. Thus, on the 2NN side, it will receive a complete and correct `fsimage` file.

However, if an error happens during data transmission, code `S4` will be executed before `getFileServer` throws an "IOException" to inform the 2NN of the transmission error. `S4` will close the connection from `response` to the NN. Therefore, when `GetImageServlet` catches this exception and tries to report an error message to NN in code `S5`, it causes an uncaught exception on the 2NN. This is because the connection is already closed, and the error message cannot be sent out. As a consequence, the NN never receives the error message, and thinks the file transfer is successful. But in fact, the `fsimage` on the 2NN is not completely transferred, or is even an empty file. When NN pulls this file from the 2NN to do recovery, it treats it as correct and does recovery based on an actually corrupted `fsimage`.

VI. Data Corruption Handling

Hadoop handles data corruption in two ways: 1) re-executing failed commands; and 2) recovering corrupted data by replicating uncorrupted blocks and deleting corrupted blocks. Problems can occur in both ways.

Although re-executing commands requires no further explanation, there are several steps involved in data corruption recovery. First, when a corrupted replica is detected and reported to the NN, it is marked as corrupted. The identifier of this replica is then added to a map which keeps track of all the replicas to be deleted. At the same time the corrupt replica identifier is removed from a second map which keeps track all valid replicas in HDFS. The NN next calculates how many valid replicas exist for each block. If the number of valid replicas for a block is below a predefined threshold value (e.g., three), the block is marked as under-replicated and then added to a third map which keeps track of all blocks needing replication. The NN then picks a block from the third map and replicates it. When replication is complete the newly created replica is added to the second map. Finally, the corrupted replica is deleted and its identifier is removed from the first map. Steps one and two are called data replication while the final step is called replica deletion. Data corruption handling problems can occur during data replication, replica deletion, and simple re-execution with percentage of 37%, 31%, and 31%, respectively. We now discuss each category in detail.

A. Data Replication

Data replication can go wrong in several ways. As described in HADOOP-1955, the NN may start block replication with a corrupted replica. Since DNs could detect and reject corrupted block replicas, corruption messages sent from the DNs to the NN may lead the NN to retry replication infinitely. Blocks may also never be replicated as shown in HDFS-2290. In this case, an under-replicated block is never added to the replica creation queue.

False alarms can also cause problems with data replication. For example, HDFS-2822 shows how a misreported corruption causes a block to be marked as needing replication. Similarly, an incorrectly handled error during replication can cause a problem at a later time. For example, HDFS-3874 describes how an on disk data corruption causes a block replication

problem. A "ChecksumException" is thrown but incorrectly handled, ultimately causing a non-existent remote DN to be reported to the NN.

B. Replica Deletion

As with data replication, errors can occur during replica deletion. Specifically, Hadoop can mistakenly delete uncorrupted replicas. HDFS-4799, HADOOP-4910 and HDFS-875 all report that the NN incorrectly marks uncorrupted replicas as replicas needing to be deleted because the NN does not distinguish between corrupted and uncorrupted replicas when determining if the system is over-replicated. HDFS-4799 shows how block verification can lead to block deletion of uncorrupted replicas. In this case, the block replication factor, responsible for determining when the system is over-replicated, is set to three. The bug manifests when there are six replicas for each block, three uncorrupted replicas and three outdated replicas. When the cluster is rebooted, the NN first receives the block reports of the three outdated blocks, correctly marking them as corrupted. However, when it next receives the block report about uncorrupted replicas, the NN finds the total number of replicas for this block is larger than the configured replication factor of three. It regards the three uncorrupted replicas as redundant and incorrectly deletes them from HDFS, leaving only the three outdated corrupt replicas in HDFS. In HADOOP-4910 and HDFS-875, the NN treats a corrupted replica as valid when calculating the number of over-replicated blocks. This incorrect calculation makes it possible for HDFS to delete uncorrupted replicas instead of corrupt replicas.

C. Simple Re-execution

When Hadoop job or command fails because of data corruption, the simplest way to handle it is to execute the command again. However, if the command cannot succeed, re-execution wastes resources. For example HADOOP-1955 describes unsuccessful but infinite attempts of the NN to replicate corrupted blocks, wasting resources in the process. HADOOP-459 shows how a memory leak can occur when `hdfsWrite` and `hdfsRead` are called repeatedly. HDFS-3440 describes that reading a corrupted `edits.log` can cause large memory consumption. HDFS-1940 reports the waste of disk space since extra replicas on disk are not deleted after a DN recovers from a disk failure.

Simple re-execution can also cause infinite loops. In HADOOP-3681, an infinite loop occurs when the Hadoop user tries to close a corrupted file. Both HADOOP-1911 and HADOOP-83 show that infinite loops can occur when users use `dfs -cat` command to access corrupted files.

VII. Related Work

Data corruption studies: Our work is most closely related to the study [24] done by Zhang et al. Their work studies Sun's ZFS as a case of end-to-end data integrity for file systems. They show that ZFS uses techniques such as checksum, replication and transactional updates for data integrity. This allows ZFS to be robust to disk corruption but less resilient to memory corruptions. Our work is complementary to this work, classifying data corruption problems that can occur in a distributed file system designed for use in a cloud computing system as opposed to identifying data corruption problems that can occur on a local file system.

Data corruption is an issue that has been widely discussed in data storage. Several studies [15], [24]–[30] have shown that data corruption can be caused by bit rot in hardware, firmware bugs, lost writes, misdirected writes, DRAM errors, or software bugs. Most of these studies are focused on disk failure or memory failures. For example, studies [25], [26] have found that DRAM failures occur more frequently than expected. Other studies [27], [31], [32] show that disk failures are prevalent in deployed systems. Our work is complementary to these studies, characterizing software-induced data corruption problems.

Data corruption detection frameworks: Data corruption correction frameworks exist which attempt to identify and fix data corruption. EXPLODE [29] is a framework for checking real storage systems for crash recovery bugs by quickly driving a storage system into corner cases. RedMPI [33] is a MPI profile library to help detect and correct soft-errors or single bit flips in memory while executing MPI applications. TAC [30] injects pointer-corruption and field-level corruption into MySQL, to show that MySQL is vulnerable to data corruption, especially metadata corruption. The insights in our characteristic study can help each of these works in ensuring high data corruption coverage.

Bug characteristic studies: Existing work has examined different categories (e.g., performance, misconfiguration) of bugs exposing existing problems in real-world systems. Performance bug [34] studies identify performance degradation and resource waste in real-world systems. Concurrency bug [35] studies describe concurrency bug identification patterns, ways in which concurrency bugs manifest, and the effectiveness of concurrency bug fix strategies. A study of misconfiguration errors [36] in different systems shows that misconfigurations are a major cause of system failures. A study of OpenStack fault resilience [37] exposes various problems in OpenStack APIs. Although these studies all focus on different categories of bugs none of them focus on data corruption bugs. Our work is complementary to existing work, providing the first comprehensive data corruption bug study of a widely deployed cloud computing system.

VIII. Conclusion and Future Work

In this paper, we present a characteristic study over 138 real world data corruption incidents caused by software bugs in cloud systems. Our study reveals several interesting findings: 1) the impact of data corruption is not limited to data integrity; 2) data corruption can be caused by various software bugs including both external and internal to HDFS; 3) existing data corruption detection mechanisms are insufficient with both mis-detections and false alarms; and 4) existing data corruption handling schemes may not only fail to prevent data loss but also trigger additional problems such as resource exhaustion.

In the future, we plan to develop efficient detection and handling schemes to combat real world data corruption problems. We propose to trace data-related operations in the cloud and perform anomaly detection over the operation logs to detect potential data corruption problems. Our goal is to

identify those anomalous data processing events early and prevent them from causing data corruptions.

IX. Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] "Aws—amazon simple storage service(s3)," http://aws.amazon.com/s3, 2014.

[2] "Google compute engine," https://cloud.google.com/compute, 2014.

[3] "Welcome to apache hadoop," http://hadoop.apache.org, 2014.

[4] C. Brooks, "Cloud storage often results in data loss," http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html, 2011.

[5] L. Mearian, "Facebook temporarily loses more than 10% of photos in hard drive failure," http://www.computerworld.com/s/article/9129263/Facebook_temporarily_loses_more_than_10_of_photos_in_hard_drive_failure, 2009.

[6] D. Hamilton, "Corrupt icloud data causes ios springboard home screen crash(with fix!)," http://www.macobserver.com/tmo/article/corrupt-icloud-data-can-cause-ios-springboard-home-screen-crash, 2013.

[7] M. Small, "Microsoft onedrive file sync problems," https://blogs.kuppingercole.com/small/2014/09/01/microsoft-onedrive-file-sync-problems, 2014.

[8] L. M. Mahon, "Hsbc hit with multi-million pound fine," http://www.insuranceage.co.uk/insurance-age/news/1469060/hsbc-hit-multi-million-pound-fine, 2009.

[9] M. calore, "Magnolia suffers major data loss,site taken offline," http://www.wired.com/business/2009/01/magnolia-suffer, 2009.

[10] C. Marsh, "Data integrity in the cloud," http://www.wwpi.com/index.php?option=com_content&id=12800:data-integrity-in-the-cloud&Itemid=2701018, 2011.

[11] R. Harris, "Data corruption at massive scale," http://www.zdnet.com/article/data-corruption-at-massive-scale, 2012.

[12] C. Balding, "A question of integrity: To md5 or not to md5," http://cloudsecurity.org/blog/2008/06/25/a-question-of-integrity-to-md5-or-not-to-md5.html, 2009.

[13] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, 2010.

[14] G. Amvrosiadis, A. Oprea, and B. Schroeder, "Practical scrubbing: Getting to the bad sector at the right time," in *DSN*, 2012.

[15] W. Jiang, C. Hu, A. Kanevsky, and Y. Zhou, "Is disk the dominant contributor for storage subsystem failures? a comprehensive study of failure characteristics," in *FAST*, 2008.

[16] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *SIGMETRICS*, 2007.

[17] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *SOCC*, 2013.

[18] "Hadoop issue tracking," http://hadoop.apache.org/issue_tracking.html, 2014.

[19] "Gridmix," http://hadoop.apache.org/docs/r1.2.1/gridmix.html, 2014.

[20] "Hbase-apache hbase," http://hbase.apache.org, 2014.

[21] "Apache hive tm," http://hive.apache.org, 2014.

[22] "Welcome to apache pig!" http://pig.apache.org, 2014.

[23] "Gzip file format specification version 4.3," http://tools.letf.org/html/rfc1952, 2014.

[24] 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.

[25] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," *ACM SIGARCH Computer Architecture News*, 2012.

[26] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," in *SIGMETRICS*, 2009.

[27] B. Schroeder, S. Damouras, and P. Gill, "Understanding latent sector errors and how to protect against them," *ACM Transactions on storage (TOS)*, 2010.

[28] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani, "Dealing proactively with data corruption: Challenges and opportunities," in *ICDEW*, 2011.

[29] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *OSDI*, 2006.

[30] 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.

[31] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" in *FAST*, 2007.

[32] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, 2008.

[33] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC*, 2012.

[34] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[35] E. S. Shan Lu, Soyeon Park and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[36] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *SOSP*, 2011.

[37] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva, "On fault resilience of openstack," in *SOCC*, 2013.

[38] H. Nguyen, D. J. Dean, K. Kc, and X. Gu, "Insight: In-situ online service failure path inference in production computing infrastructures," in *ATC*, 2014.

[39] K. Kc and X. G. ELT, "Efficient log-based troubleshooting system for cloud computing infrastructures," in *SRDS*, 2011.