Understanding Issue Correlations: A Case Study of the Hadoop System

Jian Huang
Georgia Institute of Technology
jian.huang@gatech.edu

Xuechen Zhang
University of Oregon
xczhang@uoregon.edu

Karsten Schwan
Georgia Institute of Technology
karsten.schwan@cc.gatech.edu

ABSTRACT

Over the last decade, Hadoop has evolved into a widely used platform for Big Data applications. Acknowledging its wide-spread use, we present a comprehensive analysis of the solved issues with applied patches in the Hadoop ecosystem. The analysis is conducted with a focus on Hadoop's two essential components: HDFS (storage) and MapReduce (computation), it involves a total of 4218 solved issues over the last six years, covering 2180 issues from HDFS and 2038 issues from MapReduce. Insights derived from the study concern system design and development, particularly with respect to correlated issues and correlations between root causes of issues and characteristics of the Hadoop subsystems. These findings shed light on the future development of Big Data systems, on their testing, and on bug-finding tools.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Performance, Reliability, Security

Keywords Hadoop, Issue Correlation, Bug Study, Big Data

1. Introduction

Recent extensive work on data-intensive applications and on the systems supporting them are mirrored by substantial efforts to improve and enhance well-established frameworks like Hadoop [3]. Hadoop is an open-source project which has a considerable development and deployment history, dating back to the year of 2002. Its users include Twitter [21, 32], Facebook [25], Yahoo! [39], Cloudera [34], and many startup companies, with significant contributions to Hadoop made by both the academic and industry communities [12].

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA
©2015 ACM. ISBN 978-1-4503-3651-2/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2806777.2806937

Developing and deploying Big Data systems is complex, and as a consequence, they have experienced a wide range of issues, reflected in extensive bug reports and patch histories. The importance of these issues is underlined by their repeated occurrence during system evolution, resulting in data loss or even catastrophic failures [20, 34, 47]. At the same time, given the complex codebases of Big Data systems, issue identification is challenging, even with advanced bug-finding tools [19, 29, 35]. A well-known reason for this difficulty is the many interactions between their different software components [13, 15, 42]. In the Hadoop ecosystem, for example, YARN (Apache Hadoop NextGen MapReduce) is responsible for resource scheduling for MapReduce jobs, while MapReduce interacts with HDFS for accessing input and output data. In this context, issues observed in a Hadoop job execution could result from the MapReduce framework, or from schedulers in YARN, or from I/O streams in HDFS. Furthermore, there are strong correlations between the different components even within a single subsystem, for instance, a bug in network topology management could violate data replication policies, which causes data loss in HDFS.

This paper studies and quantitatively analyzes the issue correlations in Big Data systems, with the goal of providing references and hints to help developers and users avoid potential issues and improve system performance and reliability for future data-intensive systems. We examine in detail two essential components of the Hadoop ecosystem evolved over the last decade: HDFS (Hadoop Distributed File System) [22] and MapReduce [23], which are the storage and computation platforms of the Hadoop framework. Our study focuses on solved issues with applied patches that have been verified by developers and users. Specifically, we examine 4218 issues in total, including 2180 issues from HDFS and 2038 issues from MapReduce, reported and solved over the last six years (from 10/21/2008 to 8/15/2014). We manually label each issue after carefully checking its descriptions, applied patches, and follow-up discussions posted by developers and users. We conduct a comprehensive and in-depth study of (i) correlations between issues and (ii) correlations between root causes of issues and characteristics of the Hadoop subsystems.

Target	Finding	Suggestion
Bug-finding Tools	Most issues do not depend on external components; they are internally correlated within a single subsystem; 33.0% of issues have similar causes (§4). The logging subsystem is error-prone (§5.4.2)	Log-based bug-finding tools should place a higher priority on the logs of the components in which the issue appears. Logs should be audited to reduce false positives.
File System	The file system semantics of traditional file systems like EXT4 are widely used in HDFS; many file system issues in HDFS are induced by strictly ordered operations in distributed environment (§5.2.1).	Lessons/experiences from traditional file systems could be applied to distributed file systems. For instance, similar optimizations and features like fadvise and fsck in EXT4 have been implemented in HDFS.
Storage	Problematic implementation of rack replication and data placement policies may cause data loss (§5.2.2).	Policy checking tools are required to validate the correctness of the applied policies for data placement, distributed caching, etc.
Memory	Memory leaks happen mostly due to uncleaned or unclosed objects; the stale objects are apt to cause unexpected errors; high garbage collection overhead (§5.2.3).	Lightweight or memory-friendly data structures are preferred; implement object cache to reuse objects whenever possible.
Cache	Cache management in HDFS is centralized and user-driven. It uses different data placement and destage algorithms to explore data locality (§5.2.4).	Programmers can leverage these characteristics for performance optimizations.
Networking	Wrong networking topology can cause data loss when unreliable connections appear (§5.2.5).	A set of tools are needed to check and validate networking setups (e.g., topology, configuration).
Programming	Half of programming issues are related to code maintenance(§5.3.1); inconsistency problems frequently happen due to interface changes(§5.3.2); 19% of the programming issues are caused by inappropriate usage of locks (§5.3.3) and typos(§5.3.4).	As new input/output interfaces are implemented, additional efforts are needed for their performance tuning and data structure optimization.
Configuration	Many configuration parameters are relevant to performance tuning, as misconfiguration can easily lead to suboptimal performance (§5.4.1).	Leverage auto-tuning software or configuration checkers to learn how parameters can affect system performance.
Documentation	Documentation issues are usually overlooked in system development (§5.4.3).	Avoid biased or inconsistent description which can lead to users' misunderstanding.
Testing	Not all failure cases can be simulated in Hadoop's fault injection platform; the simulation of large-scale clusters like the MiniCluster in Hadoop can make testing results less accurate (§5.4.4).	An open and shared large-scale testing platform is an urgent need; advanced simulation techniques for large-scale distributed Big Data systems are needed.
Fault Handler	A significant fraction of issues cause failures (§6.1); system reliability is most vulnerable as a consequence (§6.2); exception handling and retrying are commonly used in Hadoop; unlike other findings [47] (fault handler not implemented), we find that many issues are caused by the inappropriate usage of exceptions and bugs in their implementation (§6.3).	The exception type should be specified as concretely as possible; a program analyzer may be required to check if all possible exception cases have been covered in the implementation of fault handlers.

Table 1. Our findings and consequent suggestions for improving distributed data-intensive systems.

Key findings from our study are as follows:

- Most of the issues (79.3% in MapReduce and 94.7% in HDFS) do not depend on external factors, i.e., they are not closely related to issues in other subsystems. In other words, even in complex distributed systems like Hadoop, many issues remain relatively centralized in their corresponding subsystems.
- About half of the issues are internally correlated. We observe that up to 33.0% of the issues have similar causes, 10.5% of the issues arose because of fixing other issues (fix on fix), and up to 20.1% of the issues block other issues as one error could be caused by multiple events or issues.
- The root causes of the issues have strong correlations with the subsystem characteristics of Hadoop. For example, HDFS has 349 issues related to the implementation of file system semantics, file data snapshot, and metadata checkpointing. More findings are summarized in Table 1.

The remainder of this paper is organized as follows. The methodologies used in our study are described in §2. In §3, we present the overall patterns of the issue distribution. We then study the correlation between issues in §4 and the correlation between issues and characteristics of various aspects of Hadoop subsystems in §5. Consequences of these results, impact, and reactions to issues are discussed in §6. In §7, we present related work on issue studies of distributed systems and the relevant debugging tools. We conclude the paper in §8.

2. Methodology

This section outlines our reasons for selecting HDFS and MapReduce as target open-source Big Data systems. We then introduce the methodology used for analyzing the examined issues, and discuss how our patch database HPATCHDB helps further patch studies.

2.1 Selected Open Source Systems

We select HDFS (storage) and MapReduce (computation) as target systems because their analysis can shed light on a broad set of other data-intensive systems, for three reasons.

First, as the core components of the Hadoop ecosystem, HDFS and MapReduce have been developed into mature

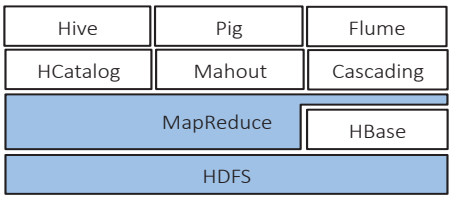


Figure 1. A subset of systems in Hadoop ecosystem. Many systems and tools for data processing and resource management being developed are based on HDFS and MapReduce.

Systems	HDFS	MapReduce
Reported issues	6900	5872
Closed issues	2359	2340
Studied issues	2180	2038
Sampled period	10/21/08-8/15/14	6/17/09-7/27/14

Table 2. Number of reported and solved issues for HDFS and MapReduce until 8/15/2014. We examine a total of 4218 issues, covering 89.8% of the closed issues.

systems over the last decade, and they are widely used to store and process large data sets from both enterprise [21, 25, 32, 34, 39] and scientific workloads [9, 40], representing the state-of-the-art distributed systems.

Second, as shown in Figure 1, other data-intensive systems are architected based on HDFS and MapReduce, including Hive (data summarization, query and analysis) [6], Pig (high-level platform for MapReduce programs generation) [8], Flume (collection, aggregation and processing for streaming data) [2], HCatalog (data cleaning and archiving) [5], Mahout (machine learning and data mining) [7], Cascading (framework for building data processing applications) [1], and HBase (distributed key-value store) [4, 28].

Third, HDFS and MapReduce share the same development and deployment tools (e.g., Ant and Maven for Java) as used in other data management and analytics systems [24]. The issues patterns in these subsystems would be reflected in the study of the essential components in Hadoop.

2.2 Sampled Data Sets

For a comprehensive study of the selected core systems, we manually examine most of the solved issues with applied patches from the issue repositories of HDFS and MapReduce. As shown in Table 2, there are 6900 issues reported between Oct. 21, 2008 and Aug. 15, 2014 in HDFS, 5872 issues reported between Jun. 17, 2009 and Jul. 27, 2014 in MapReduce. Among these issues, there are 2359 and 2340 closed issues (i.e., the issue was fixed, the attached patches passed the test and worked as expected) in HDFS and MapReduce, respectively. We only examine these closed issues, as for those, it has been established that both issues and corresponding solutions are valid. Duplicate and invalid issues are excluded in our sampled data sets to reduce sampling error. With these criteria, we used roughly 1.6 years to sample and analyze 2180 and 2038 closed issues in HDFS and MapReduce, respectively.

In order to precisely analyze and categorize each sampled issue, we tag each issue with appropriate labels after checking its description, patches, follow-up discussions, and source code analysis posted by developers and users. The labels include `IssueID`, `CreatedTime`, `CommitTime`, `SubComponent`, `Type`, `Priority`, `Causes`, `Consequence`, `Impact`, `Keyword`, `CorrelatedIssues` and `Note`. To minimize errors caused by human factors during the manual classification, each issue is inspected at least twice, the complicated and unclear issues are examined by two observers separately, and then discussed until consensus was reached.

To track the issue correlations, the labels `SubComponent`, `CorrelatedIssues`, and `Keyword` are used to build the connections with relevant components and system features. Note that one issue may have multiple keywords, since it is possible that the issue is caused by several factors or events, and multiple components are involved. For instance, in the issue `MR-5451`[1], a configurable parameter `LD_LIBRARY_PATH` cannot be set and parsed correctly on Windows platforms, resulting in MapReduce job failures. This issue will be recorded with keywords of `Configuration` and `Compatibility`. With these labels, it is easier for us to categorize and index each issue. We place all of the examined issues into our issue database HPatchDB. More detailed analytics and examples are presented throughout the paper to show how tagging and HPatchDB are performed.

2.3 Use Cases with HPatchDB

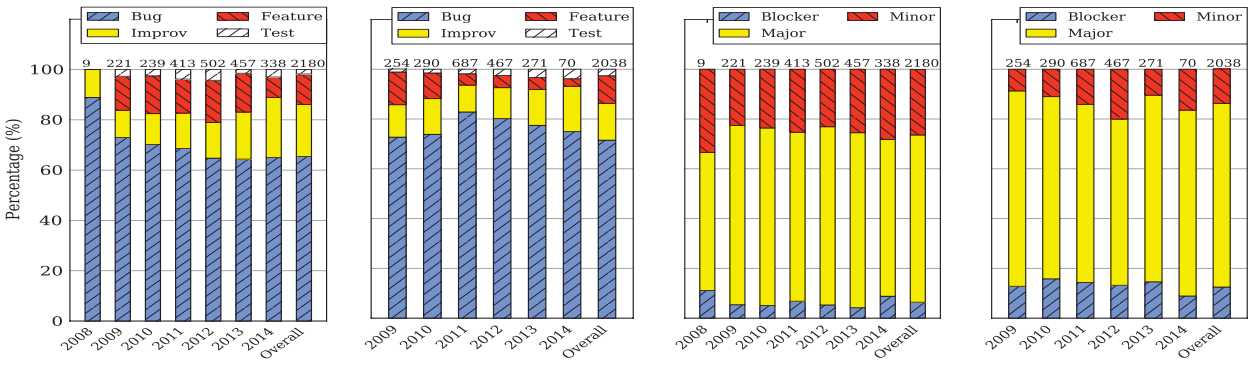
The labels with each issue identify its characteristics, and represent its root causes, correlated components, and impact on systems, etc. HPatchDB is useful for programmers, as they can use HPatchDB to find solved issues with characteristics similar to those of their new or unsolved issues, thus learning useful lessons and experiences from them. This insight is validated by our finding: 33.0% of issues have similar causes (§4). HPatchDB is also useful for bug-finding and testing tools, to conduct further studies from different viewpoints and to filter out issues of specific types. System designers can use HPatchDB to refer to the issue patterns of specific components (e.g., file system, cache management). As future work, we wish to automate the issue classification procedure to enrich HPatchDB.

3. Issue Overview

Before we discuss issue correlations, we first present the overall patterns of examined issues, and investigate how these issues are distributed.

Issues are categorized into four types with three priority levels by leveraging the `Type` and `Priority` labels. Issue types include `Bug` (issues cause systems to behave unexpect-

[1] MR represents MapReduce in the paper.



(a) Types-HDFS (b) Types-MapReduce (c) Priority-HDFS (d) Priority-MapReduce

Figure 2. Distribution of solved issues in HDFS and MapReduce.

edly), `Improvement` (software stack improvements for performance and code maintenance), `New Feature` (new functionalities), and `Test` (unit and failure tests). The three levels of priorities are `Blocker` (immediate fix needed as this type of issue blocks the software release), `Major` (important issues that have serious impact on the functionality of the system), and `Minor` (have minor impact on the functionality of the system in general but need to be fixed).

3.1 Issue Types

Issue distributions over time for different types and priority categories are depicted in Figure 2. While the number of solved issues changes each year, the percentages of different types and priorities are relatively stable.

As shown in Figure 2(a) and Figure 2(b), `Bugs` dominate solved issues over the past six years. They occupy 59.1% and 67.6% of issues on average in HDFS and MapReduce respectively. A large fraction of issues are related to `Improvement`. They include performance improvement, code maintenance, serviceability improvements, etc. Along with each patch, tests are conducted to evaluate it, but we find that many issues are reported because of bugs in the testing platform rather than in the applied patch (§5.4.4).

3.2 Issue Priority

Figure 2(c) and Figure 2(d) show the percentage breakdown of issues according to their priorities, demonstrating a significant number of `major` issues in both HDFS and MapReduce. These issues could cause job failures, incorrect outputs and runtime errors. As for `blocker` issues that may cause serious failures and unexpected behaviors such as memory leaks, they occupy 9.4% of total issues on average. A considerable subset of examined issues are `minor` issues that are normally ignored in other studies [20, 47], but they also constitute a large fraction of issues in Hadoop. These `minor` issues should not be neglected, as they can significantly affect system availability and serviceability, and some of them are not easily fixed. For instance, a typo in the name of a library path (configuration issue) could block system startup.

Type	Num of correlated issues	0	1	2	3	≥4
EXT	HDFS	94.7%	4.8%	0.5%	0.0%	0.0%
	MapReduce	79.3%	17.1%	2.8%	0.5%	0.3%
INT	HDFS	52.7%	32.8%	9.1%	3.1%	2.3%
	MapReduce	59.3%	32.7%	5.6%	1.3%	1.0%

Table 3. Percentage of issues externally and internally correlated with other issues. EXT denotes external correlation, INT denotes internal correlation.

In large-scale distributed systems, these minor issues may require substantial effort for only "one-line" fixes. Our study, therefore, covers all of these issues, with a focus on their correlations with other issues (§4) and with system characteristics (§5).

Interestingly, we observe similar issue patterns over time for both HDFS and MapReduce. This gives credence to our supposition that a study of the Hadoop ecosystem can shed light on other data-intensive systems.

4. Issue Correlation

This section further examines the correlations between issues. We distinguish correlations between issues of different subsystems in the Hadoop ecosystem (external correlation), or between issues of different components in the same subsystem (internal correlation).

The purpose of this study is to better understand the external and internal correlationships between issues and the potential implications on tracking and fixing issues in distributed systems. We record the number of correlated issues, the `IssueID` of correlated issues, and the relationships (i.e., similar causes, blocking other issues and broken by other issues) in HPATCHDB. Table 3 shows the percentages of issues for different numbers of correlated issues (0 to ≥4).

4.1 External Correlations

We first study the externally correlated issues. These issues occur in one subsystem (i.e., HDFS or MapReduce), but

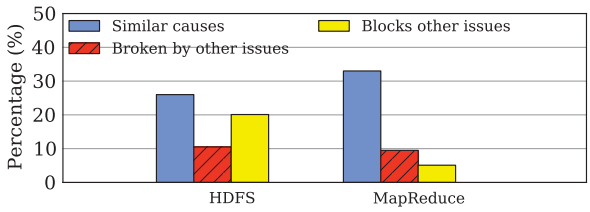


Figure 3. Classification of correlated issues.

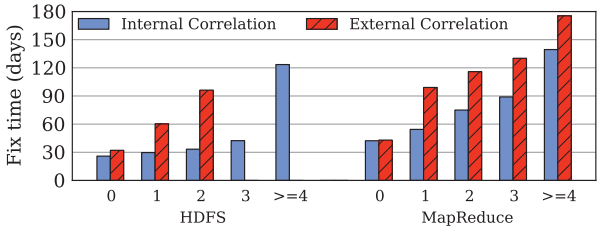


Figure 4. Efforts required to fix an issue, on average.

are related to issues in other subsystems (see §2.1), such as HBase, YARN, etc.

In HDFS, we find that a significant number of the examined issues (94.7%) are independent (the number of correlated issues is 0) without any relationships with issues from other subsystems. This is due to the fundamental role of HDFS in the Hadoop ecosystem, residing at the bottom of the software stack. For the issues related to one other issue in other subsystems, 62.5% of them are related to Hadoop Common subsystem, others are distributed across HBase (15.0%), YARN (10.0%), MapReduce (5.0%), Slider (7.5%), and etc. For the issues related to two issues in other subsystems, all are from Hadoop Common. We did not find any issues in HDFS that correlate with more than two issues from other subsystems. This indicates that bug-finding tools leveraging logs can narrow their searches for root causes by first considering the logs of the subsystems in which issues are observed and then consider other correlated subsystems (especially for cross-systems debugging and testing) [48, 49].

In MapReduce, we also observe that most of the examined issues (79.3%) are not related to any issues from other subsystems. Among the issues related to one issue in other subsystems, we find a large fraction of them to be relevant to YARN (46.3%), Hadoop Common (29.9%), HDFS (9.0%), Hive (6.0%), and others (8.8%) such as ZooKeeper, Mahout, and Spark. Among the issues whose correlation factor is 2, 72.7% are from Hadoop Common. For the issues related to more than two issues from other subsystems, most involve the YARN and Hadoop Common subsystems. Compared to HDFS, issues in MapReduce have more correlations with other subsystems, especially with YARN. This is reflected in the evolution of the computation framework in Hadoop, leading to more MapReduce issues being involved with YARN.

4.2 Internal Correlations

We now examine internally correlated issues, i.e., those with relationships to other issues in the same subsystem. Compared to the external correlation analysis, we find that more issues are internally correlated, as shown in Table 3. HDFS and MapReduce have similar distribution of issues classified according to the number of their correlated issues.

To further study internal correlations, we categorize the relationships between issues into three types: similar causes, broken by other issues (i.e., the issue is caused by fixing other issues) and blocking other issues (i.e., the issue needs to be fixed before fixing other issues). As shown in Figure 3, 26.0-33.0% of the issues have causes similar to other issues in the same system, indicating that similar issues happen repeatedly and it is worth the effort to avoid duplicate issues. This finding also inspires us to provide a new HPATCHDB feature with which users can discover issues with similar causes, facilitating learning from existing solved issues how to fix their current problems.

We further observe that about 10.5% of the issues were broken by other issues. Most of these issues happen because of the fact that one function depends on another subfunction. And conversely, we find that 5.1-20.1% of the issues could block other issues, indicating that one failure could be caused by multiple events or issues.

4.3 How Hard Is It to Fix Correlated Issues?

We conduct a quantitative study on the effort required to fix a correlated issue in Hadoop. We use the `CreatedTime` and `CommitTime` labels to calculate the fix time, which serves as an indicator for issue complexity. Because we report an average over thousands of sampled issues, this can significantly improve the accuracy of our indicator statistically and reduces the reporting errors caused by artificial factors such as developers delaying patch commits.

As shown in Figure 4, correlated issues require almost twice the fix time of independent issues, and fix times increase dramatically with increasing correlation factors. In other words, compared to internal correlated issues, external correlated issues require more effort. In HDFS, we do not find correlated issues whose correlation factor is more than 2 (see Table 3), therefore, they are not shown in Figure 4.

Summary: Most of the issues are externally independent. They are not closely related to issues from other subsystems. More issues are internally correlated, and the effort required to fix these issues increase significantly as the correlation factor increases. The good news is that most of the correlated issues only relate to one other issue.

5. Correlation with Distributed Systems

In this section, we examine the correlation between root causes of issues and subsystem characteristics of Hadoop. We first describe the issue classification and demonstrate its distributions among these categories. We then study how

	Subtype	Description
Systems	File system	Issues related to file system, e.g., inode and namespace management
	Storage	Issues related to block operations, e.g., data replication and placement
	Memory	Memory management and relevant issues, e.g., out of memory and null pointers
	Cache	Issues related to operations in caches, e.g., data caches of file system and object caches
	Networking	Issues related to networks, e.g., errors in protocols/topology and timeout
	Security	Issues related to security, e.g., access control and permission check
Programming	Code Maintenance	Code maintenance, e.g., data structure reorganization and dead code cleanup
	Interface	Issues caused by interface changes, e.g., added, combined and deleted interfaces
	Locking	Issues caused by inappropriate lock usage
	Typo	Issues caused by typos
Tools	Configuration	Inconsistent/wrong/missed configurations
	Debugging	Issues in the process of debugging
	Docs	Issues related to documents for system/function description
	Test	Issues related to unit and failure tests

Table 4. Issue classification based on their correlations with major aspects in Hadoop.

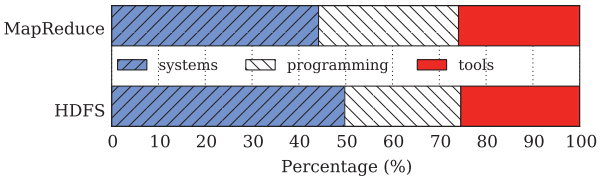


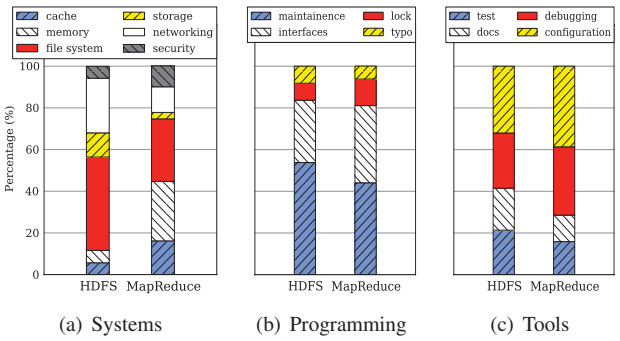
Figure 5. Overall distribution of issues in the `Systems`, `Programming`, and `Tools` categories.

causes are correlated to the characteristics of various aspects. Concrete cases will be given to facilitate our discussions.

5.1 Issue Classification

We classify issues into the three types shown in Table 4: `Systems`, `Programming`, and `Tools`. Each type is further divided into a few subtypes. Note that one issue may belong to several subtypes, so we use its primary tag for statistical analysis, but also keep the other tags in HPATCHDB for indexing and further studies.

As shown in Figure 5, the overall distributions of issues among the three categories in HDFS and MapReduce are similar. For `Systems` issues, we find that a large fraction of HDFS issues relate to file system interacting frequently with underlying systems and hardwares, while surprisingly more MapReduce issues relate to memory aspects (further discussed in §5.2). As for `Programming` and `Tools`, we obtain similar numbers for HDFS and MapReduce, which could be because they are both using the Java programming language and the same development and deployment tools. This finding further underlines our expectation that our studies of a



(a) Systems (b) Programming (c) Tools

Figure 6. Issue distribution across major aspects (i.e., systems, programming, and tools) of Hadoop systems.

subset of systems in the Hadoop ecosystem have relevance to other data-intensive infrastructures.

5.2 System Issues

In this section, we will discuss any system issues relevant to file system, storage, memory, cache, networking, and security respectively.

5.2.1 File System

HDFS is a distributed file system which not only implements the basic file system semantics, but also supports high data reliability with techniques such as metadata snapshot and checkpointing. We observe that a large number of issues (349 issues in HDFS, 117 issues in MapReduce) are related to `file system`, as shown in Figure 6(a).

The file system semantic issues (29.2%) include namespace management, fsck, file operations (i.e., read, write, delete, rename), file permissions, file format, etc. *We observe that the basic concepts used in these systems are similar to conventional file operations*, e.g., the management and checking for file access permissions in the EXT4 file system. Many optimizations used in traditional file systems have been implemented in Hadoop. For example, the cache policies used in traditional file systems can be used in HDFS, as exposing underlying architecture details (e.g., OS buffer cache) to upper layers could bring benefits for applications. `HDFS-4817` and `HDFS-4184` are two examples showing the HDFS improvements by making its advisory caching configurable on I/O patterns. Another example is `fsck` which is also implemented in HDFS for file checking.

Furthermore, *we find many consistency issues in conventional systems also exist in HDFS*. For example, violating the rule for inorder updates could result in inconsistency issues, breaking systems due to crash vulnerability [33]. In HDFS, the consistency issues become more complex, e.g., a file is deleted from NameNode but it may still reside in DataNode. Since some issues had been solved with the techniques in conventional file systems, we believe the techniques, experiences and lessons gained from those legacy systems can also be applied in distributed file systems.

On the other hand, *the distributed HDFS system is designed for solving much larger scale data sets than legacy file systems, it is more likely to suffer from performance issues*. For example, removing a directory in HDFS with millions of files can be slow using the conventional approach. User requests cannot be served for minutes while the deletion is in progress. The approach to reduce the latency experienced by users is incremental file deletion. Specifically, the target subdirectory to be deleted will be removed from the directory tree first to prevent further changes. Then the files and directories under it are deleted with a lock protected. Between two deletes, the lock is relinquished to allow NameNode operations for other clients.

Moreover, we find *many issues are caused by the in-order of log operations, happening frequently in file snapshot* (21% of `file system` issues). The file snapshot is incrementally built with operations recorded in a log after performing the last snapshot. It is error-prone when applying the logged operations to create a new snapshot. For instance, during system recovery, a block update operation may incorrectly happen after a file deletion, resulting in recovery failure and data corruption (`HDFS-6647`).

5.2.2 Storage

The block-layer design of the storage system in Hadoop provides reliability and availability via block replication, load balancing, and recovery after disk failures. A number of issues (90 in HDFS, 12 in MapReduce) are related to these designs. Specifically, a small fraction of storage issues occur when the computation framework accesses its input and output data with optimizations on data movement, intermediate data management, interfaces for new data formats.

Hadoop employs replication techniques to guarantee high reliability even after hardware failures. We observe, however, that *the implementation of fault-tolerance functionality is error-prone*, e.g., logic errors in the `rack placement policy` could cause loss of data replicas. In HDFS, replicas should not be placed into the same rack for data reliability guarantee. However, due do decommissioning of nodes for OS upgrade in a running cluster, about 0.16% of blocks and all replicas of the blocks could be in the same rack, leading to higher risk of data loss when the rack is taken offline. To solve this issue, policy checkers or tools, e.g., `Hadoop fsck`, are required to verify this for large amounts of data blocks (`HDFS-1480`). Another typical example in Figure 7(a) shows that asynchronous block operations may cause data loss. The NameNode receives the block report while the block in DataNode is still under construction. However, if the DataNode fails and NameNode accepts the block report with an old generation stamp while the write pipeline recovery is working, the corrupt block will be marked as valid.

5.2.3 Memory

Hadoop is designed to process Big Data with extremely high concurrency, but severe issues happen with large data sets under high memory pressure in JVM [41], and memory bugs (e.g., memory leaks) make matters worse. For instance, the JVM creates a thread with 512 KB stack size by default, but 4096 or more `DataXceiver` threads are created normally on DataNode, this occupies a large amount of memory (`HDFS-4814`). Similar cases are observed in reported issues such as `HDFS-5364` and `HDFS-6208`.

Figure 7(b) shows a memory leak case (`MR-5351`), where the unclosed `FileSystem` Java object will cause a memory leak. This is because the common practice that on job completion the need for cached objects to be cleaned or closed is often ignored by developers. Further, although Java provides the garbage collection (GC) mechanism to clean stale objects automatically, the system suffers from large GC overhead and unexpected errors caused by objects that are not cleaned in a timely fashion (e.g., `HDFS-5845`).

Under high levels of concurrency using certain data structures produces significant memory pressure. We found a number of `Improvement` patches addressing this issue. A simple approach is to use memory-friendly objects, e.g., replacing the standard object with LightWeightGSet. One can also use an object cache to reduce the overhead of object allocation and deallocation (`HDFS-4465`). Furthermore, programmers also need to minimize the usage of expensive operations on objects, e.g., `ConcurrentSkipListMap.size` may take minutes to scan every entry in the map to compute the size (`MR-5268`).

5.2.4 Cache

Caches in Hadoop are not only used to bridge the performance gap between memory and disk (e.g., distributed cache for data blocks on DataNode, metadata cache on NameNode, I/O stream cache), but they are also used to avoid duplicate operations, such as token cache, socket cache, file descriptor cache, and JobContext cache.

Many cache-specific issues (especially on the performance aspect) are related to their configurations in Hadoop. Differing from traditional page caches, Hadoop uses user-driven and centralized cache management [10]. Users can employ `cache directives` to specify which file path should be cached and how to cache it among distributed DataNodes. After scanning the directives, the NameNode sends cache/uncache commands to DataNodes to manage the placement of block replicas in the caches of different DataNodes. The NameNode needs to periodically rescan cache directives and adjust block placement to reflect changes in cache directives. The data placement in the distributed caches affect applications' performance significantly, so programmers need to carefully tune cache configurations such as the rescan interval and size (`HDFS-6106`).

Another common type of issues in cache is relevant to state maintenance for the cached objects. For example, NameNode uses a `retry cache` to prevent the failure of non-idempotent operations (e.g., create, append, and delete). As shown in patch (`HDFS-6229`), when a NameNode fails, one

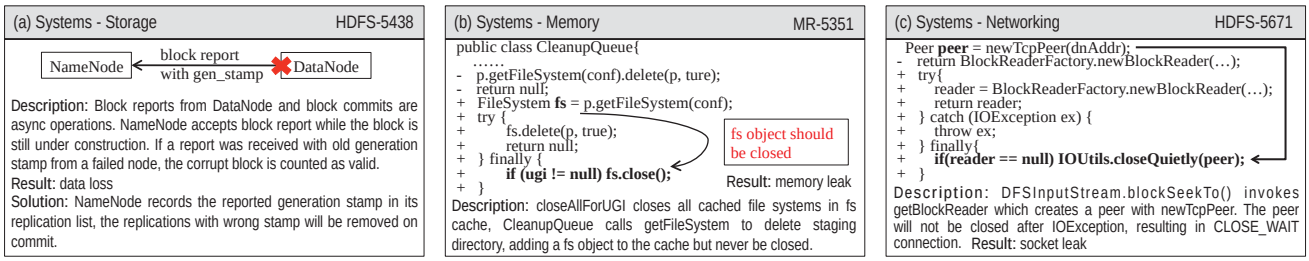


Figure 7. Real cases of issues in `systems` category.

stand-by node A in the cluster is set to become a new NameNode and starts building a complete retry cache. Before the retry cache on A is completed, another retry request, e.g., removing a block, may get served by A again and miss the retry cache. This can cause a race condition and wrong states of retry cache. To resolve it, a stronger lock or an explicit state is set on the cache so that non-idempotent operations are not allowed while building the retry cache.

5.2.5 Networking

Hadoop maintains large numbers of sockets and connections between mappers and reducers, and between NameNodes and DataNodes. Moreover, various networking protocols and topologies are supported to resolve compatibility issues across systems. We find that 204 issues in HDFS and 48 issues in MapReduce are relevant to `networking`.

We identify that approximately one quarter of `networking` *issues could cause resource wastage*. An example in Figure 7(c) demonstrates that an unclosed connection may cause a socket leak. In `blockSeekTo()` of `DFSInputStream`, getBlockReader is invoked to create a connection with the function`newTcpPeer`. However, if an `IOException` is triggered, the connection is not closed, resulting in a `CLOSE_WAIT` connection and memory wastage. Such issues could evolve into serious problems when large-scale network operations happen in data-intensive systems.

In Hadoop, networking information is used to instruct data block operations like data placement and load balance. *We observe that a number of* `networking` *issues involved in network management, such as network topology (e.g.,* `HDFS-5846`*), could lead to data loss*. This is caused by an incorrect networking location, e.g., rack id, being assigned to DataNodes, resulting in replicas of blocks on those nodes being placed in the same fault domain, such that a single failure can cause loss of two or more replicas.

5.2.6 Security

Differing from legacy distributed system, Hadoop has greater requirement for security because its applications may run in a shared cloud environment subjecting to malicious attacks. `Security` issues may involve exception handling, access control list (ACL), security policies, etc. These constitute 5.7% and 10.0% of the total system issues in HDFS and MapReduce, respectively.

Although security is always a big concern in cloud computing platforms, we did not observe specific efforts or techniques used in these open-source systems. Many security issues [2] can be fixed with existing techniques. `MR-5208` shows that a symlink attack may happen if the `ShuffleHandler` and `SpillRecord` access a map task's output and index files without using a secured I/O stream. A malicious user removes the task's log file and puts a link to the `jobToken` file of a target user. It then tries to open the `syslog` file via the servlet on the `tasktracker`, while `tasktracker` is unaware of the symlink, and blindly streams the content of the jobToken file. Thus, the malicious user can access potentially sensitive outputs of the target user's jobs. `SecureIOUtils` was implemented to fix such issues. More efforts are expected to secure the systems on both data (i.e., source, intermediate, output data) and job execution.

Summary: We observe that approximately a third of system issues are related to file systems; MapReduce suffers from more memory issues than HDFS; the majority of security issues can be solved with existing techniques used in conventional systems, and more efforts are expected for new security models for Big Data.

5.3 Programming Issues

The Java programming language facilitates programming with features like automatic object management. However, complex large-scale software still suffers from a significant number of `programming` issues, as shown in Figure 6(b).

5.3.1 Code Maintenance

Half of all programming issues are related to code maintenance (277 issues in HDFS, 269 issues in MapReduce), such as stale code and interfaces cleaning, data structure reorganization and optimization, appropriate fault handler implementation, etc. A well-known case is checking for `NULL` pointers before using them (`HDFS-6206`), which is often skipped by developers. Similar examples referring to code maintenance include `HDFS-5710`, `MR-5743`, etc. In the Hadoop system, these issues are marked as 'minor' issues by developers, but may require considerable effort to fix.

[2] e.g., HDFS-6556, HDFS-6462, HDFS-6411, HDFS-6435, MR-5809, MR-5770, MR-5475 and etc.

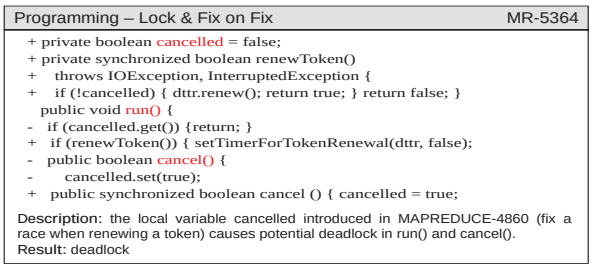


Figure 8. An example relevant to `locking`.

5.3.2 Interface

The second major group of issues is related to `interfaces` (147 issues in HDFS, 196 issues in MapReduce). In complex systems clean interfaces can dramatically reduce the possibility of producing unexpected errors and reduce the burden on programming and code maintenance. *We observe a number of issues were produced by inconsistency problems due to interface changes and inappropriate usage of interfaces*.

`Interface` issues are becoming more relevant as Big Data systems evolve rapidly to support new data analysis model and diverse data formats. The data sources could be structured data (e.g., database record, scientific data), unstructured data (e.g., plain text, webpages) and semi-structured data (e.g., movie reviewers) [43]. We find new interfaces like `DBInputformat` were implemented in new features and tasks to support database data formats (e.g., `MR-716`). However, supporting the diverse data formats requires more than just implementing new input/output interfaces, they also require additional efforts on performance tuning and data structure optimization [9, 26, 27].

5.3.3 Locking

A small fraction of issues relate to `locking` techniques. These issues (30 issues in HDFS, 18 issues in MapReduce) are normally caused by inappropriate usage of locks, synchronous or asynchronous methods, e.g., a concurrent `saveNamespace` executing as the expired token remover runs may result in NameNode deadlock (refer to `HDFS-4466`). Another case shown in Figure 8 illustrates an issue relevant to deadlock introduced by a fix on fix. In `MR-4860`, a variable `cancelled` was defined to fix a race condition when renewing tokens for jobs, however, it introduced a potential deadlock between the methods `cancel()` and `run()` in `RenewalTimerTask`.

5.3.4 Typo

Surprisingly, we observe 1.5-5.6% of `programming` *issues are caused by* `typos`. Many of them were marked as 'minor' issues. However, typos may not only cause wrong results and produce misleading outputs (e.g., `HDFS-4382`), but also result in failures (e.g., `HDFS-480`). For instance, a typo in the error information (i.e., '=' is missing in '$\geq$' condition) reported in the logs could mislead developers (`HDFS-5370`). Another serious case is `MR-5685`: a typo in the function name (`reduceContext.getCacheFiles()` is mistakenly spelled as `getCacheArchive()`) causes `getCacheFiles()` function returns `Null` in `WrappedReducer.java` file. Most of these typo issues cannot be easily detected (by compiler or spell checker) at an early stage, and they usually take enormous amount of effort for only "one-line" fix.

Summary: About half of the programming issues are related to code maintenance, including stale code and interface cleaning, data structure reorganization, and optimization. With the evolution of Big Data systems, interface issues are becoming increasingly prominent.

5.4 Issues in Tools

The subtypes `configuration`, `debugging`, `documentation (docs)`, and `test` are placed into the `Tools` category because they share the same purpose of improving the reliability and serviceability of the systems.

5.4.1 Configuration

As shown in Figure 6(c), we observe a significant fraction of issues in the `Tools` category to be related to `configuration` (166 issues in HDFS, 202 issues in MapReduce). Configuration issues have been investigated in other systems, such as CentOS, MySQL, Apache HTTP server, OpenLDAP, etc. [45, 46]. Similar `configuration` issues occur in the Hadoop ecosystem. For example, `MR-1249` shows the inconsistency issue between configuration and initial values in source code.

Beyond the configuration errors mentioned above, *a majority of configuration issues are related to system performance, caused by poorly tuned parameters*. For example, a patch was applied in `MR-353`, adding the parameters of shuffle and connection timeouts to the configuration file `mapred-default.xml`, because a huge performance difference was seen in terasort workload with the tuning of these parameters. With the evolution of systems, more configurable parameters are added to provide flexibility for users, e.g., there are about 170 configurable parameters in MapReduce's default configuration file `mapred-default.xml`, and a large fraction of them affect system performance.

5.4.2 Debugging

We observe that 107 issues in HDFS and 145 issues in MapReduce are related to `debugging`. These issues happen in logs, output delivered to users, error, fault, warning reports, etc. *Incorrect, incomplete, indistinct output may mislead users, hurting the system's serviceability*. To make matters worse, many existing bug-finding tools are based on system-generated logs (with the assumption that logs are correct), the error-prone logging system could decrease their effectiveness.

5.4.3 Documentation

Documentation is not taken seriously enough during Hadoop system development. *We identify that 1.94-3.35% of issues are related to* `docs` *issues*. These `docs` issues include typos,

inconsistency with latest versions of design and implementation, wrong information, lost docs, etc. For instance, the introduction of how to copy files with the `cp` command in the original snapshot documentation may mislead users (refer to `HDFS-6620`).

5.4.4 Test

We observe a number of issues happening in `test` with obvious symptoms of test failures. These issues are typically caused by incompatibilities (e.g., tests succeed on the Linux platform but fail on Windows) and inappropriate parameter configuration (e.g., timeout settings).

Evaluating and testing applied patches for large-scale distributed systems is important but difficult and challenging for three reasons. First, confirming every fix or applied patch in a real system can take a long time and require substantial computational resources on petascale data sets. Second, few developers have access to large-scale clusters configured with thousands of machines and hundreds of petabytes of storage. Third, the simulation of large-scale clusters like the MiniCluster of Hadoop could make testing results less accurate. A recent study proposed to use data compression techniques to enable large-scale tests on smaller-scale cluster, but it mainly targets scalability tests and does not work for failure tests [44]. Hadoop has implemented a fault injection platform but not all failure cases can be simulated. For example, with reference to `HDFS-6245`, the `IOError` cannot be simulated from `getCanonicalPath` using unit tests. An effective testing platform is an urgent need for the development of distributed data-intensive systems.

Summary: We observe a large number of configuration issues as investigated in previous studies. However, in data-intensive systems, more of these relate to performance tuning. The logging system is also error-prone, which increases the difficulty of detecting bugs for log-based approaches.

5.5 Discussion

Across the whole system stack, we identify a significant number of issues to be related to consistency and compatibility. We observe 134 issues in HDFS and 102 issues in MapReduce are related to `consistency`. They include the inconsistency between inode and data (e.g., `HDFS-4877`), the namespace and state inconsistency in jobtracker (e.g., `MR-5375`, `MR-5260`), etc. These consistency issues are difficult to identify using bug-finding tools because they are closely tied to function and component logic. For instance, accessing files via file paths in HDFS could cause namespace conflicts when a file being written is renamed and another file with the same name is created. This issue was fixed in `HDFS-6294` using a unique `inode ID` for each `inode` instead of relying on a file name.

We also observe a large number of issues (124 in HDFS, 287 in MapReduce) related to `compatibility`. These issues are mainly caused by use of new interfaces, features, and platforms (e.g., Hadoop on Windows, JDK version changes). For example, a number of `compatibility` issues were observed after YARN was released.

Consequence	Description
Build error	Codebase cannot be compiled or built successfully.
Failure	It includes corruption, crash, hang and job failures.
Runtime error	Faults/exceptions happen during program execution.
Test failure	Unit or failure tests fail.
Wrong	Incorrect output, unexpected workflow.
Potential error	No direct impact on system operations.

Table 5. Classification of consequences.

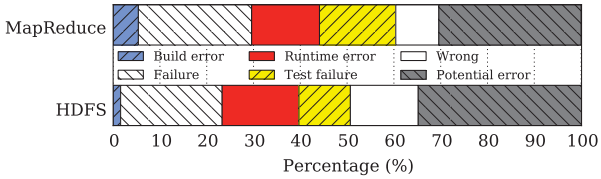


Figure 9. Consequences of sampled issues.

Types	Corruption	Crash	Hang	Job failures
HDFS	32 (1.5%)	47 (2.2%)	124 (5.7%)	268 (12.3%)
MapReduce	10 (0.5%)	46 (2.3%)	57 (2.8%)	379 (18.6%)

Table 6. General classification of failures. It illustrates the percentages of each failure type in total HDFS and MapReduce issues, respectively.

6. Consequences, Impact, and Reactions

We further analyze the severity of examined issues with a focus on the correlations between issue types and their consequences. We then investigate how the system reacts to handle the issues in Hadoop.

6.1 Issue Consequence

We classify consequences into six categories as shown in Table 5. We list the top three serious consequences `failure`, `runtime error`, and `wrong` in this paper.

Failure refers to `corruption`, `crash`, `hang`, and `job failures`. In general, we observe that 21.7% of HDFS issues and 24.2% of MapReduce issues cause failures directly. Specifically, we give the percentages of each subtype of failure in Table 6, showing the distribution of these issues among the total examined issues. A large number of issues cause job failures. They are usually easy to catch using exception reports or logs. For issues that cause the system to hang, many are caused by deadlocks (e.g., `MR-5364`, `HDFS-4466`) and inconsistent states (e.g., `MR-3460`). Debugging of these issues is hard due to limited logging information and because it requires knowledge of function logic. Many issues causing system crashes relate to resource management such as out of memory, reference to non-existent objects, etc. Furthermore, we observe more corruption cases

Impact	Description
Availability	Systems continue to work even when faults happen.
Functionality	The components or functions work correctly according to the specifications.
Performance	Systems operate with lower overhead and higher performance.
Reliability	The ability to consistently perform as expected.
Serviceability	The ease with systems that can be maintained, repaired and used.

Table 7. Classification of impact.

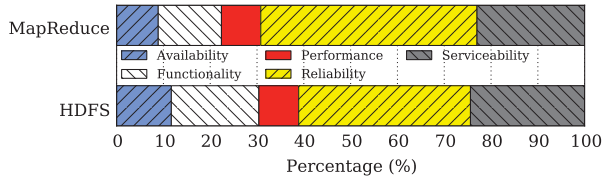


Figure 10. Impact of issues on systems.

in HDFS than in MapReduce. A large number of these issues were involved in block operations and data layout changes.

Runtime errors are distributed widely among the components of the systems. Many of them are caused by inappropriate usage of exceptions and bugs in fault handling code (see §5.4). Unlike the findings in [47], most of the fault handlers are implemented in open-source systems, but an uncovered exception case could trigger serious errors.

Wrong refers to any issues that produce incorrect output or execution in some unexpected path. These results are dangerous because they may mislead users or hide potential errors. Further, it is hard to detect these issues with normal bug-finding tools.

For issues relevant to `code maintenance` and `new features`, they may not cause direct system destruction at report time, but they still have the potential to trigger faults (`Potential Error`). We observe 30.4-34.9% of issues potentially cause errors. For example, the issues relevant to `code maintenance` were reported because the old interfaces may confuse developers or they are no longer called. With the evolution of systems, the codebase becomes more complex, uncleaned code could inhibit software development and incur unexpected errors.

6.2 Issue Impact

Figure 10 illustrates the distribution of issues among the six types (Table 7) of system impacts. *We observed system* `reliability` *is the most vulnerable in Hadoop.* The majority of these issues come from `bugs`. The second largest group of issues influences `serviceability`, mainly caused by issues in `tools` (§5.4). More specifically, we observed many of them to be related to `documentation` and `debugging`. The third largest group of issues have impact on `functionality`. To fix these issues, specific knowledge about component logic and workflows are required. For example, as reported in `HDFS-6680`, BlockPlacementPolicyDefault may choose favored datanodes incorrectly, hurting the fault tolerance of HDFS. Fixing this issue requires the understanding of how datanodes are allocated for blocks.

We observe a large fraction of `availability` issues were triggered in fault handling methods. These methods may have programming issues, inappropriate usage of exceptions, etc. Finally, we observe 8.5% of issues relate to system `performance`. Most of these issues (e.g., `HDFS-6191`, `HDFS-4721`, `MR-5368`, `MR-463`) belong to the types of `improvement` and `new feature`.

6.3 Reactions to Issues

We further examine how systems take actions to handle issues before they are detected by users. The study of the system reactions could help us locate the causes of issues from a different viewpoint. We identified four reaction methods commonly used in Hadoop: `exception handling`, `retrying`, `silent reaction`, `restart and recovery`.

6.3.1 Exception Handling

Programmers usually use exceptions to catch signals that errors or exceptional conditions have occurred, and they implement the corresponding fault handlers to handle caught exceptions. We observe a significant fraction of the Hadoop codes to use `try-catch-finally` exception handling. However, *exception handling itself is error-prone according to our study*. Some Java compilers force the programmers to use exception handling when they call specific functions such as `socket creation`, `open file`, and so on. But compiler checking cannot avoid the situation in which the fault handlers are not implemented appropriately. Furthermore, the inappropriate and inaccurate usage of exception types and throwing (e.g., `HDFS-4388`) further weaken the functionalities of exception handling.

6.3.2 Retrying

Retrying is another mechanism used frequently in distributed systems. When an exceptional fault occurs, the corresponding components will re-execute programs with defined retry policies. Hadoop implements three retry policies: `RETRY_FOREVER` (keep trying forever), `TRY_ONCE_DONE_FAIL` (fail silently, or re-throw the exception after trying once) and `TRY_ONCE_THEN_FAIL` (re-throw the exception after trying once). The `retry` method can overcome some transient errors. For instance, as the networking connection may terminate for a short period, retrying the connection within the defined timeout could rebuild the connection. However, *when the* `retry` *method does not succeed, it can result in system hangs or failures*. We observed that 1.1-2.3% of issues with such consequences were reported.

6.3.3 Silent Reaction

Systems can continue to work after silent reactions to some 'trivial' or 'minor' issues. However, this may hide potential

issues and transient errors. It also increases the complexity of bug detection, and *their evolution could result in severe problems like data loss and service unavailability [14]*. We find 0.6% of issues are caused by silent reactions, including ignorance of return values, skipped sanity checking, etc.

6.3.4 Restart and Recovery

System restart and recovery are the "last line of defense" to handle failures. In Hadoop, metadata (e.g., copies of `FSImage` and `edit log`) and data blocks are replicated to several machines. If hardware errors or file corruption happens, replicated copies are used to recover the system. However, *maintaining a consistent view of replicated copies and snapshots is hard*. We observe that 3.7% of the total issues are involved in the procedure of checkpointing.

Summary: Exception handling is widely used to catch software errors, but unfortunately the fault handler itself is error-prone. Retry methods can overcome transient faults but can cause serious consequences if transient faults are turned into non-transient faults.

7. Related Work

Bug and patch analysis: A number of characteristics studies on bugs and patches in various systems have been conducted. Rabkin et al. [34] examined a sample of 293 customer support cases happened during the period of spring and summer 2011 in Cloudera's CDH3 Hadoop distribution. Gunawi et al. [20] studied 3655 'major' issues in cloud systems over a period of three years (1/1/2011–1/1/2014). We share the purpose with these studies, but offer a unique focus on internal and external correlations of issues. We also reveal the unique bug patterns and its correlations with characteristics of various Hadoop aspects.

In addition, our work is different from other prior studies in three ways. First, we conduct our study based on a larger set of issues, covering 4218 issues over a longer period of six years. Second, all examined issues have been solved with applied patches, and have been proven to operate correctly, which guarantees that every issue examined in our study is valid. Third, we not only study 'major' issues, but also cover 'minor' issues, as we observe that these 'minor' issues also cause serious consequences.

Recent work has investigated bug patterns [30] and error management [37] in conventional Linux file systems, misconfiguration errors in commercial storage systems [46], operating system errors in Linux kernels [11], concurrency bugs in MySQL [16] and other sever applications [31]. All of these studies addressing conventional software and applications have motivations similar to ours: to learn from mistakes and experience. We focus on state-of-the-art distributed data-intensive systems.

Bug-finding tools: Detecting and finding bugs in distributed systems is challenging [29, 35]. Yuan et al. [47] proposed a static checker for testing error handling code, based on their finding that the majority of serious failures are caused by error handling (e.g., fault handler is not implemented) in distributed systems. Unlike their findings, we observe that many issues occurring in error handling code are caused by the inappropriate usage of exceptions and by incorrect logic in fault handler implementations. Gunawi et al. [19] proposed a testing framework to tackle recovery problems for cloud systems. Reynolds et al. [35] proposed an infrastructure to expose structural errors and performance problems by comparing actual behavior and expected behavior in distributed systems. Geels et al. [18] proposed to log the execution of distributed applications, and then replay the logs to reproduce non-deterministic failures. Sambasivan et al. [38] discussed end-to-end tracing methods for distributed system. Our study complements these works with insights on issue characteristics and their intrinsic correlations, and offers useful hints and findings to assist in the development of bug-finding tools for distributed data-intensive systems.

Reliable distributed systems: Recent work has proposed to build more reliable distributed systems against failures and unexpected errors. Although distributed systems can leverage fault tolerance techniques like Byzantine Fault Tolerance [36] to protect systems from unexpected behaviors like data corruption, they still suffer from software bugs and cannot work properly if their implementations have issues [14]. Do et al. [14] built an enhanced version of HDFS named HARDFS to protect it against fail-silent bugs. Fryer et al. [17] presented a file system with specific metadata interpretation and invariant checking to protect file system metadata against buggy file system operations. Our work would be helpful for building reliable distributed systems based on the revealed issue patterns in such systems.

8. Conclusion

This paper presents a comprehensive and in-depth study of 4218 solved issues in the two representative, widely used storage (HDFS) and computation (MapReduce) platforms designed for Big Data applications. The study covers the reported and solved issues over the last six years with the purpose of summing up learned lessons and experiences for the development of distributed data-intensive systems. It examines these issues from several dimensions, including types, correlations, consequences, impacts, and reactions. These finding should help the development, deployment, and maintenance of existing and future distributed data-intensive systems and their associated bug-finding and debugging tools.

Acknowledgments

We would like to thank our shepherd Sumita Barahmand as well as anonymous reviewers. We thank Taesoo Kim and Kei Davis for their feedback. We also thank Xin Chen for his initial support on collecting MapReduce issues. This work was supported in part by the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] Apache Cascading. http://www.cascading.org/.

[2] Apache Flume. http://flume.apache.org/.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] Apache HBase. http://hbase.apache.org/.

[5] Apache HCatalog. https://cwiki.apache.org/confluence/display/Hive/HCatalog.

[6] Apache Hive. https://hive.apache.org/.

[7] Apache Mahout. https://mahout.apache.org/.

[8] Apache Pig. http://pig.apache.org/.

[9] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *SC'11*, Seattle, WA, Nov. 2011.

[10] Centralized Cache Management in HDFS. https://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *SOSP'01*, Oct. 2001.

[12] Contributors to Hadoop. http://blog.cloudera.com/blog/2011/10/the-community-effect/.

[13] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu. HiTune: Dataflow-Based Performance Analysis for Big Data Cloud. In *USENIX ATC'11*, 2011.

[14] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *FAST'13*, San Jose, CA, Feb. 2013.

[15] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *SOSP'11*, Cascais, Portugal, Oct. 2011.

[16] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A Study of the Internal and External Effects of Concurrency Bugs. In *DSN'10*.

[17] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying File System Consistency at Runtime. In *FAST'12*, San Jose, CA, Feb. 2012.

[18] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *USENIX ATC'06*, May 2006.

[19] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESIGN: A Framework for Cloud Recovery Testing. In *NSDI'11*, Boston, MA, Mar. 2011.

[20] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SOCC'14*, Nov. 2014.

[21] Hadoop at Twitter. https://blog.twitter.com/2010/hadoop-twitter.

[22] Hadoop Distributed File System. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[23] Hadoop MapReduce. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[24] Hadoop Systems. http://hadoop.apache.org/.

[25] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *FAST'14*.

[26] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE'11*, Apr. 2011.

[27] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major Technical Advancements in Apache Hive. In *SIGMOD'14*.

[28] J. Huang, X. Ouyang, J. Jose, M. W. Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over Infiniband. In *IPDPS'12*.

[29] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou. SETSUDO: Perturbation-based Testing Framework for Scalable Distributed Systems. In *TRIOS'13*.

[30] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A Study of Linux File System Evolution. In *FAST'13*, Feb. 2013.

[31] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS'08*, Seattle, WA, Mar. 2008.

[32] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. In *SIGMOD'13*, New York, USA, June 2013.

[33] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *OSDI'14*, Broomfield, CO, Oct. 2014.

[34] A. Rabkin and R. H. Katz. How Hadoop Clusters Break. *IEEE Software*, pages 88–94, 2013.

[35] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI'06*, San Jose, CA, May 2006.

[36] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *SOSP'01*, Banff, Canada, Oct. 2001.

[37] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[38] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger. So, you want to trace your distributed system? Key design insights from years of practical experience. *Technical Report, CMU-PDL-14-102*, 2014.

[39] A. Silberstein, R. Sears, W. Zhou, and B. F. Cooper. A Batch of PNUTS: Experiences Connecting Cloud Batch and Serving Systems. In *SIGMOD'11*, Athens, Greece, June 2011.

[40] M. Tatineni. Hadoop for Scientific Computing. *SDSC Summer Institute: HPC Meets Big Data*, 2014.

[41] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[42] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *Middleware'12*, Montreal, Quebec, Canada, Dec. 2012.

[43] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *HPCA'14*, Flordia, USA, Feb. 2014.

[44] Y. Wang, M. Kapritsos, L. Schmidt, L. Alvisi, and M. Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI'14*, Seattle, WA, Apr. 2014.

[45] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do Not Blame Users for Misconfigurations. In *SOSP'13*, Farmington, Pennsylvania, Nov. 2013.

[46] 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'11*.

[47] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *OSDI'14*, Broomfield, CO, Oct. 2014.

[48] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *OSDI'12*, Hollywood, CA, Oct. 2012.

[49] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *ASPLOS'11*, Newport Beach, California, Mar. 2011.