Chapter 11
Fault Tolerance in MapReduce: A Survey

Bunjamin Memishi, Shadi Ibrahim, María S. Pérez and Gabriel Antoniu

11.1 Introduction

Data-intensive computing has become one of the most popular forms of parallel computing. This is due to the explosion of digital data we are living. This data expansion has mainly come from three sources: (i) scientific experiments from fields such as astronomy, particle physics, or genomics; (ii) data from sensors; and (iii) citizens publications in channels such as social networks.

Data-intensive computing systems, such as Hadoop MapReduce, have as main goal the processing of an enormous amount of data in a short time, by transmitting the computation where the data resides. In failure-free scenarios, these frameworks usually achieve good results. Given that failures are common at large scale, these frameworks exhibit some fault tolerance and dependability techniques as built-in features. In particular, MapReduce frameworks tolerate machine failures (crash failures) by re-executing all the tasks of the failed machine by the virtue of data replication. Furthermore, in order to mask temporary failures caused by network or machine overload (timing failure) where some tasks are performing relatively slower than other tasks, Hadoop relaunches other copies of these tasks on other machines.

B. Memishi (✉) · M.S. Pérez
OEG, E.T.S. Ingenieros Informáticos, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
e-mail: bmemishi@fi.upm.es

M.S. Pérez
e-mail: mperez@fi.upm.es

S. Ibrahim · G. Antoniu
Inria Campus Universitaire de Beaulieu, Rennes, 35042 Brittany, France
e-mail: shadi.ibrahim@inria.fr

G. Antoniu
e-mail: gabriel.antoniu@inria.fr

F. Pop et al. (eds.), *Resource Management for Big Data Platforms*,
Computer Communications and Networks, DOI 10.1007/978-3-319-44881-7_11

Foreseeing MapReduce usage in the next generation Internet [46], a particular concern is the aim of improving the MapReduce's reliability by providing better fault-tolerant mechanisms. As far as we know, there is not a complete review of the research in MapReduce fault tolerance, in such a way that it represents an overall picture of what has been done and what is missing. This survey addresses this gap, by means of the following contributions:

- An exhaustive study on the MapReduce framework, and its default fault-tolerant mechanisms. As one may assume, the leading MapReduce-based systems which are open source, with particular emphasis on Hadoop MapReduce, have evolved significantly since their first appearance. This study has taken into account all of these releases, by considering the most important advances in the fault-tolerant area.
- A systematic literature review on contributions with extensive analysis and proposals of new fault-tolerant mechanisms in MapReduce-based systems.
- A discussion about the open issues and key challenges for providing efficient fault tolerance in MapReduce-based systems.

This book chapter is organized as follows. Section 11.2 introduces the methodology which has guided the survey. Section 11.3 represents the MapReduce fundamentals, with particular emphasis on its fault-tolerant mechanisms. Section 11.4 gives an extensive analysis of the literature review. Section 11.5 describes some of the most popular data-intensive computing systems, mentioning their fault tolerance mechanisms. Section 11.6 discusses the opportunities and challenges to design efficient fault-tolerant mechanisms in MapReduce. Finally, Sect. 11.7 concludes the book chapter.

11.2 Methodology

In order to analyze the existing work, a previous identification of the main areas related to the topic addressed in this survey was performed. Concretely, four groups were defined:

- *Context* Contributions related to the general context of dependability.
- *MapReduce* Contributions that introduce the fundamentals of the MapReduce framework.
- *Optimizations* Contributions related to direct fault-tolerant solutions on MapReduce-based systems.
- *Others* Contributions proposing different solutions for other systems, which could be considered as added values in the context of MapReduce-based systems.

Additionally, this study has made a general analysis of the different types of failures in computing systems, and their connection to the MapReduce framework. This has enabled a solid construction of boundaries between different failures types

within MapReduce and other distributed systems. In the computer science literature, it is also very common to have a distinction between faults, errors, and failures [7], considering faults and errors as implications of failures. According to the existing literature [8, 11, 50], the most common failure-type division in MapReduce is: crash, omission, arbitrary, network and security failures. More specifically:

- *Crash failure* The process crashes at a specific point of time and never recovers after that time.
- *Omission failure* It is a more general kind of failures. A process does not send (or receive) a message that it is supposed to send (or receive).
- *Arbitrary (Byzantine) failure* The process fails in an arbitrary manner if it can deviate in any conceivable way from the algorithm assigned to it. It is the most expensive failure to tolerate. Even though it is assumed mostly as intentional and malicious, the arbitrary failure can simply be a bug in the implementation, the programming language, or even the compiler.
- *Network failure* The processes cannot communicate with each other. There are two kind of failures: (i) *One-way link.* There is difficulty in communication between two processes (e.g., one communication party can send, but the other party cannot receive); and (ii) *Network partition.* A line connecting two larger sections of a network fails.
- *Security failure* The messages between processes are inspected, modified, or prevented from being delivered. This group also considers *eavesdropping failures*, which are those failures related to leaking information obtained in an algorithm to an outside entity, possibly threatening the confidentiality of the data handled by the algorithm.

Section 11.4 will further clarify these concepts within MapReduce-based systems.

11.3 MapReduce Framework

The MapReduce framework is one of the most widespread approaches of data-intensive computing. It represents a programming model for processing large datasets [19, 33]. MapReduce has been discussed by researchers for more than a decade, including the database community. Even though its benefits have been questioned when compared to parallel databases, some authors suggest that both approaches have their own advantages, and there is not a risk that one could become *obsolete* [54]. MapReduce's advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs. Other advantages are simplicity, automatic parallelism, and scalability. These features make MapReduce an appropriate option for data-intensive applications, being more and more popular in this context. Indeed, it is used for different large-scale computing environments, such as Facebook Inc. [23], Yahoo! Inc. [65], and Microsoft Corporation [45].

By default, every MapReduce execution needs a special node, called *master*; the other nodes are called *workers*. The master keeps several data structures, like

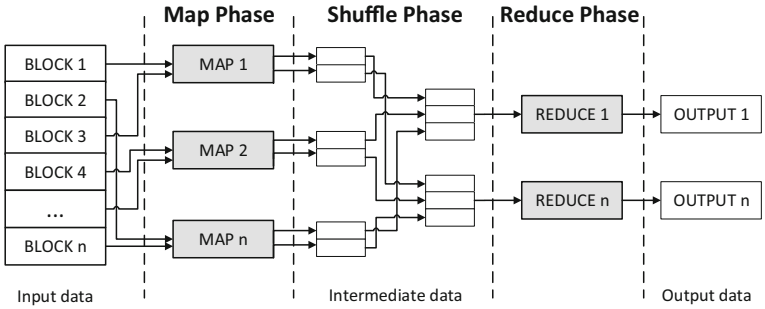

Fig. 11.1 MapReduce logical workflow

the state and the identity of the worker machines. Different tasks are assigned to the worker nodes by the master. Depending on the phase, tasks may execute two different functions: Map or Reduce. As explained in [19], users have to specify a Map function that processes a $key/value$ pair to generate a set of intermediate $key/value$ pairs, and a Reduce function that merges all intermediate values associated with the same intermediate key. In this way, many real-world problems can be expressed by means of the MapReduce model.

A simple MapReduce data workflow is shown in Fig. 11.1. This figure represents a MapReduce workflow scenario, from the input data to the output data. The most common implementations keep the input and output data in a reliable distributed file system, while the intermediate data is kept in the local file system at the worker nodes.

11.3.1 MapReduce 1.0 Versus MapReduce 2.0

The most common implementation of MapReduce is part of the Apache Hadoop open-source framework [56]. Hadoop uses the Hadoop Distributed File System (HDFS) as the underlying storage backend, but it was designed to work on many other distributed file systems as well.

The main components of Apache Hadoop are MapReduce and HDFS. Hadoop MapReduce consists of a JobTracker and many TaskTrackers, which constitute the processing master and workers, respectively. TaskTrackers consist of a limited number of slots for running map or reduce tasks. The MapReduce workflow is managed by the JobTracker, whose responsibility goes beyond the MapReduce process. For instance, the JobTracker is also in charge of the resource management. HDFS consists of a NameNode and many DataNodes, that is, the storage master and workers, respectively, whereas the NameNode manages the file system metadata, DataNodes hold a portion of data in blocks.

The traditional version of Hadoop has faced several shortcomings on large-scale systems, concerning scalability, reliability, and availability. The YARN (*Yet Another Resource Negotiator*) project has recently been developed with the aim of addressing these problems [57].

In the classic version of Hadoop, the JobTracker handles both resource management and job scheduling. The key idea behind YARN is to separate concerns, by splitting up the major functionalities of the JobTracker, resource management, and job scheduling/monitoring, into separate entities. In the new architecture, there is a global ResourceManager (RM) and per-application ApplicationMaster (AM). The ResourceManager and a per-node slave, the NodeManager (NM) compose the data-computation framework. The per-application ApplicationMaster is in charge of negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the progress of the tasks. The ResourceManager includes two components: a scheduler and application manager. Whereas the scheduler is in charge of resource allocation, the application manager accepts job submissions, and initiates the first job container for the job master (ApplicationMaster). This architectural change has as main goals to provide scalability and remove the single point of failure presented by the JobTracker. However, the resource scheduler, the application manager, and the application master now become single points of failure in the YARN architecture.

11.3.2 MapReduce Fault Tolerance

In Fig. 11.2 we show a big picture of the default fault-tolerant concepts and their mechanisms in MapReduce.

At the core of failure detection mechanism is the concept of heartbeat. Any kind of failure that is detected in MapReduce has to fulfill some preconditions, in this case to miss a certain number of heartbeats, so that the other entities in the system detect the failure. The classic implementation of MapReduce has no mechanism for dealing with the failure of the master, since the heartbeat mechanism is not used to detect this kind of failure. Workers send a heartbeat to the master, but the master's

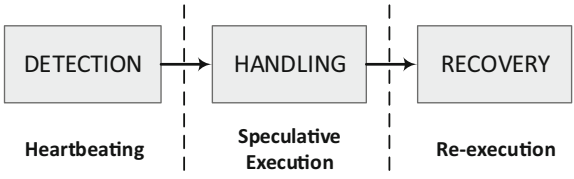

Fig. 11.2 Fault tolerance in MapReduce: The basic fault tolerance definitions (detection, handling, and recovery) with their corresponding implementations

health is monitored by the cluster administrator. This person must first detect this situation, and then manually restart the master.

Because the worker sends heartbeats to the master, its eventual failure will stop this notification mechanism. From the worker side, there is a simple loop that periodically sends heartbeat method calls to the master; by default, this period has been adjusted to 3 s in most of the implementations. The master makes a checkpoint every 200 s, in order to detect if it has missed any heartbeats from a worker for a period of 600 s, that is, 10 min. If this condition is fulfilled, then a worker is declared as dead and removed from the master's pool of workers upon which can schedule tasks on. After the master declares the worker as dead, the tasks running on a failed worker are restarted on other workers. Since the map tasks that completed its work, kept their output on the dead worker, they have to be restarted as well. On the other hand, reduce tasks that were not completed need to be executed in different workers, but since completed reduce tasks saved its output in HDFS, their re-execution is not necessary.

Apart from telling to the master that a worker is alive, heartbeats also are used as a channel for messages. As a part of the heartbeat, a worker states whether it is ready to run a new task, and in affirmative case, the master will use the heartbeat return value for communicating the actual task to the worker. Additionally, if a worker notices that it did not receive a progress update for a task in a period of time (by default, 600 s), it proceeds to mark the task as failed. After this, the worker's duty is to notify the master that a task attempt has failed; with this, the master reschedules a different execution of the task, trying to avoid rescheduling the task on the same worker where it has previously failed.

The master's duty is to manage both, the completed and ongoing tasks on the worker to be re-executed or speculated, respectively. In the case of a worker failure, before the master decides to re-execute the completed and ongoing tasks so that may skip the default timeout of MapReduce (10 min), there is only one opportunity left, speculative execution.

The speculative execution is meant to be a method of launching another equivalent task as a backup, but only after all the normal tasks have been launched, and after the average running time of the other tasks. In other words, a speculative task is basically run for map and reduce tasks that have its completion rate below a certain percentage of the completion rate of the majority of running tasks.

An interesting dilemma is how to differentiate the handling and recovery mechanisms. A simple question arises: Does MapReduce differentiate between handling and recovery?

In some sense, both, speculative execution and re-execution try to complete a MapReduce job as soon as possible, with the least processing time, while executing its tasks on minimal resources (e.g., to avoid long occupation of resources by some tasks).[1] However, the sequence of performing the speculative execution and re-execution is what makes them different, therefore considering the former one be

[1]This is not particularly true in the case of speculative execution, since it has proven to exhaust a considerable amount of resources, when executed on heterogeneous environments [39, 72] or when the system is going under failures [22].

part of the handling process, and the latter one part of the recovery. An additional difference to this is that, while the re-execution mechanism tries to react after the heartbeat mechanism has declared that an entity has failed, the speculative execution does not need the same timeout condition in order to take place; it reacts sooner.

Regarding to the nomenclature related to failures and errors, we consider a job failure when the job does not complete successfully. In this case, the first task that fails can be considered as an error, because it will request its speculation or re-execution from the master. A task failure can happen because the network is overloaded (in this case, this is also an error, because the network fault is active and loses some deliveries). In order to simplify this, we assume that in MapReduce, a task or any other entity is facing a failure, whenever it does not fulfill its intended function.

From the point of view of Hadoop's MapReduce, failures can happen in the master and worker. When the master fails, this is a single point of failure. But in the case of the worker, it may have a task fail (map or reduce task, or shuffle phase) or the entire worker. During a map phase, if a map task crashes, Hadoop tries to recompute it in a different tasktracker. In order to make sure that this computation takes place, most of the reducers should complain for not receiving the map task output or the number of notifications is higher or equal to three [51]. The failed tasks have higher priority to be executed than the other ones; this is done to detect when a task fails repeatedly due to a bug and stop the job. In a reduce phase, a reduce task failure will have to be executed in a different tasktracker, having in mind that the three reduce phases should start from the beginning. The reduce task is considered as failed, if the majority of its shuffle attempts fails, the shuffle phase does not succeed to get five map outputs, or its progress stops for a long time. During the shuffle phase, a failure may also happen (in this case, a network failure), because two processes (in our case two daemons) can be in a working state, but a network failure may stop any data interchange between them. MapReduce implementations have been improved by means of the Kerberos authentication system, preventing a malicious reduce task from requesting another user's map output.

11.4 Analysis

Several projects have addressed different reliability issues in data-intensive frameworks, in particular for MapReduce. In this section, we have tried to collect those studies, adapting them to the most common failure type divisions in distributed systems [8, 11, 50]: crash, omission, arbitrary, network and security failures. Table 11.1 summarizes the main studies included in this review, listed in publication date order.

In [22], authors have evaluated Hadoop, demonstrating a large variation in Hadoop job completion time in the presence of failures. According to authors, this is because Hadoop uses the same functionality to recover from worker failure, regardless of the cause or failure type. Since Hadoop couples failure detection and recovery with overload handling into a conservative design with conservative parameter choices, it is often slow reacting to failures, exhibiting different response times under failure.

Table 11.1 The main studies included in this review, listed in publication date order

Year and study	Algorithms	Concepts
2004, [19]	Speculative execution	The foundations
2008, [72]	Longest Approximate Time to End (LATE)	Heterogeneity considerations
2009, [15]	UpRight library ($3f + 1$)	Byzantine fault tolerance
2009, [59]	Metadata replication	Crash, failure handling
2009, [10]	Re-execution evaluation	Crash, omission failure, failure recovery
2009, [58]	MapReduce setup simulation	Crash, network failure
2010, [39]	Dedicated compute resources, Hibernate state, Hybrid task scheduling	Volunteer computing systems
2010, [4]	Outlier culling, Cause- and resource-aware	Crash, omission failure, failure detection, failure handling, failure recovery, Data locality, Network hotspot
2010, [37]	Intermediate storage system—ISS (through asynchronous replication, rack-level replication, selective replication)	Crash, omission failure, failure handling, failure recovery
2010, [16]	Pipelined intermediate data, Online aggregation, Continuous queries	MapReduce and beyond (e.g., Interactive applications), Crash, omission failure, failure handling, failure recovery
2010, [48]	BlackBox approach (based on OS-level metrics)	Crash, omission failure, failure detection and diagnosis
2010, [55]	Massive fault tolerance, replica management, barriers free execution, latency-hiding optimization, distributed result checking	Network failure, failure handling, failure recovery
2010, [14]	Hybrid calculation model ($n - step$ probability, VM expected lifetime, cost of termination)	Network, failure handling, failure recovery
2010, [51]	Mandatory access control, differential privacy	Security failure, failure detection, failure handling
2011, [41]	Map and shuffle (phases) overlap, Task duplication, pull mechanism, queues	Cloud-based MapReduce, Crash, single point of failure, failure detection, failure recovery
2011, [73]	Adaptive interval, reputation-based detector	Crash, failure detection
2011, [9]	Metadata replication	Crash, failure handling
2011, [34]	Stochastic prediction model	MapReduce fault tolerance understanding
2011, [18]	Deferred execution, Tentative reduce execution, Digest outputs, Tight storage replication	Crash, and Byzantine fault tolerance, failure handling

(continued)

Table 11.1 (continued)

Year and study	Algorithms	Concepts
2011, [42]	Split message format modification, save the intermediate work, commit mechanism modification	Network, failure handling, failure recovery
2012, [52]	Kerberos protocol	Security, failure detection, failure handling
2012, [24]	JobTracker re-architecture	Crash, single point of failure, failure handling, failure recovery
2013, [57]	JobTracker re-architecture through Hadoop YARN	Crash, single point of failure, failure handling, failure recovery
2013, [29]	Encryption	Eavesdropping failure, failure detection, failure handling
2013, [35]	Analytical failure measurement	Crash, failure handling, failure recovery
2014, [3]	Greedy Speculative Scheduling (GS), Resource Aware Speculative Scheduling (RAS)	Approximation analytics
2014, [13]	Maximum Cost Performance (MCP)	Weighted moving average (EWMA)
2014, [62]	Accountability test	Byzantine fault tolerance, failure detection, failure handling
2014, [64]	Early cloning, Enhanced Speculative Execution (ESE)	Single job
2015, [63]	Smart Cloning Algorithm (SCA), Enhanced Speculative Execution (ESE)	Multiple jobs
2015, [44]	Diarchy algorithm	Single point of failure, failure handling
2015, [30]	Failure recovery evaluation	Crash failure, multiple jobs
2015, [66]	Failure-aware scheduling	Failure handling, failure recovery, multiple jobs
2015, [49]	Energy cost of speculation	Omission failure, resource heterogeneity, energy considerations
2016, [43]	MapReduce timeout analysis	Failure detection, handling

Authors conclude that Hadoop makes unrealistic assumptions about task progress rates, rediscovers failures individually by each task at the cost of great degradation in job running time, and does not consider the causes of connection failures between tasks, which leads to failure propagation to healthy tasks.

In [43], authors report that, in the presence of single machine failure the applications latencies vary not only in accordance to the occupancy time of the failure, similar to [22], but also vary with the job length (short or long).

In [10], authors have evaluated the performance and overhead of both the checkpointing-based fault tolerance and the re-execution based fault tolerance in MapReduce through event simulation driven by Los Alamos National Labs (LANL)

data. Regarding MapReduce, the fault tolerance mechanism which was explored is re-execution, where all map or reduce tasks from a failed core are reallocated dynamically to operational cores whether the tasks had completed or not (i.e., partial results are locally stored), and execution is repeated completely. In the evaluation of the performance of MapReduce in the context of real-world failure data, it was identified that there is pressure to decrease the size of individual map tasks as the cluster size increases.

In [35], authors have introduced an analytical study of MapReduce performance under failures, comparing it to MPI. This research is HPC oriented and proposes an analytical approach to measure the capabilities of the two programming models to tolerate failures. In the MapReduce case, they have started with the principle that any kind of failure is isolated in one process only (e.g., map task). Due to this, the performance modeling of MapReduce was built on the analysis of each single process. The model consists of introducing an upper bound of the MapReduce execution time when no migration/replica is utilized, followed by an algorithm to derive the best performance when replica-based balance is adopted. According to the evaluation results, MapReduce achieves better performance than MPI on less reliable commodity systems.

11.4.1 Crash Failure

During a crash failure, the process crashes at a specific point of time and never recovers after that time. Since a crash failure involves process failing to finish its function according to its general definition, this means that in MapReduce a crash failure can lead to a node (machine), daemon (JobTracker or TaskTracker), or task (map, reduce) failure. These failures surge when a node simply crashes, and affects all of its daemons. But this is not only the case for a crash failure; there are many other cases when particular daemons or tasks crash due to Java Virtual Machine (JVM) issues, high overloads, memory or CPU errors, etc.

At this section point, we will summarize the master crash failures first, and then continue with other crash failures in MapReduce. An important contribution to the high availability of JobTracker is the work of Wang et al. [59]. Their paper proposes a metadata replication-based solution to enable Hadoop high availability by removing single point of failure in Hadoop, regardless of whether it is NameNode or a JobTracker. Their solution involves three major phases:

- Initialization phase. Each standby/slave node is registered to active/primary node and its initial metadata (such as version file and file system image) are caught up with those of active/primary node.
- Replication phase. The runtime metadata (such as outstanding operations and lease states) for failover in future are replicated.
- Failover phase. Standby/new elected primary node takes over all communications.

A well-known implementation of this contribution has been done by Facebook, Inc. [9], by creating the active and standby AvatarNode. This node is simply wrapped

to the NameNode, and the standby AvatarNode takes the role of the active AvatarNode in less than a minute; this is because every DataNode speaks with both AvatarNodes all the time.

However, the above solution did not prove to give the optimum for the company requirements, since their database has grown by 2500× in the past 4 years. Therefore, another approach named Corona [24] was used. This time, for Facebook researchers it was obvious that they should separate the JobTracker responsibilities: resource management and job coordination. The cluster manager should look for cluster resources only, while a dedicated JobTracker is created per each job. As you can notice, at many points, the design decisions of Corona are similar to Hadoop YARN. Additionally, Corona has been designed to use push-based scheduling, as a major difference to the pull-based scheduling of the Hadoop MapReduce.

In the work [47], authors propose an automatic failover solution for the JobTracker to address the single point of failure. It is based on the Leader Election Framework [11], using Apache Zookeper [5]. This means that multiple JobTrackers (at least three) are started together, but only one of them is the leader at a particular time. The leader does not serve any client, but receives periodical checkpoints from the remaining JobTrackers. If one of the NameNodes fails, the leader recovers its availability from the most recent checkpointed data. However, this solution within Yarn has not been explored for job masters [57] and only addresses other single points of failure, such as the resource manager daemon.

In a recent study [44], the authors propose Diarchy, a novel approach for management of masters, whose aim is to increase the reliability of Hadoop YARN, based on the sharing and backup of responsibilities between two masters working as peers. Despite the fact that Diarchy seems only to improve the reliability of failure handling between masters, its functioning also puts a lower boundary in the worst-case assumption, with the number of Diarchy failed tasks not surpassing the half number of failed tasks of Hadoop YARN.

In case of a TaskTracker crash failure, its tasks are by default re-executed in the other TaskTrackers. This is valid for both, map and reduce tasks. Map tasks completed on the dead TaskTracker are restarted because the job is still in the progress phase and did not finish yet, and contains n number of reduce tasks, which need that particular map output. Reduce tasks are re-executed as well, except for those reduce tasks that have completed, because they have saved its output in a distributed file system, that is, in HDFS.

MapReduce philosophy is based on the fact that a TaskTracker failure does not represent a drastic damage to the overall job completion, especially long jobs. This is motivated by large companies [20], which use MapReduce on a daily basis, and argue that even with a loss of a big number of machines, they have finished in a moderate completion time.[2] Any failure would simply speculate/re-execute the task in a different TaskTracker.

[2]Jeff Dean, one of the leading engineers in Google, said: (we) "lost 1600 of 1800 machines once, but finished fine".

There are cases where TaskTrackers may be blacklisted by mistake from the JobTracker. In fact, this happens because the ratio of the number of the failed tasks in the respective TaskTracker is higher than the average failure rate on the overall cluster [61]. By default, the Hadoop's blacklist mechanism marks a TaskTracker as blacklisted if the number of tasks that have failed is more than four. After this, the JobTracker will stop assigning future tasks to that TaskTracker for a limited period of time. These blacklisted TaskTrackers can be brought to live, only by restarting them; in this way, they will be removed from the JobTracker's blacklist. The blacklisting issue could also go beyond this. This can be explained with one scenario. Let us assume that, at some point, reduce tasks that are running in the other TaskTrackers will try to connect to the failed TaskTracker. Some of the reduce tasks need the map output from the failed TaskTracker. However, as they cannot terminate the shuffle phase (because of the missing map output from the failed TaskTracker), they fail. Experiments in [22] show that reduce tasks die within seconds of their start (without having sent notifications) because all the conditions which declare the reduce task to be faulty become temporarily true when the failed node is chosen among the first nodes to connect to. In these cases, when most of the shuffles fail and there is little progress made, there is nothing left except re-execution, while wasting an additional amount of resources.

The idea behind the paper [18] is doubling each task in execution. This means that if one of the tasks fails, the second backup task will finish on time, reducing the job completion time using larger (intuitively, you may guess that doubling the tasks leads to approximately doubling the resources) amounts of resources.

In [73], authors have proposed two mechanisms to improve the failure detection in Hadoop via heartbeat, but only in the worker side, that is, the TaskTracker. While the adaptive interval mechanism adjusts the TaskTracker timeout according to the estimated job running time in a dynamic way, the reputation-based detector compares the number of fetch errors reported when copying intermediate data from the mapper and when any of the TaskTrackers reaches a specific threshold that TaskTracker will be announced as a failed one. As authors explain, the adaptive interval is advantageous to small jobs while the reputation-based detector is mainly intended to longer jobs.

In the early versions of Hadoop (including the Hadoop 0.20 version), a crash failure of the JobTracker involved that all active work was lost entirely when restarting the JobTracker. The next Hadoop version 0.21 gave a partial solution to this problem, making periodic checkpoints into the file system [56], so as to provide partial recovery.

In principle, it is very hard to recover any possible data after a TaskTracker's failure. That is why Hadoop's reaction is to simply re-execute the tasks in the other TaskTrackers. However, there are works which have tried to take the advantage of checkpointing [16], or saving the intermediate data in a distributed file system [37, 38].

Regarding [16], among the interesting aspects of the pipelined Hadoop implementation is that it is robust to the failure of both map and reduce tasks, introducing the "checkpoint" concept. It works on the principle that each map and reduce task notifies the JobTracker, which spreads or saves the progress, informing the other nodes about it. For achieving this, a modified MapReduce architecture is proposed that allows data to be pipelined between operators, preserving the programming interfaces and fault tolerance models of a full-featured MapReduce framework. This provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. The paper has also demonstrated the benefits for batch processing: by pipelining both within and across jobs, the proposed implementation can reduce the time to job completion. This study work can also be considered as an optional solution to an omission failure.

In [37], authors propose an intermediate storage system, with two features in mind: data availability and minimal interference. According to this paper, these issues are solved with ISS (intermediate storage system), which is based on three techniques:

- Asynchronous replication. This does not block the ongoing procedures of the writer. Moreover the strong consistency is not required when having in mind that in platforms like Hadoop and similar, there is a single writer and single reader for intermediate data.
- Rack-level replication. This technique is chosen, because of the higher bandwidth availability within a rack, taking into account that the rack switch is not heavily used as the core switch.
- Selective replication. It is used considering that the replication will be applied only to the locally consumed data; in case of failure, the other data may be fetched again without problems.

This work is important to be mentioned, because for every TaskTracker failure, every map task that has been completed, it has already saved its output in a reliable storage system different from the local file system. In this way, the amount of redundant work for re-executing the map task that has been completed on the failed TaskTracker is reduced again.

A recent study [30] has investigated the impact of failures in shared Hadoop clusters. Accordingly, the authors evaluated the performance of Hadoop under failure when applying several schedulers (i.e., Fifo, delay, and capacity schedulers). They observe that the current failure handling mechanism is entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. This in turn results in a significant increase of the execution time of jobs with failed tasks. The performance degradation is caused by: (i) the waiting time for free resources to re-execute failed tasks, and (ii) not considering locality when scheduling failed tasks. In [66], the authors have proposed Chronos, a failure-aware scheduling strategy in shared Hadoop cluster.

Chronos triggers a lightweight preemption technique to free resources as soon as failure is detected, thus eliminating the waiting time. Furthermore, Chronos takes into consideration data locality of recovery tasks when preempting a running task. As a result, Chronos is able to correct the operation of Hadoops schedulers while improving the performance of MapReduce applications under failures.

11.4.2 Omission Failure (Stragglers)

An omission failure is a more general kind of failures. This happens when a process does not send (or receive) a message that it is supposed to send (or receive). In MapReduce terminology, omission failures are synonym for stragglers. Indeed, the concept of stragglers is very important in the MapReduce community, especially task stragglers, which could jeopardize the job completion time. Typically, the main causes of a MapReduce straggler task are: (i) a slow node, (ii) network overload, and (iii) input data skew [4].

Most of the state of the art in this direction has intended to improve the job execution time, by means of doubling the overall small jobs [2], or just by doubling the suspected tasks (stragglers) through different speculative execution optimizations [4, 13, 19, 32, 64, 72].

In [72], authors have also proposed a new scheduling algorithm called Longest Approximate Time to End (LATE) to improve the performance of Hadoop in a heterogeneous environment, brought by the variation of VM consolidation amongst different physical machines, by preventing the incorrect execution of speculative tasks. In this work, authors try to solve the issue of finding the real stragglers[3] among the MapReduce tasks, so as to speculatively execute them, while giving them the deserved priority. As the node heterogeneity is common in the real-world infrastructures and particularly cloud infrastructures, the speculative execution in the default Hadoop's MapReduce implementation is facing difficulties to give a good performance. The paper proposes an algorithm which should in some way improve the MapReduce performance in heterogeneous environments. It starts giving some assumptions made by Hadoop, and how they are broken down in practice. Later on, it proposes the LATE algorithm, which is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. The paper has an extensive experimental evaluation, which proves the valuable idea implemented in LATE.

Mantri [4] is another important contribution related to omission failures, which are called outliers in this paper. The main aim of the contribution is to monitor and cull or

[3]It is important to mention that, differently from [72] which considers tasks as stragglers, in the default paper of Google [19], a straggler is "a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation".

relax the outliers, accordingly to their causes. Based on their research, outliers have many causes, but mainly are enforced by MapReduce data skew, crossrack traffic, and bad (or busy) machines. In order to detect these outliers, Mantri does not rely only on task duplication. Instead, its protocol enhances according to outlier causes. A real-time progress score is able to separate long tasks from real outliers. Whereas the former tasks are allowed to be run, the real outliers are only duplicated when new available resources arise. Since the state-of-the-art contributions were mostly duplicating tasks at the end of the job, Mantri is able to make smart decision even before this, in case the progress score of the task is heavily progressing. Apart from data locality, Mantri places task based on the current utilization of network links, in order to minimize the network load and avoid self-interference among loads. In addition, Mantri is also able to measure the importance of the task output, and according to a certain threshold, it decides whether to recompute task or replicate its output. In general, the real-time evaluations and trace-driven simulations show Mantri to improve the average completion time for about 32 %.

GRASS [3] is another novel optimization framework, which is oriented to trimming the stragglers for approximation jobs. Approximation jobs are very common in the last period, because many domains are willing to have partial data in a specific deadline or error margin, instead of processing the entire data in an unlimited time or with 0 % error margin. After the introduction of the MapReduce programming model, which came with a simple solution of speculative execution of slow tasks (stragglers), the research community proposed decent alternatives, such as LATE [72] or Mantri [4]. However, they were not meant to give near to optimal solution for the domain of approximation analytics. And this is the advantage of GRASS, which is basically formed of two algorithms:

1. Greedy Speculative Scheduling (GS). This algorithm is intended to greedily pick a task that will be scheduled next.
2. Resource Aware Speculative Scheduling (RAS). This algorithm is able to measure the cost of leaving an old task to run or schedule a new task, according to some important parameters (e.g., time, resources, etc.).

GRASS is a combination of GS and RAS.

Depending on the cluster infrastructure size, but also on other parameters, the scheduler could impose different limitations per user or workload. Among others, it is common to place a limit on the number of concurrent running tasks. The overall set of these simultaneous tasks per each user (or workload) is known as wave. If a GRASS job requires many waves, then it starts with RAS and finally, in the last two waves uses GS. If the jobs are short, it may use only GS. This switching is mostly dependent on:

- Deadline–error bound.
- Cluster utilization.
- Estimation accuracy for two parameters, t_{rem} (remaining time for and old job), and t_{new} (an estimated time for a new job).

Evaluations show that GRASS improves Hadoop and Spark, regardless of the usage of LATE or Mantri, by 47 and 38 %, respectively, in production workloads of Facebook and Microsoft Bing. Apart from approximation analytics, the speculative execution of GRASS also shows to be better for exact computations.

In [13], authors propose an optimized speculative execution algorithm called Maximum Cost Performance (MCP) that is characterized by:

- Apart from the progress rate, it takes into consideration the process bandwidth in a phase, in order to detect the slow tasks.
- It uses exponentially weighted moving average (EWMA), whose duty is to predict the process speed and also predict the task remaining time.
- It builds a cost-aware model that determines what task needs a backup based on the cluster load.

In addition, the MCP contribution is based on the disadvantages of previous contributions, which mainly rely on the task progress rate to predict stragglers, inappropriate reaction on input data skews scenarios, unstable cost comparison between the backup and ongoing straggler task, etc. Evaluation experiments on a small-cluster infrastructure show MCP to have 39 % faster completion time and 44 % improved throughput when compared to default Hadoop.

In [64], authors propose an optimized speculative execution algorithm that is oriented to solving a single-job problem in MapReduce. The advantage of this work is that it takes into account two cluster scenarios, heavy and lightly loaded case. For the lightly loaded cluster, authors introduce two different speculative execution policies, early cloning, and later speculative execution based on the task progress rate. During the stage of heavily loaded cluster, the intuition is to use a later backup task. In this case, an Enhanced Speculative Execution (ESE) algorithm is proposed, which basically extends the work of [4]. Same authors have also introduced an additional extended work that assumes to work for multiple MapReduce jobs [63].

An important project related to Hadoop's omission failures is presented in [15]. In this work, authors have tried to build separate fault tolerance thresholds in the UpRight library for omission and commission failures, because omission failures are likely to be more common than commission failures. As we have mentioned before, during omission failures, a process fails to send or receive messages specified by the protocol. Commission failures exclude omission failures, including the failures upon which a process sends a message not specified by the protocol. Therefore, in the case of omission failures, the library can be fine-tuned in order to provide the liveness property (meaning that the system is "up") despite any number of omission failures.

In [49], the authors have studied the implications of speculative execution on the performance and energy consumption in Hadoop clusters. They observed that

speculative execution may result in a reduction in the energy consumption of Hadoop cluster if and only if the execution time of MapReduce application is noticeably reduced to compensate the energy cost of speculative execution (i.e., the extra power consumption due to the extra used resources).

The TaskTracker omission failures have also been addressed in some of the previous works we have mentioned [16, 18].

11.4.3 Arbitrary (Byzantine) Failure

The work discussing the omission failures in [15], is actually a wider review that includes the byzantine failures in general. The main properties upon which the UpRight library is based are:

- An UpRight system is safe ("right") despite r commission failures and any number of omission failures.
- An UpRight system is safe and eventually live ("up") during sufficiently long synchronous intervals when there are at the most u failures of which at most r are commission failures and the rest are omission failures.

The contribution of this paper is to establish byzantine fault tolerance as a viable alternative to crash fault tolerance for at least some cluster services rather than any individual technique. As authors say, much of their work involved making existing ideas fit well together, rather than presenting something new. Additionally, the performance is a secondary concern, with no claim that all cluster services can get low-cost BFT (byzantine fault tolerance).

The main goal of the work presented in [18] is to represent a BFT MapReduce runtime system that tolerates faults that corrupt the results of computation of tasks, such as the cases of DRAM and CPU errors/faults. These last ones cannot be detected using checksums and often do not crash the task they affect, but can only silently corrupt the result of a task. Because of this, they have to be detected and their effects masked by executing each task more than once. This BFT MapReduce follows the approach of executing each task more than once, but in particular circumstances. However, as the state machine approach requires $3f + 1$ replicas to tolerate at the most f faulty replicas, which gives a minimum of four copies of each task, this implementation uses several mechanisms to minimize both the number of copies of tasks executed and the time needed to execute them. In case there is a fault, from the evaluation results, it is confirmed that the cost of this solution is close to the cost of executing the job twice, instead of three times as the naive solution. Authors argue that this cost is acceptable for critical applications that require high level of fault tolerance. They introduce an adaptable approach for multicloud environments in [17].

In [62], authors propose another solution for commission failures called Accountable MapReduce. This proposal forces each machine in the cluster to be responsible

for its behavior, by means of setting a group of auditors that perform an accountability test that checks the live nodes. This is done in real time, with the aim of detecting the malicious nodes.

11.4.4 Network Failure

During a network failure, many nodes leave the Hadoop cluster; this issue has been discussed in different publications [14, 39, 42, 55], although for particular environments.

The work presented in [39] introduces a new kind of implementation environment of MapReduce called MOON, which is MapReduce on Opportunistic eNvironments. This MapReduce implementation has most of the resources coming from volunteer computing systems that form a Desktop Grid. In order to solve the resource unavailability, which is vulnerable to network failure, MOON supplements a volunteer computing system with a small number of dedicated compute resources. These dedicated resources keep a replica in order to enhance high reliability, maintaining the most important daemons, including the JobTracker. To enforce its design architecture, MOON differentiates files into reliable and opportunistic. Reliable files should not be lost under any circumstances. In contrast, opportunistic files are transient data that can tolerate some level of unavailability. It is normal to assume that reliable files have priority for being kept in dedicated computers, while opportunistic files are saved in these resources only when possible. In a similar way, this separation is also managed for read and write requests. MOON is very flexible in adjusting these features, based on the Quality of Service (QoS) needs. A reason for this is the introduction of a hibernate state and hybrid task scheduling. The hibernate state is an intermediate state whose main duty is to avoid having an expiry interval that is too long or short, which can incorrectly consider a worker node as dead or alive. A worker node enters in this state earlier than its expiry interval, and as a consequence it will not be supplied with further requests from clients. MOON changes the speculative execution mechanism by differentiating straggler tasks in frozen and slow lists of tasks, adjusting their execution based on the suspension interval, which is significantly smaller than the expiry interval. An important change to speculating tasks is their progress score, which divides the job into normal or homestretch. During the normal phase, a task is speculatively executed according to the default Hadoop framework; in a homestretch phase, a job is considered to have advanced toward its completion, therefore MOON tries to maintain more running copies of straggler tasks.

A later project similar to MOON is presented in [55]. Here, authors try to present a complete runtime environment to execute MapReduce applications on a Desktop Grid. The MapReduce programming model is implemented on top of an open-source middleware, called BitDew [25], extending it with three main additional software components: the MapReduce Master, MapReduce worker programs, and the MapReduce library (and several functions written by the user for their particular MapReduce

application). Authors wanted to benefit from the BitDew basic services, in order to provide highly needed features in Internet Desktop Grid, such as "massive fault tolerance, replica management, barriers free execution, and latency-hiding optimization, as well as distributed result checking". The last point (distributed checking) is particularly interesting, knowing that result certification is very difficult for intermediate results which might be very large to send for verification on the server side. The introduced framework implements majority voting heuristics, even though it involves larger redundant computation.

A research paper presented in [41] describes Cloud MapReduce (CMR), a new fully distributed architecture to implement the MapReduce programming model on top of the Amazon cloud OS. The nodes are responsible for pulling job assignments and their global status in order to determine their individual actions. The proposed architecture also uses queues to shuffle results from map tasks to reduce tasks. Map tasks are meant to write results as soon as they are available and reduce tasks need to filter out results from failed nodes, as well as duplicate results. The preliminary results of the work indicate that CMR is a practical system and its performance is comparable to Hadoop. Additionally, from the experimental results it can be seen that the usage of queues that overlap the map and shuffle phase seems to be a promising approach to improve MapReduce performance.

The works presented in [14, 42] are related to cloud environments, with particular emphasis on Amazon cloud. They discuss the MapReduce implementation on environments consisting of Spot Instances (SIs).[4]

In [14], a simple model has been represented. This model calculates the n-step probability, the expected lifetime of a VM, and the cost of termination, that is, the amount of time lost compared to having the set of machines stay up until completion of the job. Using the spot instances, in cases when there is no fault, the completion time may be speeded up. Otherwise, if there are failures, the job completion time may be longer than without using spot instances.

Liu's contribution [42] is a more mature proposal than the previous work. Here authors have tried to prove that their implementation, called Spot Cloud MapReduce, can take full advantage of the spot market, proposed by Amazon WS. As the name suggests, this implementation has been built on top of Cloud MapReduce (CMR), with additional changes:

- Modifying the split message format in the input queues (adding a parameter which indicates the position in the file where the processing should start).
- Saving the intermediate work when a node is terminated.
- Changing the commit mechanism to perform a partial commit.

[4]Spot instances are virtual machines resources in Amazon Web Services (WS), for which a user defines a maximum biding price that he/she is willing to pay. If there is no concurrence, the prices are lower and the possibility of using them is higher. But when the demand is higher, then Amazon WS has the right to stop your spot instances. If the spot instances are stopped by Amazon, the user does not pay, otherwise if the user decides to stop them before completing the normal hour, the user is obliged to pay for that consumption.

- Changing the way CMR determines the successful commit for a map split (electing a set of commit messages that is one more than the last key–value pair's offset).

The experimental evaluation shows that Spot CMR can work well in the spot market environment, significantly reducing cost by leveraging spot pricing.

11.4.5 Security Failure

The security concept is basically the absence of unauthorized access to, or handling of, system state [7]. This means that, authentication, authorization, and auditing go hand in hand, in order to ensure a system security. Whereas authentication refers to the initial identification of the user, the authorization determines the user rights, after he or she has entered into the system. Finally, the audit process represents an official user inspection (monitoring) to check if the user behaves according to its role. In other words, we could equate these terms with the pronouns *who* (authentication), *what* (authorization), and *when* (audit).

MapReduce's security in Hadoop is strictly linked to the security of HDFS; as the overall Hadoop security is grounded in HDFS, this means that other services including MapReduce store their state in HDFS. While Google's MapReduce does not make any assumption on security [19], early versions of Hadoop assumed that HDFS and MapReduce clusters would be used by a group of cooperating users within a secure environment. Furthermore, any access restriction was designed to prevent unintended operations that could cause accidental data loss, rather than to prevent unauthorized data access [40, 61].

The basic security definitions that include authentication, authorization, and auditing, were not present in Hadoop from the beginning. The authorization (managing user permissions) had been partially implemented. The auditing took place in the version 0.20 of Hadoop. The authentication was the last one, which came with Kerberos, an open-source network authentication protocol.

A user needs to be authenticated by the JobTracker before submitting, modifying, or killing any job. Since Kerberos authentication is bidirectional, even the JobTracker authenticates itself to the user; in this way, the user will be assured that the JobTracker is reliable. Additionally, each task is seen as an individual user, due to the fact that tasks now are run from the client perspective, the one which submitted the job, and not from the TaskTracker owner. In addition, the JobTracker's directory is not readable and writable by everyone as it happens with the task's working directories. During the authentication process, each user is given a token (also called a ticket) to authenticate once and pass credentials to all the tasks of a job; the token's default lifetime is meant to be around 8 h. While the NameNode creates these tokens, the JobTracker manages a token's renewal; token expiration is reasonably JobTracker dependent, in order not to expire prematurely for long-running jobs.

At the same year when Kerberos was implemented in Hadoop, another proposal called Airavat [51] tried to ensure security and privacy for MapReduce computations

on sensitive data. This work is an integration of mandatory access control (MAC) and differential privacy. MAC's duty is to assign security attributes to system resources, to constrain the interaction of subject with objects (e.g., subject can be a process, object can be a simple file). On the other side, differential privacy is a methodology which ensures that the aggregated computations maintain the integrity of each individual input. The evaluation of Airavat on several case studies shows flexibility in the maintenance of both accurate and private-preserving answers on runtimes within 32 % of the default Hadoop's MapReduce.

Apart from the different improvements in Hadoop security [31, 61], the work for preventing the Hadoop cluster from eavesdropping failures, has been slow. The explanation from the Hadoop community was that encryption is expensive in terms of CPU and I/O speed [52].

At the beginning, the encryption over the wire was dedicated only to some socket connections. In the case of Remote Procedure Call (RPC), an important protocol for communication between daemons in MapReduce, its encryption was added only after the main security improvement in Hadoop (by integrating Kerberos [36]). Most of the other encryption improvements (for instance, the shuffle phase encryption) came in a very recent Hadoop version [29], taking into consideration that Hadoop clusters may also hold sensitive information.

11.4.6 Apache Hadoop Reliability

Since its appearance in 2006, Apache Hadoop has undergone many releases [27]. Each of them has tried to improve different features of previous version, including fault tolerance. Table 11.2 shows Apache Hadoop 1.0 fault tolerance patches in a tree-like form, from the first Apache Hadoop 1.0 release (0.1.0) until release 1.2.1 which is the latest stable release up to the time of writing. These upgrades have played an important role in the later Hadoop evolution. An example of this is the introduction of speculative execution for reduce tasks, which caused many bugs in the previous days of its implementation. Therefore, the overall speculative execution mechanism was turned off by default later on, due to bugs in the framework. Actually the speculative execution mechanism was removed for some period, and later on, placed once again in the default functioning of the Apache Hadoop.

The Hadoop community was very active at the beginning, but this changed drastically through the years. A crucial reason for this was the existence of parallel projects, which tested new proposed features, but that were in their early phases (alpha or beta). Finally, a new release, Apache Hadoop 2.0, widely known as Hadoop YARN, was created. Table 11.3 shows Apache Hadoop 2.0 fault tolerance patches in a tree-like form, from the first Apache Hadoop 2.0 release (0.23.0) until release 2.7.1, which is the latest stable release up to the time of writing.

Table 11.2 Apache Hadoop 1.0: timeline of its fault tolerance patches

Year	Release	Patch
2006	0.1.0	The first release
	0.2.0	Avoid task rerun where it has previously failed (142); Don't fail reduce for a map impossibility allocation (169, 182); Five client attempts to JT before aborting a job (174); Improved heartbeat (186)
	0.3.0	Retry a single fail read, to not cause a failure task (311)
	0.7.0	Keep-alive reports, changed to seconds [10] rather than records [100] (556); Introduced killed state, to distinguish from failure state (560); Improved failure reporting (568); Ignore heartbeats from stale TTs (506)
	0.8.0	Make DFS heartbeats configurable (514); Re-execute failed tasks first (578)
	0.9.0	Introducing speculative reduce (76)
	0.9.2	Turn off speculative execution (827)
2007	0.10.0	Fully remove killed tasks (782)
	0.11.0	Add support for backup NNs, to get snapshotting (227, 959); Rack awareness added in HDFS (692)
	0.12.0	Change mapreduce.task.timeout to be per-job based (491); Make replication computation as a separate thread, to improve heartbeat in HDFS's NN (923); Stop assigning tasks to a dead TT (654)
	0.13.0	Distinguish between failed and killed task (1050); If nr of reduce tasks is zero, map output is written directly in HDFS (1216); Improve blacklisting of TTs from JTs (1278); Make TT expiry interval configurable (1276)
	0.14.0	Re-enable speculation execution by default (1336); Timed-out tasks counted as failures rather than killed (1472)
	0.15.0	Add metrics for failed tasks (1610)
2008	0.16.0	File permissions improvements (2336, 1298, 1873, 2659, 2431); Fine-grain control over speculative execution for map and reduce phase (2131); Heartbeat and task even queries interval, dependent on cluster size (1900); NN performance degradation from large heartbeat interval (2576)
	0.18.0	Completed map tasks should not fail if nr of reduce tasks is zero (1318)
	0.19.0	Introducing job recovery when JT restarts (3245); Add FailMon for hardware monitoring and analysis (3585)
2009	0.20.0	Improved blacklisting strategy (4305); Add test for injecting random failures of a task or a TT (4399); Fix heartbeating (4785, 4869); Fix JT (5338, 5337, 5394)
2010	0.20.202.0 (unreleased)	Change blacklist strategy (1966, 1342, 682); Greedily schedule failed tasks to cause early job failure (339); Fix speculative execution (1682); Add metrics to track nr of heartbeats by the JT (1680, 1103); Kerberos

(continued)

Table 11.2 (continued)

Year	Release	Patch
2011	0.20.204.0	TT should handle disk failures by reinitializing itself (2413)
	0.20.205.0	Use a bidirectional heartbeat to detect stuck pipeline (724); Kerberos improvements
2012	1.0.2	A single failed name dir can cause the NN to exit (2702)
	1.1.0	Lower minimum heartbeat between TT and JT for smaller clusters (1906)
2013	1.2.0	Looking for speculative tasks is very expensive in 1× (4499)
	1.2.1	The last stable release

Table 11.3 Apache Hadoop 2.0: timeline of its fault tolerance patches

Year	Release	Patch
2011	0.23.0	The first release; Lower minimum heartbeat interval for TaskTracker (MR-1906); Recovery of MR AM from failures (MR-279); Improve checkpoint performance (HDFS-1458)
2012	0.23.1	NM disk-failures handling (MR-3121); MR AM improvements: job progress calculations (MR-3568), heartbeat interval (MR-3718), node blacklisting (MR-3339, MR-3460), speculative execution (MR-3404); Active nodes list versus unhealthy nodes on the webUI and metrics (MR-3760)
	0.23.3	Timeout for Hftp connections (HDFS-3166); Hung tasks timeout (MR-4089); AM Recovery improvement (MR-4128)
	0.23.5	Fetch failures versus map restart (MR-4772); Speculation + Fetch failures versus hung job (MR-4425); INFO messages quantity on AM to RM heartbeat (MR-4517)
	2.0.0-alpha	NN HA improvements: fencing framework (HDFS-2179), active and standy states (HDFS-1974), failover (HDFS-1973), standbyNode checkpoints (HDFS-2291, HDFS-2924), NN health check (HDFS-3027), HA Service Protocol Interface (HADOOP-7455), in standby mode, client failing back and forth with sleeps (HADOOP-7896); haadmin with configurable timeouts for failover commands (HADOOP-8236)
	2.0.2-alpha	Encrypted shuffle (MR-4417); MR AM action on node health status changes (MR-3921); Automatic failover support for NN HA (HDFS-3042)

(continued)

Table 11.3 (continued)

Year	Release	Patch
2013	0.23.6	AM timing out during job commit (MR-4813)
	2.0.3-alpha	Stale DNs for writes (HDFS-3912); Replication for appended block (HDFS-4022); QJM for HDFS HA for NN (HDFS-3901, HDFS-3915, HDFS-3906); Kerberos issues (HADOOP-9054, HADOOP-8883, HADOOP-9070)
	2.1.0-beta	Reliable heartbeats between NN and DNs with LDAP (HDFS-4222); Tight DN heartbeat loop (HDFS-4656); Snapshots replication (HDFS-4078); Flatten NodeHeartbeatResponse (YARN-439); NM heartbeat handling versus scheduler event cause (YARN-365); NMTokens improvements (YARN-714, YARN-692); Resource blacklisting for Fifo scheduler (YARN-877); NM heartbeat processing versus completed containers tracking (YARN-101); AMRMClientAsync heartbeating versus RM shutdown request (YARN-763); Optimize job monitoring and STRESS mode versus faster job submission. (MR-3787); Timeout for the job.end.notification.url (MR-5066)
	2.1.1-beta	RM failure if the expiry interval is less than node-heartbeat interval (YARN-1083); AMRMClient resource blacklisting (YARN-771); AMRMClientAsync heartbeat versus runtime exception (YARN-994); RM versus killed application tracking URL (YARN-337); MR AM recovery for map-only jobs (MR-5468)
	2.2.0	MR job hang versus node-blacklisting feature in RM requests (MR-5489); Improved MR speculation, with aggressive speculations (MR-5533); SASL-authenticated ZooKeeper in ActiveStandbyElector (HADOOP-8315)
2014	2.3.0	SecondaryNN versus cache pools checkpointing (HDFS-5845); Add admin support for HA operations (YARN-1068); Added embedded leader election in RM (YARN-1029); Support blacklisting in the Fair scheduler (YARN-1333); Configuration to support multiple RMs (YARN-1232)
	2.4.0	DN heartbeat stucks in tight loop (HDFS-5922); Standby checkpoints block concurrent readers (HDFS-5064); Make replication queue initialization asynchronous (HDFS-5496); Automatic failover support for NN HA (HDFS-3042)
	2.4.1	Killing task causes ERROR state job (MR-5835)
	2.5.0	NM Recovery. Auxiliary service support (YARN-1757); Wrong elapsed time for unstarted failed tasks (YARN-1845); S3 server-side encryption (HADOOP-10568); Kerberos integration for YARN's timeline store (YARN-2247, HADOOP-10683, HADOOP-10702)

(continued)

Table 11.3 (continued)

Year	Release	Patch
	2.6.0	Encryption for hftp. (HDFS-7138); Optimize HDFS Encrypted Transport performance (HDFS-6606); FS input streams do not timeout (HDFS-7005); Transparent data at rest encryption (HDFS-6134); Operating secure DN without requiring root access (HDFS-2856); Work-preserving restarts of RM (YARN-556); Container-preserving restart of NM (YARN-1336); Changed NM to not kill containers on NM resync if RM work-preserving restart is enabled (YARN-1367); Recover applications upon NM restart (YARN-1354); Recover containers upon NM restart (YARN-1337); Recover NMTokens and container tokens upon NM restart (YARN-1341, YARN-1342); Time threshold for RM to wait before starting container allocations after restart/failover (YARN-2001); Handle app-recovery failures gracefully (YARN-2010); Fixed RM to load HA configs correctly before Kerberos login (YARN-2805); RM causing apps to hang when the user kill request races with AM finish (YARN-2853)
	2.6.1 (unreleased)	Make MR AM resync with RM in case of work-preserving RM restart (MR-5910); Support for encrypting Intermediate data and spills in local filesystem. (MR-5890); Wrong reduce task progress if map output is compressed (MR-5958)
2015	2.7.0	Block reports process during checkpointing on standby NN (HDFS-7097); DN heartbeat to Active NN may be blocked and expire if connection to Standby NN continues to time out (HDFS-7704); Active NN and standby NN have different live nodes (HDFS-7009); Expose Container resource information from NM for monitoring (YARN-3022); AMRMClientAsync missing blacklist addition and removal functionality (YARN-1723); NM fail to start with NPE during container recovery (YARN-2816); Fixed potential deadlock in RMStateStore (YARN-2946); NodeStatusUpdater cannot send already-sent completed container statuses on heartbeat (YARN-2997); Connection timeouts to NMs are retried at multiple levels (YARN-3238); Add configuration for MR speculative execution in MR2 (MR-6143); Configurable timeout between YARNRunner terminate the application and forcefully kill (MR-6263); Make connection timeout configurable in s3a. (HADOOP-11521)
	2.7.1	The last stable release

11.5 Other Data-Intensive Computing Systems

As mentioned before, MapReduce framework represents the de facto standard in the data-intensive computing community. However, there are many other projects, whose design and functionality differ from the basic MapReduce framework. Next, we present a collection of projects with significant impact in data-intensive computing.

11.5.1 Dryad/DryadLINQ

Knowing the benefits of Google's MapReduce, Microsoft designed its own data processing engine. In this way, Dryad [32] was introduced in 2007. After 1 year, Microsoft introduced a high-level language system for Dryad, composed of LINQ expressions, and called it DryadLINQ [67].

Dryad represents a general-purpose distributed execution engine, whose main target is coarse-grain data-parallel applications. In order to form a dataflow graph, Dryad combines computational *vertices* with communication *channels*. An application is run in Dryad by executing the vertices of the graph on a set of available machines, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.

Whereas mainly inspired from the (i) graphic processing units (GPUs) languages, (ii) Google's MapReduce, and (iii) parallel databases, Dryad is built also having in mind their disadvantages. As a consequence, Dryad as a framework allows the developer to have fine control over the communication graph, as well as the subroutines that live at its vertices. In order to describe the application communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices, a Dryad application developer can specify an arbitrary directed acyclic graph (DAG). By directly specifying this kind of graph, the developer has also greater flexibility to easily compose basic common operations, leading to a distributed analogue of "piping" together traditional Unix utilities such as grep, sort, and head.

Dryad graph vertices are enabled to use an arbitrary number of inputs and outputs. It is assumed that the communication flow determines each job structure. Consequently many other Dryad mechanisms (such as resource management, fault tolerance, etc.) follow this pattern. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. It is a logical computation graph that is automatically mapped onto physical resources by the runtime. At runtime each channel is used to transport a finite sequence of structured items.

Every Dryad job is coordinated by a master called "job manager" that runs either within the cluster or on a user's workstation, by having network access to the cluster. The job manager contains (i) the application-specific code, that allows to construct the job's communication graph, and (ii) library code, that allows to schedule the work across the available resources. Vertices transfer the data between them, therefore the job manager is only responsible for control decisions.

In Dryad, a failure of job manager means that the entire job fails, although other mechanisms (for example, checkpointing or replication) could be considered. As mentioned before, the fault tolerance policy works on a common case that all the vertex executions are deterministic. Due to failures, every vertex may be executed multiple times in sequence, or its many instances at any given time. If a vertex program runs slower than its peers, it gets duplicate executions; otherwise, after each heartbeat timeout, it gets re-executed.

The authors admit that for the nondeterministic vertices, Dryad does not provide any fault tolerance. However, this issue was planned as future goal of the framework.

Indeed, the Dryad fault-tolerant policy could be implemented by means of an extensible mechanism that allows nonstandard applications the possibility to modify their own behavior.

DryadLINQ represents a very important extension of Dryad, since it is a set of language extensions and the corresponding system that can automatically and transparently compile SQL, MapReduce, Dryad, and similar programs in a general-purpose language into distributed computations that can run on large-scale infrastructures. DryadLINQ does this in two ways, by (i) adopting an expressive data model of .NET objects; and (ii) by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language. A DryadLINQ program is based on LINQ expressions that are sequentially run on top of datasets. The DryadLINQ main duty is to translate the parallelism portion of the program into a distributed execution, ready to be executed on the Dryad engine.

11.5.2 SCOPE

SCOPE is a scripting language for massive data analysis [12], also coming from Microsoft. Its design has a strong resemblance to SQL, which was intentionally decided. SCOPE is a declarative language. As in the case of MapReduce, it hides the complexity of the lower platform and its implementation.

A user SCOPE script runs the basic SCOPE modules, (i) compiler, (ii) runtime, and (iii) optimizer, before initiating the physical execution. In order to manipulate input and output, SCOPE provides respective customizable commands, which are, *extract* and *output*. The *select* command of SCOPE is similar to the SQL one, with the main difference that subqueries are not allowed. To solve this issue, a user should rewrite complex queries with outer joins.

Apart from the SQL functionalities, SCOPE provides MapReduce-alike commands, which manipulate rowsets: *process*, *reduce*, and *combine*. The process command takes a rowset as input, and after processing each row, it outputs a sequence of rows. The reduce command takes a rowset as input, which has been grouped on the grouping columns specified in the ON clause. Then, it processes each group, and returns as output zero, one or multiple rows per group. The combine command takes two rowsets as input. It combines them depending on the requirements, and outputs a sequence of rows. The combine command is a binary operator.

Every SCOPE script resembles SQL, but its expression is implemented with C#, which needs to pass through the SCOPE compiler and optimizer, in order to be ready to run on parallel execution plan, which gets executed on the cluster as a Cosmos (Dryad) job. According to different evaluation experiments, SCOPE demonstrates its powerful query execution performance, that scales in a linear manner with respect to the cluster and data sizes.

The SCOPE fault tolerance policy relies on the Dryad's one. In SCOPE, Dryad has been renamed as Cosmos. In order to ensure availability, Cosmos replicates the data and its metadata by a quorum group of $2f + 1$ replicas, so as to tolerate f number

of failures. In order to ensure reliability, Cosmos enforces end-to-end checksums, to detect crash faulty components, whereas the data on disks is periodically scrubbed, to detect any corrupted or bit rot data before usage.

11.5.3 Nephele

In [60], authors present the basic foundations of Nephele, a novel research project at the time, whose aim was parallel data processing in dynamic clouds.

According to authors, state-of-the-art frameworks like MapReduce and Dryad are cluster-oriented models, which assume that their resources are a static set of homogeneous nodes. Therefore, these frameworks are not prepared enough for production clouds, whose exploitation of the dynamic resource allocation is a must. Based on this, they propose Nephele, a project which shares many similarities with Dryad, but providing more flexibility.

Nephele's architecture has a master–worker design pattern, with one Job Manager and many Task Managers. Each instance (aka VM) has its own Task Manager. As in Dryad, every Nephele job is expressed as a directed acyclic graph (DAG), where the vertices are tasks, and graph edges define the communication flow.

After writing the code for particular tasks, the user should define a Job Graph, consisting of linked edges and vertices. In addition, a user could specify other details, such as the number of subtasks in total, the number of subtasks per instance, instance types, etc. Each user Job Graph is then transformed into an Execution Graph by the Job Manager. Every specified manual configuration is taken into account by the Job Manager. Otherwise, the Job Manager places the default configuration according to the type of the respective job.

Compared to the default Hadoop on a small cloud infrastructure, the evaluation metrics are impressive and in favor to Nephele, showing better performance and resource utilization.

The main drawback of Nephele is that, due to its academic origin, it was not embraced by the research and industry community. One of the reasons could be its similarity with Dryad. An additional drawback of Nephele was its complexity, mainly compared to Hadoop MapReduce.

Finally, the Nephele fault tolerance policy still remains an open question. The authors admit the fault tolerance as desired but uncertain feature, because, as they mention, the optimal strategy is dependent on many parameters, among others, the task, its operations, the data, and the environment.

11.5.4 Spark

Spark is a novel framework for in-memory data mining on large clusters, whose main focus are applications that reuse the same dataset across multiple operations [69].

In this domain we found basically applications that are based on machine learning algorithms, such as text search, logistic regression, alternating least squares, etc. Spark programs are executed on top of the Mesos environment [28], where each of them needs its own driver (master) program to manage the control flow of the operations. Currently, Spark can also be run on top of other resource management frameworks, such as Hadoop YARN.

The main abstractions of Spark are:

- Resilient Distributed Datasets (RDDs) [68]. These are read-only collections of objects that are spread on cluster nodes.
- Parallel operations. These operations can be performed on top of the RDDs. Examples of these operations are *reduce*, *collect*, *foreach*, etc.
- Shared variables. These variables may be twofold: (i) broadcast variables, that copy the data value once to each worker; and (ii) accumulators, that can only "add" for being used as an associative operation, whose purpose (value) is readable by the driver only.

The most important abstraction of Spark are RDDs. Its primary use is to enable efficient in-memory computations on large clusters. This abstraction evolves in order to solve the main issues of parallel applications, whose intermediate results are very important in future multiple computations.

The main advantage of RDDs is the efficient data reuse, which comes with a good fault tolerance support. Its interface is based on coarse-grained transformations, by applying the same operation in parallel to a large amount of data. Each RDD is represented through a common interface, consisting of:

- A set of partitions. These are atomic pieces of the dataset.
- Set of dependencies. These are dependencies on parent RDDs.
- A function for computing the dataset from its parent.
- Metadata about its (i) partitioning scheme, and (ii) data placement.

Examples of applications that can take advantage of this feature are iterative algorithms and interactive data mining tools. Spark shows great results on some of these applications, outperforming Hadoop by 10× [68, 69, 71].

The Spark fault tolerance policy as well the scalability property are mainly based on the fundamental properties of MapReduce. In the case of the RDDs, they deploy the fault tolerance by means of the lineage concept, that is, if a node fails and an RDD partition is lost, the failed RDD partition is rebuilt from its parent datasets and eventually cached on other nodes. In addition to this, the Spark authors are planning to extend the fault tolerance in order to support other levels of persistence by means of in-memory replication across multiple nodes.

As part of Spark, the research community has proposed different modules, such as D-Streams [70, 71], GraphX [26], Spark SQL [6], and many others.

D-Streams represents a stream processing engine, an alternative to live queries (or operators) maintained by distributed event processing engines. Authors argue that it is better to have small batch computations using the advantages of in-memory

RDDs, instead of using long-live queries which are more costly and complex, mainly in terms of fault tolerance.

GraphX is a graph processing framework, which is built on top of Spark. GraphX represents an alternative to the classical graph processing systems, because it can efficiently handle iterative processing requirements of graph algorithm, unlike the general-purpose frameworks, such as MapReduce. The advantage of GraphX with respect to the classical graph processing frameworks is that it enables wider range of computations, and preserves the advantages of general-purpose dataflow frameworks, mainly the fault tolerance.

Finally, Spark SQL is another Apache Spark module, which enables an efficient intersection between relational processing and Spark functional programming. It does this by introducing the (i) *DataFrame API*, which enables the execution of relational operations, and (ii) *Catalyst*, which is another module that optimizes queries, and in addition simplifies data sources additions, and optimization rules, among others.

The main idea behind Apache Spark is to use iterative queries that are main memory based, which is also its main drawback. If the user request is not related to the previous and recent RDDs, the query process should start from the beginning. In this scenario, if we have to go back to the first iteration, MapReduce usually performs better than Spark.

11.6 Discussion

MapReduce programming model and the above-mentioned state-of-the-art improvements have filled many gaps on data processing requirements. However, there are many fault-tolerant issues that have not been solved yet. Below we address some open challenges.

It has been more than a decade since Google introduced MapReduce [19], but there are no many projects that analyze MapReduce with real-life traces and real-time large-scale infrastructures. What we can observe from the current situation is that leading companies process sensitive data with MapReduce. This data processing is highly confidential for their business and is a sufficient reason to avoid giving any details related to these results. As a consequence, most of the today's contributions are based on simulating failures [10, 21, 22, 35], or simulating the overall environment [58].

In addition, we are not aware of any large-scale comparison of datasets. The most common comparison we have seen in the data-intensive community field is the sorting time of 1 PB of data [53]. It would be desirable to have benchmark competitions of companies' data processing engines in the Big Data field. Indeed, it would be really challenging for the research and industry community to formalize a competition to measure different metrics of data-intensive processing frameworks, such as performance, scalability, or dependability.

Fault-tolerant abstraction models are another issue which is indeed missing in data-intensive computing systems. There are some projects that have offered different

approaches for this issue. Jin et al. [34] have derived a stochastic model that in some way predicts the performance of MapReduce applications under failures (crash failures). Their goal was to have a better understanding of fault tolerance mechanisms in Hadoop. A partial contribution to a theoretical failure model can be found in [58], which gives a simulation environment, and the possibility of fine-tuning different kinds of parameters. Another interesting work is presented in [48], where the authors propose a black box approach in order to detect and diagnose faults in MapReduce systems. While this contribution is valuable, it fails to solve half of the injected failures. Moreover, the mechanism has been evaluated only offline.

On the other side, there are studies that have expressed the view that this model is unworkable. In [10], it was explicitly stated that there is neither equivalent mathematical analysis that starts with a failure distribution and derives expected run time in the presence of failures nor optimization of the parameters of the system to minimize expected run time of the parallel applications under execution.

We consider that a theoretical model has not been presented yet, accepted by the whole community. We believe that additional challenges are important as well as possible, starting from the basic fault tolerance definitions, such as the failure detection.

While the handling and recovery in MapReduce fault tolerance via data replication and task re-execution seem to work well even at large scale [1, 37, 72], there is relatively little work on detecting failures in MapReduce. Accurate detection of failures is as important as failures recovery, in order to improve applications latencies and minimize resource waste. A new methodology to adaptively tune the timeout detector can significantly improve the overall performance of the applications, regardless of their execution environment [43]. Every MapReduce job should have its proper timeout, because in this way it could be possible to efficiently detect failures.

When reliability is improved, it is reasonable to think that these improvements are made at the expense of additional resource consumption. For instance, the replication improves the reliability, but increases the cost. The same occurs with cloning or speculating a task. More work is needed in improving reliability, maintaining similar resource utilization.

11.7 Summary

Many research projects have studied the MapReduce framework in the last few years, including its fault tolerance concepts and mechanisms. However, as far as we know, there is not a complete review of the research in MapReduce fault tolerance as an overall picture of what has been done and what is not solved yet. This survey addresses this gap, providing a systematic literature review on many contributions and an extensive analysis of new fault-tolerant mechanisms in MapReduce-based systems. Since there are other data-intensive approaches that have tried to go beyond the fundamental MapReduce functionality, we have also listed a relevant selection of these systems.

Finally, we have outlined some opening issues and key challenges for building efficient fault tolerance mechanisms in the MapReduce context. We argue that it is worth having a joint project from the different communities in order to handle issues such as a large-scale study of failures, where major companies could have a crucial role, and innovative and optimized failure models, where research communities can provide a significant contribution.

Acknowledgments The research leading to these results has received funding from the H2020 project reference number 642963 in the call H2020-MSCA-ITN-2014.

References

1. Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., Harris, E.: Scarlett: coping with skewed content popularity in mapreduce clusters. In: Proceedings of the Sixth Conference on Computer Systems, ACM, New York, NY, USA, EuroSys '11, pp. 287–300, (2011). http://doi.acm.org/10.1145/1966445.1966472
2. Ananthanarayanan, G., Ghodsi, A., Shenker, S., Stoica, I.: Effective straggler mitigation: Attack of the clones. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'13, pp. 185–198, (2013). http://dl.acm.org/citation.cfm?id=2482626.2482645
3. Ananthanarayanan, G., Hung, M.C.C., Ren, X., Stoica, I., Wierman, A., Yu, M.: GRASS: trimming stragglers in approximation analytics. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'14, pp. 289–302, (2014). http://dl.acm.org/citation.cfm?id=2616448.2616475
4. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using Mantri. In: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'10, pp. 1–16, (2010). http://dl.acm.org/citation.cfm?id=1924943.1924962
5. Apache Zookeeper: (2015). http://zookeeper.apache.org/
6. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '15, pp. 1383–1394 (2015). http://doi.acm.org/10.1145/2723372.2742797
7. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
8. Barborak, M., Dahbura, A., Malek, M.: The consensus problem in fault-tolerant computing. ACM Comput. Surv. **25**(2), 171–220 (1993). http://doi.acm.org/10.1145/152610.152612
9. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., Aiyer, A.: Apache Hadoop goes realtime at Facebook. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM, New York, NY, USA, SIGMOD '11, pp. 1071–1080 (2011). http://doi.acm.org/10.1145/1989323.1989438
10. Bressoud, T.C., Kozuch, M.A.: Cluster fault-tolerance: An experimental evaluation of checkpointing and MapReduce through simulation. In: Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops, IEEE, pp. 1–10 (2009). http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5289185
11. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer (2011)

12. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. Proc. VLDB Endow **1**(2), 1265–1276 (2008). http://dl.acm.org/citation.cfm?id=1454159.1454166
13. Chen, Q., Liu, C., Xiao, Z.: Improving mapreduce performance using smart speculative execution strategy. IEEE Trans. Comput. **63**(4), 954–967 (2014). doi:10.1109/TC.2013.15
14. Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., Krintz, C.: See spot run: using spot instances for MapReduce workflows. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, USENIX Association, Berkeley, CA, USA, HotCloud'10, pp. 7–7 (2010). http://dl.acm.org/citation.cfm?id=1863103.1863110
15. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, New York, NY, USA, SOSP '09, pp. 277–290 (2009). http://doi.acm.org/10.1145/1629575.1629602
16. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'10, pp. 21–21 (2010). http://dl.acm.org/citation.cfm?id=1855711.1855732
17. Correia, M., Costa, P., Pasin, M., Bessani, A., Ramos, F., Verissimo, P.: On the feasibility of byzantine fault-tolerant mapreduce in clouds-of-clouds. In: 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS), pp. 448–453 (2012). doi:10.1109/SRDS.2012.46
18. Costa, P., Pasin, M., Bessani, A., Correia, M.: Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In: Proceedings of the 3rd IEEE Second International Conference on Cloud Computing Technology and Science, IEEE Computer Society, Washington, DC, USA, CLOUDCOM '11, pp. 17–24 (2010). http://dx.doi.org/10.1109/CloudCom.2010.25
19. Dean, J., Ghemawat, S., Inc, G.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, USENIX Association, OSDI'04 (2004)
20. Dean, J.: Building software systems at google and lessons learned. Stanford EE Computer Systems Colloquium (2010). http://www.stanford.edu/class/ee380/Abstracts/101110-slides.pdf
21. Dinu, F., Ng, T.S.E.: Hadoop's Overload Tolerant Design Exacerbates Failure Detection and Recovery. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, ACM, New York, NY, USA, NetDB'11, pp. 1–7 (2011)
22. Dinu, F., Ng, T.E.: Understanding the effects and implications of compute node related failures in Hadoop. In: HPDC '12: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, ACM, New York, NY, USA, pp. 187–198 (2012). http://doi.acm.org/10.1145/2287076.2287108
23. Facebook, Inc.: (2015). https://www.facebook.com/
24. Facebook, I.: Under the Hood: Scheduling MapReduce jobs more efficiently with Corona (2012). http://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920
25. Fedak, G., He, H., Cappello, F.: BitDew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. J Netw. Compu. Appl. **32**(5), 961–975 (2009)
26. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'14, pp. 599–613 (2014). http://dl.acm.org/citation.cfm?id=2685048.2685096
27. Hadoop Releases: (2015). http://hadoop.apache.org/releases.html
28. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'11, pp. 22–22 (2011). http://dl.acm.org/citation.cfm?id=1972457.1972488

29. How-to: Set Up a Hadoop Cluster with Network Encryption: (2013). http://blog.cloudera.com/blog/2013/03/how-to-set-up-a-hadoop-cluster-with-network-encryption/
30. Ibrahim, S., Phuong, T.A., Antoniu, G.: An Eye on the Elephant in the Wild: A Performance Evaluation of Hadoop's Schedulers Under Failures. In: Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC-2015), held in conjunction with PODC'15 (2015)
31. Introduction to Hadoop Security: (2013). http://www.cloudera.com/content/cloudera/en/home.html
32. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys 2007, ACM, New York, NY, USA, EuroSys '07, pp. 59–72 (2007). http://doi.acm.org/10.1145/1272996.1273005
33. Jin, H., Ibrahim, S., Qi, L., Cao, H., Wu, S., Shi, X.: The MapReduce programming model and implementations. Cloud Computing: Principles and Paradigms pp. 373–390. doi:10.1002/9780470940105.ch14
34. Jin, H., Qiao, K., Sun, X.H., Li, Y.l.: Performance under Failures of MapReduce Applications. In: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE Computer Society, Washington, DC, USA, CCGRID '11, pp. 608–609 (2011). http://dx.doi.org/10.1109/CCGrid.2011.84
35. Jin, H., Sun, X.H.: Performance comparison under failures of MPI and MapReduce: An Analytical Approach. Future Gener. Comput. Syst. **29**(7), 1808–1815 (2013). http://dx.doi.org/10.1016/j.future.2013.01.013
36. Kerberos: The Network Authentication Protocol: (2015). http://web.mit.edu/kerberos/
37. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: Making cloud intermediate data fault-tolerant. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, New York, NY, USA, SoCC '10, pp. 181–192 (2010). http://doi.acm.org/10.1145/1807128.1807160
38. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: On availability of intermediate data in cloud computations. In: Proceedings of the 12th conference on Hot topics in operating systems, USENIX Association, Berkeley, CA, USA, HotOS'09, pp. 6–6 (2009). http://dl.acm.org/citation.cfm?id=1855568.1855574
39. Lin, H., Ma, X., Archuleta, J., Feng, W.c., Gardner, M., Zhang, Z.: MOON: MapReduce On Opportunistic eNvironments. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, New York, NY, USA, HPDC '10, pp. 95–106 (2010). http://doi.acm.org/10.1145/1851476.1851489
40. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Tech. rep., University of Maryland, College Park (2010)
41. Liu, H., Orban, D.: Cloud MapReduce: A MapReduce implementation on top of a cloud operating system. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 464–474 (2011). doi:10.1109/CCGrid.2011.25
42. Liu, H.: Cutting MapReduce Cost with Spot Market. In: Proceedings of the 3rd USENIX Conference on Hot topics in Cloud Computing, USENIX Association, Berkeley, CA, USA, HotCloud'11, pp. 5–5 (2011). https://www.usenix.org/conference/hotcloud11/cutting-mapreduce-cost-spot-market
43. Memishi, B., Ibrahim, S., Pérez, M.S., Antoniu, G.: On the Dynamic Shifting of the MapReduce Timeout. In: Kannan, R., Rasool, R.U., Jin, H., Balasundaram, S. (eds) Managing and Processing Big Data in Cloud Computing, IGI Global, Hershey, Pennsylvania (USA), pp. 1–22 (2016). doi:10.4018/978-1-4666-9767-6
44. Memishi, B., Pérez, M.S., Antoniu, G.: Diarchy: An Optimized Management Approach for MapReduce Masters. Procedia Comput. Sci. **51**, 9–18 (2015). http://www.sciencedirect.com/science/article/pii/S1877050915009874. International Conference On Computational Science, ICCS Computational Science at the Gates of Nature
45. Microsoft, Inc.: (2015). http://www.microsoft.com/
46. Mone, G.: Beyond Hadoop. Commun. ACM **56**(1), 22–24 (2013). http://doi.acm.org/10.1145/2398356.2398364

47. Okorafor, E., Patrick, M.K.: Availability of Jobtracker machine in Hadoop/MapReduce Zookeeper coordinated clusters. Adv. Comput.: An Int. J. **3**(3), 19–30 (2012). http://www.chinacloud.cn/upload/2012-07/12072600543782.pdf
48. Pan, X., Tan, J., Kavulya, S., Gandhi, R., Narasimhan, P.: Ganesha: blackBox diagnosis of MapReduce systems. SIGMETRICS Perform. Eval. Rev. **37**(3), 8–13 (2010). http://doi.acm.org/10.1145/1710115.1710118
49. Phan, T.D., Ibrahim, S., Antoniu, G., Bougé, L.: On Understanding the energy impact of speculative execution in Hadoop. In: IEEE International Conference on Green Computing and Communications (GreenCom 2015), Sydney, Australia (2015). https://hal.inria.fr/hal-01238055
50. RedHat: A guide for developers using the JBoss Enterprise SOA Platform (2008). http://www.redhat.com/docs/en-US/JBoss_SOA_Platform/4.3.GA/html/Programmers_Guide/index.html,programmersGuide
51. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for MapReduce. In: Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'10, pp. 20–20 (2010). http://dl.acm.org/citation.cfm?id=1855711.1855731
52. Shih, J.: Hadoop security overview—from security infrastructure deployment to high-level services. Hadoop & BigData Technology Conference (2012). www.hbtc2012.hadooper.cn/subject/keynotep8shihongliang.pdf
53. Sorting 1PB with MapReduce: (2013). http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html
54. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**:64–71 (2010). http://doi.acm.org/10.1145/1629175.1629197
55. Tang, B., Moca, M., Chevalier, S., He, H., Fedak, G.: Towards MapReduce for Desktop Grid Computing. In: Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, IEEE Computer Society, Washington, DC, USA, 3PGCIC '10, pp. 193–200 (2010). http://dx.doi.org/10.1109/3PGCIC.2010.33
56. The Apache Hadoop Project: (2015). http://hadoop.apache.org/
57. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, ACM, New York, NY, USA, SoCC '13, p. 5:1–5:16 (2013). http://doi.acm.org/10.1145/2523616.2523633
58. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: A simulation approach to evaluating design decisions in MapReduce setups. In: 17th Annual Meeting of the IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE, MASCOTS 2009, pp. 1–11
59. Wang, F., Qiu, J., Yang, J., Dong, B., Li, X., Li, Y.: Hadoop high availability through metadata replication. In: Proceedings of the First International Workshop on Cloud Data Management, ACM, New York, NY, USA, CloudDB '09, pp. 37–44 (2009). http://doi.acm.org/10.1145/1651263.1651271
60. Warneke, D., Kao, O.: Nephele: Efficient parallel data processing in the cloud. In: Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers, ACM, New York, NY, USA, MTAGS '09, pp. 8:1–8:10 (2009). http://doi.acm.org/10.1145/1646468.1646476
61. White, T.: Hadoop—The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated). O'Reilly (2012)
62. Xiao, Z., Xiao, Y.: Achieving accountable MapReduce in cloud computing. Future Gener. Comput. Syst. **30**, 1–13 (2014). http://dx.doi.org/10.1016/j.future.2013.07.001
63. Xu, H., Lau, W.C.: Optimization for speculative execution in a MapReduce-like cluster. In: 2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26–1May 1, 2015, pp. 1071–1079. http://dx.doi.org/10.1109/INFOCOM.2015.7218480

64. Xu, H., Lau, W.C.: Speculative execution for a single job in a mapreduce-like system. In: 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), pp. 586–593 (2014). doi:10.1109/CLOUD.2014.84
65. Yahoo! Inc: (2015). http://www.yahoo.com/
66. Yildiz, O., Ibrahim, S., Phuong, T.A., Antoniu, G.: Chronos: Failure-aware scheduling in shared Hadoop clusters. In: IEEE International Conference on Big Data (BigData 2015), pp 313–318 (2015). doi:10.1109/BigData.2015.7363770
67. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., Currey, J.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'08, pp. 1–14 (2008). http://dl.acm.org/citation.cfm?id=1855741.1855742
68. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, NSDI'12, pp. 2–2 (2012). http://dl.acm.org/citation.cfm?id=2228298.2228301
69. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, USENIX Association, Berkeley, CA, USA, HotCloud'10, pp. 10–10 (2010). http://dl.acm.org/citation.cfm?id=1863103.1863113
70. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, SOSP '13, pp. 423–438 (2013). http://doi.acm.org/10.1145/2517349.2522737
71. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing, USENIX Association, Berkeley, CA, USA, HotCloud'12, pp. 10–10 (2012). http://dl.acm.org/citation.cfm?id=2342763.2342773
72. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'08, pp. 29–42 (2008). http://dl.acm.org/citation.cfm?id=1855741.1855744
73. Zhu, H., Haopeng, C.: Adaptive failure detection via heartbeat under Hadoop. In: Proceedings of the 2011 IEEE Asia-Pacific Services Computing Conference, IEEE, New York, NY, USA, ApSCC'11, pp. 231–238 (2011)