RAFTing MapReduce: Fast Recovery on the Raft

Technical Report, Saarland University

Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

MapReduce is a computing paradigm that has gained a lot of popularity as it allows non-expert users to easily run complex analytical tasks at very large-scale. At such scale, task and node failures are no longer an exception but rather a characteristic of these systems. This makes fault-tolerance a critical issue for the efficient operation of any application. MapReduce automatically reschedules failed tasks to available nodes, which in turn recompute such tasks from scratch. However, this policy can significantly decrease performance of applications. In this paper, we propose a family of *R*ecovery *A*lgorithms for *F*ast-*T*racking (RAFT) MapReduce. As ease-of-use is a major feature of MapReduce, RAFT focuses on simplicity and also non-intrusiveness, in order to be implementation independent. To efficiently recover from task failures, RAFT exploits the fact that MapReduce produces and persists intermediate results at several points in time. RAFT piggy-backs checkpoints on the task progress computation. To deal with multiple node failures, we propose query metadata checkpointing. Therefore we keep track of the mapping between input key-value pairs and intermediate data. Thereby, RAFT does not need to re-execute completed map tasks entirely. Instead RAFT only recomputes intermediate data that were processed by local reducers and hence not shipped to another node for processing. We also introduce a scheduling strategy taking full advantage of these recovery algorithms. We implemented RAFT on top of Hadoop and evaluate it on a 45-node cluster using three common analytical tasks. Overall, our experimental results demonstrate that RAFT outperforms Hadoop runtimes by 23% on average under task and node failures. The results also show that RAFT has negligible runtime overhead.

1. INTRODUCTION

Data-intensive applications process vast amounts of data with special-purpose programs, e.g. the ever-popular inverted index and the PageRank Google applications. Even though the computations behind these applications are conceptually simple, the size of input datasets requires them to be run over thousands of computing nodes. For this, Google developed the MapReduce framework [4], which allows non-expert users to easily run complex tasks over very large datasets on large clusters. A salient feature of MapReduce is that users do not need to worry about issues such as parallelization and failover.

Fault-tolerance is, in fact, an important aspect in large clusters because the probability of node failures increases with the growing number of computing nodes. This is confirmed by a 9-year study of node failures in large computing clusters [19]. Moreover large datasets are often messy, containing data inconsistencies and missing values (bad records). This may, in turn, cause a task or even an entire application to crash. The impact of task and node failures can be considerable in terms of performance [20, 24, 4].

MapReduce makes task and node failures invisible to users; it automatically reschedules failed tasks[1] to available nodes. However, recomputing failed tasks from scratch can significantly decrease the performance of long-running applications [24] — especially for those applications composed of several MapReduce jobs — by propagating and adding up delays. A natural solution is to checkpoint the state of ongoing computation on stable storage and resume computation from the last checkpoint in case of failure. Nevertheless, checkpointing ongoing computation to improve the performance applications under failures is *challenging* for several reasons:

(1) Checkpointing techniques require the system to replicate intermediate results on stable storage. This can significantly decrease performance as MapReduce jobs often produce large amounts of intermediate results.

(2) Persisting checkpoints on stable storage usually requires intensive use of network bandwidth, which is a scarce resource in MapReduce systems [4].

(3) Recovering tasks from failures requires fetching intermediate results from stable storage, which in turn utilizes both network and I/O resources heavily.

As a result, using a straight-forward implementation of traditional checkpointing techniques [7, 17] would significantly decrease the performance of MapReduce jobs. In the remainder of this section, we describe how MapReduce deals with failures; we then illustrate the failover problem with a running example, and finally; we list our major contributions.

1.1 Fault-Tolerance in MapReduce

In the MapReduce framework, one central process acts as the *master* and coordinates MapReduce jobs while all other processes

[1] Failed tasks are tasks that crashed due to bad records or node failures.

act as *workers* on different nodes. Workers execute tasks as assigned by the master. For clarity, we denote a worker running a map task as *mapper* and running a reduce task as *reducer*. A worker can run multiple map and reduce tasks at the same time. Mappers (resp. reducers) running on the same worker are denoted *local* mappers (resp. *local* reducers), while those running on different workers are denoted *remote* mappers (resp. *remote* reducers). In MapReduce, there exist three kinds of failures: *task*, *worker*, and *master* failures.

(1.) *Task failures.* A task failure is an interruption on a running task, requiring the system to re-execute the interrupted task. There are several reasons to have a task failure: (i) *bad records*: MapReduce jobs typically process large messy datasets; (ii) *contention*: computing nodes compete for shared resources, which makes tasks slow-down and thus to be considered as failed; (iii) *media corruption*: MapReduce jobs do not account for all possible forms of corrupted media, like disk corruption [20], and; (iv) *bugs*: this includes bugs in MapReduce jobs as well as in third-party software.

When one of these reasons occurs, workers mark an interrupted task as failed and inform the master of the task failure. The master, in turn, puts the task back in its scheduling queue. If the failed task is a map task, the master notifies reducers of the re-execution decision as soon as the task is rescheduled. In case of bad records, MapReduce executes failing tasks twice before skipping a single bad record [4].

(2.) *Worker failures.* We denote a failure that causes a worker to stop accepting new tasks from the master as worker failure. This failure often results from hardware failures, e.g. memory errors, hard disk failures, or overheating CPUs. For example, a 10-node cluster at Saarland University experienced two network card failures and three hard disk failures within a couple of months. Additionally, network maintenance may also cause a worker to fail as reported in [4].

In MapReduce, the master relies on periodical communication with all workers to detect worker failures. If the master does not receive any response from a given worker during a certain amount of time, the master marks the worker as failed. Additionally, the master reschedules running (and some completed) map tasks and reduce tasks.

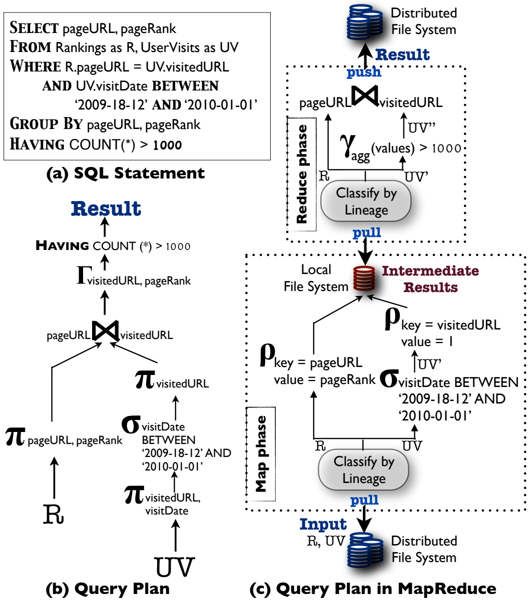
(3.) *Master failures.* In MapReduce, the master is a single point of failure, but this can be easily solved by having a backup node maintaining the status of the master node. Thus, the backup node can take over the master role in case of failure. This is why we consider the problem as a trivial engineering issue and do not cover it in this paper.

1.2 Motivating Example

MapReduce has some drawbacks when recovering from task and worker failures that significantly impact its performance. For clarity, let us illustrate this using a web log analysis example, containing *ranking* and *user visit* information for a large number of web pages [2]. Given the following two relations taken from the benchmark proposed in [15]:

Rankings(R)=(pageURL, pageRank, avgDuaration)
UserVisits(UV)=(sourceIP, visitedURL, visitDate).

Suppose that we would like to know the pageURL and pageRank of those pages that were visited more often than 1000 times during and between Christmas and New Year holidays. The SQL statement for this query is illustrated in Figure 1(a). Notice that this query is similar to the join query used in [15]. A DBMS may perform this query by executing the query plan depicted in Figure 1(b), while MapReduce would perform it as shown in Figure 1(c). Based on the Facebook application that loads 10 TB of compressed data per day [22], assume that the size of relation R is 100 GB and that the size of relation UV is 10 TB. Suppose that we want to analyze these datasets using 100 workers, each being able to perform two map tasks concurrently. Using input splits of 256 MB — which is a common configuration in practice — a single worker processes on average $(10.097\,\text{TB}/100)/256\,\text{MB} = 414$ map tasks. As a worker runs two tasks concurrently, it requires 207 waves of map tasks. Based on the results for the join query presented in [15], assume each map task is processed in $\sim$17 seconds. Thus, assuming perfectly partitioning, $17 \times 207 = 3519$ seconds ($\approx$1 hour) on average are required to perform the complete map phase of our running example.



```
SELECT pageURL, pageRank
FROM Rankings as R, UserVisits as UV
WHERE R.pageURL = UV.visitedURL
      AND UV.visitDate BETWEEN
          '2009-18-12' AND '2010-01-01'
GROUP BY pageURL, pageRank
HAVING COUNT(*) > 1000
```

(a) SQL Statement

(b) Query Plan

(c) Query Plan in MapReduce

Figure 1: Running example.

Since in Mapreduce failures are the rule and not the exception [4], let's assume that each input split contains one bad record in the middle, i.e. at offset 128MB. This is realistic as MapReduce jobs often process large messy datasets and thus can contain missing fields or incorrect formatting [23] — e.g. the date format might be incorrect. As a result, any map task will fail just after $\sim$8.5 seconds of processing time. Recall that MapReduce executes failing tasks twice before deciding to skip a bad record [4]. Since each worker performs 207 waves of map tasks, the entire MapReduce job will be delayed by at least $8.5 \times 2 \times 207 = 3519$ seconds ($\approx$1 hour) with a single bad record per task. This is a 100% runtime overhead, which clearly shows the need for more sophisticated algorithms that allow for reducing delays caused by these failures.

1.3 Contributions and Paper Outline

We propose a family of *R*ecovery *A*lgorithms for *F*ast-*T*racking (RAFT) MapReduce, which allows applications to significantly reduce delays caused by task and worker failures. Two salient features of RAFT are: *inexpensive* checkpoints and *fast* recovery. RAFT is inexpensive since it exploits the fact that MapReduce persists intermediate results at several points in time. RAFT thus persists only little additional data, such as split and task identifiers. RAFT is fast since it requires a few milliseconds to collect all the checkpoint information required, which results in a negligible de-

[2] An additional motivating example can be found in [24].

lay in MapReduce jobs.

Contributions. In summary, the major contributions of this paper are as follows:

(1.) *Local Checkpointing (RAFT-LC)*. We exploit that MapReduce persists intermediate results at several points in time to checkpoint the computing progress done by tasks (map and reduce tasks). This enables MapReduce to resume tasks from last checkpoints in case of task failure and hence to speed up applications under these failures.

(2.) *Remote Checkpointing (RAFT-RC)*. We invert the way in which reduce tasks obtain their input from mappers: we push the intermediate results into all reduce tasks as soon as results are produced by mappers. This allows MapReduce to avoid rescheduling completed map tasks in case of worker failures. Additionally, since some reduce tasks are local to mappers producing data — which will cause a loss of the data for local reducers in case of worker failure —, we replicate the intermediate results for local reducers to remote workers.

(3.) *Query Metadata Checkpointing (RAFT-QMC)*. We identify two problems when replicating intermediate results required by local reducers, which may significantly decrease the performance of applications. First, MapReduce jobs usually produce large amounts of intermediate results. Second, RAFT-RC can recover from only one worker failure. Therefore, instead of replicating intermediate results, we replicate a file consisting of the offset of all those input key-value pairs that produce an intermediate result and the identifier of the reducers that consume such results. As a result, we are able to significantly speed-up the re-computation of completed map tasks as well as to recover from more than one worker failure.

(4.) *Scheduling*. We propose a scheduling strategy that takes advantage of the local and remote checkpoints. To do so, our scheduling strategy differs from current MapReduce schedulers in that (i) it delegates the responsibility to workers for rescheduling failed tasks due to task failures (exploiting local checkpoints), and (ii) it preassigns reduce tasks to workers in order to allow mappers to push intermediate results to reduce tasks (enabling remote checkpoints).

(5.) *Exhaustive Validation*. We use realistic data to evaluate several aspects of the RAFT algorithms: performance under task and worker failures, overhead, scale-up, and speed-up. The results demonstrate that RAFT algorithms significantly outperform Hadoop by up to 27% in runtime performance.

The remainder of this paper is structured as follows. We survey related work in Section 2 and give a brief description of the MapReduce workflow in Section 3. We then present in Section 4 our family of algorithms and our scheduling strategy in Section 5. We demonstrate that RAFT produces the same output as normal MapReduce in Section 6. We then present our experimental results in Section 7. Finally, we conclude this paper in Section 8.

2. RELATED WORK

MapReduce was proposed by Google in 2004 [4] as a framework to facilitate the implementation of massively parallel applications processing large data sets. By design, MapReduce is already fault-tolerant. However, the algorithms it implements to recover from both task and node failures are quite simple and significantly decrease the performance of applications under these failures. In particular, applications whose pipelines consist of several MapReduce jobs, e.g. Pig [8, 14], Hive [21], and Sawzall [16], would benefit from better algorithms for failure recovery. This is because a delayed job completion is awkward as it blocks subsequent jobs, propagating and adding up delays.

A standard way to deal with task and worker failures is checkpointing, which has been extensively discussed in the database and distributed systems literature [2, 10, 12, 7, 17]. Generally speaking, the idea is to checkpoint ongoing computation on stable storage so that, in case of a failure, the state can be restored and the computation can resume from the last checkpoint. In MapReduce, however, network bandwidth typically is a scarce resource [4]. At the same time, MapReduce jobs produce large amounts of intermediate results (e.g. this is the case for an *inverted index* application). Therefore, replicating intermediate results may significantly decrease the performance of applications. As a result, straight-forward implementations of existing distributed checkpointing techniques are not suitable for MapReduce. A recent work [3] envisions a basic checkpointing mechanism, but the authors neither discuss it nor implement it in their prototype as of July 2010. One can then easily integrate our algorithms into [3] to increase performance in case of failures.

Some other research efforts have been made with the aim of combining traditional DBMSs with MapReduce concepts, where fault tolerance in distributed settings plays a key role. For example, HadoopDB [1] aims at increasing DBMSs fault-tolerance and scalability by using MapReduce as communication layer among several nodes hosting local DBMSs. Hadoop++ [5] makes usage of index and co-partitioned join techniques to significantly improve the performance of MapReduce jobs. However, these approaches have the same recovery limitations as the original MapReduce.

Recently, Yang et al. [24] propose Osprey, a distributed SQL database that implements MapReduce-style fault-tolerance techniques. However, they have done no improvement on recovery. As RAFT algorithms are quite general, one could apply them to Osprey [24] as well as to HadoopDB [1] and Hadoop++ [5] to speed up these systems failures.

Finally, ARIES [13] uses fuzzy checkpoints to speed-up both the checkpointing process and the analysis phase. ARIES, instead of checkpointing the contents of dirty pages, checkpoints the identifier of dirty pages (Dirty Pages Table). In contrast to RAFT, the metadata information used by ARIES is at the physical level. RAFT operates at a logical level; it keeps track of input records that produce intermediate results, instead of checkpointing the actual intermediate results themselves.

To the best of our knowledge, our research is the first to provide algorithms to recover from task and worker failures with performance improvement as the main goal.

3. PRELIMINARIES

MapReduce has gained a lot of popularity, from both research community and academia, because of its ease-of-use and robustness. While users simply need to describe their analytical tasks using two functions *map* and *reduce*, the MapReduce framework handles everything else including parallelization, replication, and failover. The MapReduce implementation of Google is not freely available, but an open source implementation exists, coined Hadoop [9].

MapReduce operates in three phases. In the first phase (map phase), the framework runs a set of M mappers in parallel on several nodes. Each mapper is assigned a disjoint subset of the input files. A mapper then executes a map-call for each input "record" and stores the output into main memory; from time to time workers spill buffered intermediate results to disk, which usually happens whenever the buffer is on the verge to overflow. When spilling intermediate results to disk, they are logically partitioned into R parts based on an intermediate key. Thus, $R \times M$ files will be generated in this phase. In the second phase (shuffle phase), the output of each mapper is grouped by intermediate key and redistributed across reducers. In the third phase (reduce phase), each reducer executes a reduce-call for each distinct intermediate key in its input and the set of associated intermediate values. A reducer stores the output for each group into a single file. Thus, the output of the MapReduce job will be distributed over R files.

For clarity, let us illustrate the MapReduce model via the Web log analysis example we presented in Section 1.2. In the example we would like to know the `pageURL` and `pageRank` of those pages that were visited more often than 1000 times during and between Christmas and New Year holidays. MapReduce performs this query as described below — see [5] for more details on the Hadoop plan.

First, mappers pull relations `Rankings` (`R`) and `UserVisits` (`UV`) from the distributed file system and feed the map function with records of both relations. Second, since a map function cannot know the origin of records, it has to classify such input (e.g. by counting attributes) by its lineage. Third, for `R`, a mapper takes the attribute `pageURL` as key and `pageRank` as value, and for `UV`, it first applies the date filter over `visitDate` and issues `visitedURL` as key assigned with a constant value of 1 (for those tuples having passed the filter, $\texttt{UV}'$). Fourth, MapReduce groups this intermediate output by the key `pageURL` and store it on local disk. Fifth, reducers pull the intermediate data required to run their reduce tasks and eventually merge partial groups originating from different mappers into one group per intermediate key. Sixth, the reduce function can then be called once per group. A reducer divides each group based on their lineage (by inspecting if the value is 1). It then aggregates values from $\texttt{UV}'$ and only keeps those having an aggregated value higher than `T` ($\texttt{UV}''$). Finally, the reducer joins `R` with $\texttt{UV}''$ and writes the output back to the distributed file system.

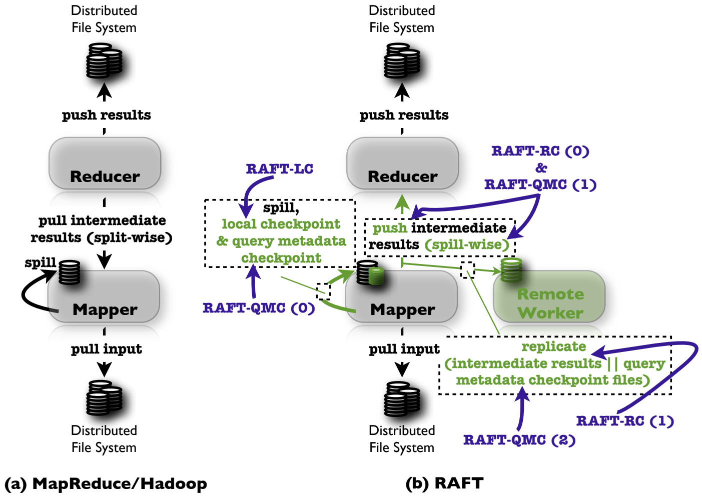
4. RAFT: A FAMILY OF FAST RECOVERY ALGORITHMS

We propose a family of fast and inexpensive *Recovery Algorithms for Fast-Tracking* (RAFT) MapReduce. The beauty of RAFT is that: (i) it requires only a few milliseconds to collect all the required checkpoint information (fast) and (ii) it piggy-backs checkpoints on the materialization of intermediate results (inexpensive). In the following, we first give an overview of our solution and then provide details on how RAFT recovers from task and worker failures.

4.1 Overview

A natural solution for recovering from task and worker failures is to checkpoint ongoing computation on stable storage. Such a solution, however, can decrease performance of MapReduce jobs because intermediate results are usually quite large. Nonetheless, MapReduce already produces and persists intermediate results at several points in time and even copies such results over the network anyway. We exploit this by introducing three types of checkpoints: *local*, *remote*, and *query metadata checkpoints*. Our checkpointing algorithms write checkpoints when intermediate results are persisted by workers. Piggy-backed checkpoints cause minimal overhead and are still highly effective. Figure 2 illustrates the key differences (green parts) between the original MapReduce and RAFT, which we briefly describe below and detail in the following sections.

- **Local Checkpointing**. As MapReduce does not keep tasks progress computation after task failures, it must perform failed tasks from the beginning. In contrary, we propose to perform *Local Checkpointing* (RAFT-LC) to deal with this. RAFT-LC stores tasks progress computation on the local disk of workers without sending replicas through the network so as to not increase network contention ("RAFT-LC" part in Figure 2(b)). One may think that RAFT-LC may considerable slow down tasks, since it has to repeatedly write all checkpoint information to disk, including the output produced so far. In our proposal, however, workers only perform RAFT-LC when they spill intermediate results of tasks to disk anyway. Hence, RAFT-LC comes almost for free.



(a) MapReduce/Hadoop

(b) RAFT

Figure 2: MapReduce vs RAFT (all three checkpointing algorithms).

Algorithm 1: Spill & Create Local Checkpoint

```
// Called by workers when spilling
Input    : DataObject $spill$, InputBuffer $inputBuffer$, Task $taskID$
begin
    String $spillID = spill.getSpillID()$;
    int $offset = inputBuffer.getLastProcessedByte()$;
    fork
        boolean $success = spill.flushToDisk()$;
        if $success$ then
            LocalCheckpoint $lcp =$
            $new\ LocalCheckpoint(taskID, spillID, offset)$;
            boolean $ok = lcp.checkConsistency()$;
            if $ok$ then
                $lcp.flushToDisk()$;
end
```

- **Remote Checkpointing**. In case of worker failures, MapReduce must reschedule completed map tasks in order to reproduce the intermediate results required by failed reducers or non-scheduled reduce tasks. To avoid this, we propose a *Remote Checkpointing* (RAFT-RC) algorithm that inverts the way in which reduce tasks obtain their input from workers. Rather than reduce tasks pulling their required intermediate results, mappers *push* their produced intermediate results into all reduce tasks (scheduled or not) as soon as they spill them to local disk ("RAFT-RC (0)" part in Figure 2(b)). Notice that this algorithm involves no additional network or disk I/O overhead, as each reduce task still receives only that data it needs to process anyway. Obviously, RAFT-RC does not push intermediate results to the reduce tasks that are local to the mapper producing the data (local reducers), because such results are already locally available. However, these intermediate results will be lost in case of worker failure. For this reason, RAFT-RC replicates intermediate results required by local reducers to remote workers ("RAFT-RC (1)" part in Figure 2(b)).

- **Query Metadata Checkpointing**. As a result of RAFT-RC, the scheduler does not need to reallocate completed map tasks in case of worker failures, because relevant data for all reducers is already copied remotely. Nevertheless, even if replicating intermediate results required by local reducers is a natural solution to avoid losing that information, RAFT-RC may decrease the performance of applications as they usually produce large amounts of intermediate results. Therefore, we propose a *Query Metadata Checkpointing* (RAFT-QMC) algorithm to deal with this problem. RAFT-QMC creates a query metadata checkpoint file per map task, consisting of both: the offsets of all input key-value pairs that produce intermediate results; and the identifiers of the reducers that will consume such results ("RAFT-QMC (0)" part in Figure 2(b)). RAFT-QMC then replicates these files to remote workers ("RAFT-QMC (2)" part in Figure 2(b)) — typically 8 bytes per log record, which generates negligible overhead. Like RAFT-RC, RAFT-QMC also pushes intermediate results to remote reducers ("RAFT-QMC (1)" part in Figure 2(b)).

4.2 Local Checkpointing (RAFT-LC)

Our local checkpointing algorithm RAFT-LC allows us to deal with task failures. Algorithm 1 shows the RAFT-LC pseudo-code for creating the local checkpoints. A mapper executes this algorithm when it spills intermediate results to local disk. RAFT-LC first retrieves progress information from the buffer containing input data (lines 2 and 3) before allowing for any further computation on the input buffer. After that, the mapper writes the spill to local disk in a parallel thread (lines 4 and 5). If the spill is correctly written, it proceeds to store the local checkpoint information on disk (lines $6-10$). A simple triplet of 12 bytes length is sufficient to store the checkpoint information: *taskID*, a unique task identifier that remains invariant over several attempts of the same task; *spillID*, the local path to the spilled data; *offset*, specifying the last byte of input data processed by spilling time. If an earlier checkpoint existed, it would be simply overwritten. Notice that spilled data is implicitly chained backwards. Thus, any checkpoint with a reference to the latest spill is sufficient to locate all earlier spill files as well.

To take advantage of local checkpoints persisted by RAFT-LC after a task failure, workers initialize tasks as shown in Algorithm 2. That is, a worker first verifies whether the allocated task is a new attempt of a previously failed task (line 3). In that case, the worker checks whether a checkpoint is available on disk, deserializes it, and verifies whether it is complete and consistent (lines $4-7$). If so, it simply resumes the task by updating all relevant state and progress information to the checkpoint. This simulates a situation where previous spills appear as if they were just produced by the current task attempt (lines $8-11$). Otherwise, the task is either just starting its first attempt or no valid checkpoint from the previous attempts could be retrieved. In that case, a worker simply processes again the task from the beginning.

Algorithm 2: Recover Task Locally

```
   // Called by workers during task initialization
   Input   : InputFileSplit $inputSplit$, int $currentTaskAttempt$
   Output  : ControlObject $progress$
1  begin
2      boolean $recovering = false$;
3      if $currentTaskAttempt \geq 2$ then
4          Directory $taskDir = getLocalFS()$;
5          if taskDir.containsCheckpoint() then
6              LocalCheckpoint
               $checkpoint = taskDir.getCheckpoint()$;
7              if checkpoint.isConsistent() then
8                  $inputSplit.seek(checkpoint.getOffset())$;
9                  $progress = checkpoint.getProgress()$;
10                 $checkpoint.getSpillFiles()$;
11                 $recovering = true$;

12     if recovering then
13         return $progress$; // proceed from last checkpoint
14     else
15         return $new\ ControlObject()$; // start from scratch
16 end
```

Algorithm 3: Create Remote Checkpoint

```
   // Called by workers when spilling
   Input   : DataObject $output$, ReducerScheduling $reducers$
1  begin
2      Partitions[] $P = spill.getReducerPartitions()$;
3      foreach $p$ in $P$ do fork
4          $PhysicalNode\ reducerNode = reducers.getNode(p)$;
5          if $reducerNode\ != LOCAL\_NODE$ then
6              $replicateData(p, reducerNode)$;
7          else
8              PhysicalNode $backupNode = getBackupNode()$;
9              if doCreateReplica(p, backupNode) then
10                 $replicateData(p, backupNode)$;

11 end
```

4.3 Remote Checkpointing (RAFT-RC)

We, now, present a remote checkpointing (RAFT-RC) algorithm that allows us to deal with worker failures. The main idea behind RAFT-RC is to push logical partitions[3], from intermediate results produced by mappers, to the reducers that will consume them. As opposed to the pull model, the push model significantly increases the probability of having intermediate results already backed up to the destination node when a worker fails. Consequently, the push model reduces the amount of data that need to be recomputed after such failures.

Algorithm 3 shows the RAFT-RC pseudo-code for creating the remote checkpoints. A mapper retrieves all the logical partitions from a spill (line 2) and for each part it gets the identifier of the reducer that will consume such a part (lines 3 and 4). Then, the mapper then pushes the partitions to remote reducers (lines 5 and 6). For fault-tolerance issues, the mapper also keeps its local copy of intermediate results even after it pushed them to remote workers. Indeed, a mapper does not push the logical partitions for local reducers — for clarity reasons we denote these partitions as local partitions — as they are already copied locally. Nevertheless, local partitions would be lost in case of worker failures. Though the number of local partitions is typically very small ($0 \leq localPartitions \leq 2$), such a loss is severe because all map tasks completed by a failed worker must be recomputed entirely. To avoid this, RAFT-RC replicates local partitions to pre-assigned backup nodes (lines $7-10$). This allows for not allo-

[3] As explained in Section 3, mappers logically partitions intermediate results based on an intermediate key and the number of reducers.

Algorithm 4: Startup With Remote Checkpoints

```
   // Called by reducers when recovering
   Input  : File[] $checkpoints$, MapperNodes $mappers$
   Output : ControlObject $progress$
1  begin
2      foreach $m$ in $mappers$ do fork
3          File[] $allFiles = computeExpectedFiles(m)$ ;
4          File[] $missing = allFiles \backslash checkpoints$;
5          foreach $x$ in $missing$ do fork
6              $backupNode = m.getBackupNode()$ ;
7              File $part = backupNode.pullData(x)$;
8              if $part \neq null$ then
9                  $part.store()$;
10             else
11                 $waitFor(x)$; // wait until pushed
       // join all parallel threads, then...
12     $sortMerge(allFiles)$;
13     $return\ progress$;
14 end
```

Algorithm 5: Create Query Metadata Checkpoint

```
   // Called by mappers after producing an output
   Input : Task $taskID$, InputBuffer $inputBuffer$, IntermediateKey $key$,
           LogBuffer[ ] $logBuffer$, File[ ] $file$
1  begin
2      int $offset = inputBuffer.getLastProcessedByte()$;
3      int $id = Partitioner.getReducer(key)$;
4      if $logBuffer[taskID].remaining() == 0$ then
5          $file[taskID].append(logBuffer[taskID])$;
6          $logBuffer[taskID].clear()$;
7      $logBuffer[taskID].put(offset + id)$;
8  end
```

Algorithm 6: Read Query Metadata Checkpoint

```
   // Called by mappers before setting
   // the next key-value pair for the map function
   Input     : InputFileSplit $inputSplit$, Set $offsets$
   Variables : int $startPos, currentPos$; // buffer positions
1  begin
2      if $offsets.hasNext()$ then
3          int $off = offsets.next()$;
4          int $bytesToSkip = off - startPos$;
5          if $bytesToSkip > MIN\_BYTES\_TO\_SKIP$ then
6              $inputSplit.seek(off)$; // random i/o
7              $startPos, currentPos = off$;
8          else
9              $inputSplit.skip(bytesToSkip)$; // seq. i/o
10             $currentPos = off$;
11             while $currentPos - startPos > BUFFER\_SIZE$ do
12                 $startPos$ += $BUFFER\_SIZE$;
13     else
14         $inputSplit.moveToEnd()$;
15 end
```

cating completed map tasks in case of worker failures, because reducers have their required data locally available and local reducers can pull local partitions from backup nodes as well. Notice that deciding when to replicate local partitions depends on several factors such as the number and size of local partitions as well as the current network bandwidth. A discussion on all these factors is out of the scope of this paper. We then assume that function $doCreateReplica$ (line 9) returns a per-application parameter set by the user launching the application — based on his knowledge of the application and the MapReduce cluster.

Algorithm 4 shows the pseudo-code of RAFT-RC for recovering from worker failures. For this, RAFT-RC initializes reduce tasks by taking into account remote checkpoints if there are any. Recovery thus happens on the fly. That is, for each mapper, a reducer first finds out the files — intermediate results previously pushed by a mapper — that are not available locally (lines $2-4$ of Algorithm 4). If there is any missing file, the reducer pulls each missing file from the backup node of that mapper (lines $5 - 9$). In case that a backup node does not have a missing file, a reducer simply waits until the mapper producing such a file pushes it (line 11). When all files are available locally, a reducer finally merges and sorts all files. Obviously, RAFT-RC works better if most of the remote checkpoints are available locally on the reducer. We achieve this by informing workers about reducers pre-scheduling decisions early. We discuss our scheduling strategy in more detail in Section 5.2.

4.4 Query Metadata Checkpointing (RAFT-QMC)

In the previous section, we presented our remote checkpointing algorithm that allows us to deal with worker failures in a more efficient way than the original MapReduce. Nonetheless, with RAFT-RC, one has to deal with two problems that may decrease performance of MapReduce jobs. First, MapReduce jobs typically produce large amounts of intermediate results and hence RAFT-RC will cause a significant overhead when replicating local partitions. Second, RAFT-RC can recover from a single worker failure only. This is because, in case of many worker failures, a failed reducer on any of the failed workers will require to pull intermediate results from the other failed workers. A simple solution to deal with this second problem is to replicate all the intermediate results produced by mappers. However, this solution is not suitable as it will only aggravate the first problem.

Instead, we propose a *Query Metadata Checkpointing* (RAFT-QMC) algorithm to tackle these two problems. The idea is to keep track of input key-value pairs that produce intermediate results in addition to push intermediate results to remote reducers. In this way, mappers can quickly recompute local partitions for failed reducers by processing only those key-value pairs that produce intermediate results belonging to such partitions. To achieve this, RAFT-QMC creates a query metadata checkpoint file per map task by logging the offset of each key-value pair that produces intermediate results (see Algorithm 5). A simple tuple of 8 bytes length is sufficient to store the query metadata checkpoint information: *offset*, specifying the byte of the input key-value pair; *reducerID*, the reducer requiring the intermediate results produced by the input key-value pair. RAFT-QMC pushes intermediate results to remote reducers similarly as RAFT-RC (lines $2 - 6$ of Algorithm 3), but it does not replicate local partitions. Instead, as soon as a mapper finishes a map task, RAFT-QMC replicates the query metadata checkpoint files to the preassigned backup nodes. Notice that, query metadata checkpoint files are typically quite small and hence RAFT-QMC generates much less overhead than RAFT-RC.

When a worker fails, the master simply reschedules the failed tasks as new tasks. If the failed tasks contain a reduce task, the master reschedules all map tasks that were completed by the failed worker — the worker that was running the failed reducer. However, these completed map tasks only have to recompute the local partitions lost by the failed worker. To do so, a mapper only processes the key-value pairs that produce a relevant output for missing local partitions as shown in Algorithm 6. That is, a mapper moves the pointer in the input buffer to the next offset in the query metadata checkpoint file (lines $2 - 12$). Notice that, RAFT-QMC performs a seek (lines $5 - 7$) on the input split only when this results in better performance than performing a sequential read (Lines $8 - 12$) — typically one should skip at least 4 MB to perform a seek. As soon as there is no more offset left in the query metadata checkpoint file, it moves the pointer to the end of the buffer — which results in the

Algorithm 7: Master Node: Reallocate Tasks

```
   // Called by the master whenever a failure occurs
   Input    : Failure $fail$
1  begin
2      if fail.isNodeFailure() then
3          Node $node = fail.getNode()$;
4          Task[] $interruptedTasks = node.getInterruptedTasks()$;
5          Task[] $reduceTasks$;
6          foreach $t \in interruptedTasks$ do
               // reschedule
7              $schedule(t)$;
8              if $t.isReduceTask$ then
9                  $reduceTasks.add(t)$;
           // re-compute output for local reducers
10         Task[] $completedMapTasks = node.getCompletedMapTasks()$;
11         if $!reduceTasks.isEmpty$ then
12             foreach $t \in completedMapTasks$ do
13                 $schedule(t, reduceTasks.getIdSet())$;
14     else
           // reschedule task on same node as previously
15         $schedule(fail.getTask(), fail.getNode())$;
16 end
```

finalization of a map task. As a result of this process, RAFT-QMC is able to significantly speed-up map tasks computation as mappers do not perform full scans of their input splits again.

5. SCHEDULING TASKS WITH RAFT

Like in the original MapReduce, our scheduler only assigns new tasks to available workers. Our scheduler, however, differs significantly from the original MapReduce when reallocating tasks after failures. We describe this reallocation behavior in this section and sketch it in Algorithm 7. For simplicity, we consider a single MapReduce job in the following.

5.1 Scheduling Map Tasks

We use a data locality optimization as in [4], that is, we aim at allocating map tasks as close as possible to the data they consume. Thus, when a mapper is available, the scheduler picks, if possible, a map task from its queue that requires data stored on the same node. If not, the scheduler tries to pick one map requiring data located on the same rack.

To deal with task and worker failures, our scheduler proceeds as follows.

- *Task failures.* Our scheduler allocates a failed map task to the same computing node right after its failure so as to reuse the existing local checkpoints (line 15 of Algorithm 7). Furthermore, this allows us to significantly reduce the waiting time for rescheduling failed map tasks. After the reallocation of a failed map task, a mapper then has to restart the failed task as discussed in Section 4.2.

- *Worker failures.* Our scheduler puts failed map tasks into its queue. Hence, these tasks become again eligible for scheduling to available workers (lines 6 and 7). Furthermore, unlike schedulers proposed in the literature [4, 11, 25], our scheduler reallocates map tasks — completed by failed workers — to recompute only the local partitions. This results in a significant speed-up of map tasks (lines $10 - 13$). To do so, mappers process input splits by considering only relevant key-value pairs as described in Algorithm 6.

5.2 Scheduling Reduce Tasks

So far, we saw that RAFT-RC as well as RAFT-QMC push intermediate results to all reducers, even if they are not scheduled yet, in order to recover from worker failures efficiently. To achieve this, our scheduler pre-assigns all reduce tasks to workers when launching a MapReduce job; then, it informs mappers of the pre-scheduling decision. With this pre-assigment, mappers know in advance to which workers to push the data. Then, when a worker is available to perform one reduce task, the scheduler simply allocates a task from its set of reduce tasks to it by taking into account the previous pre-assignment. This allows us to guarantee data locality with the intermediate results pushed by mappers. Some workers, however, typically complete tasks faster (fast workers) than others (slow workers). This occurs for two main reasons:

(1.) fast workers simply have more computing resources than slow worker (heterogenous clusters),

(2.) workers may take unusual long time to perform tasks because of hardware or software dysfunctions (strugglers).

As fast workers usually finish their reduce tasks before slow workers, our scheduler allocates tasks from other sets of reduce tasks (belonging to slow workers) to fast workers as soon as they finish with their own reducers set. In these cases, our scheduler falls back to the standard Hadoop: fast workers have to pull the required intermediate results from slow workers. Thus, to reduce the cost of data shipping in these cases, our scheduler picks reduce tasks from a large set of remaining reduce tasks — giving priority to those sets located in the same rack. Our scheduler proceeds as follows to deal with task and worker failures.

- *Task failures.* As for map tasks, our scheduler allocates a failed reduce task to the same computing node right after its failure in order to take advantage of our RAFT-LC algorithm (line 15 of Algorithm 7). Reducers then resume failed reduce tasks as discussed in Section 4.2.

- *Worker failures.* In these cases, our scheduler falls back to the standard Hadoop: it first puts failed reduce tasks back into its queue and reallocates them when one worker becomes free to perform one reduce task (lines $6 - 9$). When a reduce task is rescheduled to a new worker, it pulls all required intermediate results from all mappers containing part of such results. To reduce the impact on performance caused by this shuffle phase, our scheduler strives to allocate failed reduce tasks to one of the workers storing a part of their required data. Notice that, in contrast to map tasks, the scheduler reallocates running reduce tasks only, because completed reduce tasks store their output into stable storage such as GFS or HDFS.

6. CORRECTNESS

In this part, we show the equivalence of output between our recovery algorithms and the original MapReduce. As in [4], we assume that MapReduce jobs are deterministic.

Let $I_t = \{i_0, ..., i_n\}$ be the set of input records required by a task t. For any input record $i \in I_t$, t produces a set O_i of output results, where O_i might be the empty set — i.e., t produces no result by processing input record i. Given a set I_t, t produces a set $O_t = \bigcup_{i=0}^{n} O_i$. In other words, a set I_t maps to a set O_t. Formally, after task t processes the complete set I_t of input records, MapReduce ensures Equation 1.

$$\forall\, i \in I_t, \exists\, O_i \subseteq O_t :\; i \rightarrow O_i \qquad (1)$$

In case of task or worker failure, MapReduce still ensures above equation by processing again the entire set I of input records. We now demonstrate that our set of recovery algorithms ensure the same output as MapReduce for the same set of input records even in case of failure. With this in mind, we first demonstrate that any task (map or reduce task) receives the same input and thus produces the same output as original MapReduce in case of task failure.

LEMMA 1. *Given a task t, RAFT-LC ensures the same sets I_t and O_t as the original MapReduce in case of task failure.*

PROOF SKETCH. Let $I_c = \{i_0, ..., i_c\}$ be the set of input records read by a task t from set I_t (i.e. $I_c \subseteq I_t$) until the last local checkpoint done by the task — where i_c is the last input record whose set $O_{i_c} = \bigcup_{i=0}^{c} O_i$ of output results is included in the last checkpoint. By Equation 1, we have the corresponding set O_c of output results after processing I_c, i.e. $I_c \rightarrow O_c$. By convention, if task t fails, t is rescheduled to the same worker and RAFT-LC only processes the set $\overline{I_c} = I_t \backslash I_c = \{i_{c+1}, .., i_n\}$ of input records, i.e., the set of input records whose output was not persisted yet. Again, we have the corresponding set O'_c of output records after RAFT-LC processes the set $\overline{I_c}$ of input records, i.e. $\overline{I_c} \rightarrow O'_c$. By inferring from Equation 1, we have that $O'_c \cup O_c = O_t$ as $\overline{I_c} \cup I_c = I_t$. Therefore, RAFT-LC ensures the same input I_t and the same output O_t as the original MapReduce. In case that no local checkpoint was produced by t, RAFT-LC falls back to the original recovery process of MapReduce. Hence, it fetches again the entire set I_t of input records and trivially produces the same set O_t as the original MapReduce. □

We now demonstrate that any task receives the same input and produces the same output as MapReduce in case of worker failures. Due to space constraints, we prove the correct operation of RAFT-RC only.

LEMMA 2. *Given task t, RAFT-RC ensures the same sets I_t and O_t as the original MapReduce in case of worker failure.*

PROOF SKETCH. Let $I_w \subseteq I_t$ be the set of intermediate results stored by remote workers (denoted by set W) and $I_b \subseteq I_t$ be the set of intermediate results required by local reducers and thus stored on a backup worker (with $I_w \cap I_b = \emptyset$). Then, let $I_u = I_t \backslash (I_w \cup I_b)$ denote the set of intermediate results not yet produced. That is, $I_w \cup I_b$ are the intermediate results produced by completed map tasks, while I_u are the intermediate results to be produced by running and non-scheduled map tasks. Now, given a worker failure, RAFT-RC first reschedules a running task t to a different worker. If t is a reduce task, RAFT-RC pulls both set I_w from W and set I_b from the backup worker of the failed worker. Then, it waits for set I_u to be pushed by uncompleted map tasks. Thus, when uncompleted map tasks finish to push set I_u, t will contain the same set I_t as the original MapReduce. Hence, by Equation 1, t will produces the same set O_t of output records as MapReduce. If failed task t is a map task, RAFT-RC falls back to the original recovery algorithm of MapReduce by fetching again the entire set I_t. Thus, by Equation 1, RAFT-RC ensures the same set O_t as the original MapReduce. □

We can then conclude with the following theorem.

THEOREM 1. *Given a task t, RAFT algorithms always ensure the same sets I_t and O_t, of input and output records respectively, as the original MapReduce.*

PROOF. Implied by Lemma 1 and Lemma 2. □

7. EXPERIMENTS

We measure the performance of the RAFT algorithms with three main objectives in mind:

(1) to study how they perform under task and worker failures,

(2) to evaluate how well they scale up, and

(3) to measure the overhead they generate over Hadoop.

We first present the systems we consider (Section 7.1), the configuration of the testbed (Section 7.2), the benchmarks for our experiments (Section 7.3), and the methodology for running our experiments (Section 7.4).

7.1 Tested Systems

We take Hadoop as baseline since it is the most popular open-source implementation of the MapReduce framework [4]. We consider the following configuration settings.

- *Hadoop*. We use Hadoop 0.20.1 running on Java 1.6 and the Hadoop Distributed File System (HDFS). We make the following changes to the default configuration settings: (1) we store data into HDFS using 256 MB data blocks, (2) we use a sort buffer size of 50 MB, (3) we enable Hadoop to reuse the task JVM executor instead of restarting a new process per task, (4) we allow a worker to concurrently run two map tasks and a single reduce task, and (5) we set HDFS replication to 2.

We implemented all RAFT algorithms in a prototype we built on top of Hadoop 0.20.1 using the same configuration settings as above. To better evaluate the RAFT algorithms, we benchmark each recovery algorithm we proposed in Section 4 separately. In other words, we evaluate the following systems:

- **RAFT-LC**, local checkpointing algorithm that allows for dealing with task failures. For this, RAFT-LC checkpoints the progress of tasks to local disk at the same time that workers spill intermediate results to local disk.

- **RAFT-RC**, remote checkpointing algorithm that allows for recovering from worker failures. For this, RAFT-RC pushes intermediate results to remote reducers and replicates intermediate results required by local reducers.

- **RAFT-QMC**, query metadata checkpointing algorithm that, contrary to RAFT-RC, allows for recovering from more than one worker failure while replicating less amounts of data. Like RAFT-RC, RAFT-QMC pushes intermediate results to remote reducers, but we instead use the pull model in the RAFT-QMC experiments (Section 7.6). We do this since we want to measure the real impact in performance of creating query metadata checkpoints only.

- **Hadoop+Push**, we use this variant of Hadoop to evaluate the benefits for MapReduce when using using the push model in the shuffle phase, instead of the pull model.

- **RAFT**, to evaluate how well our techniques perform together, we evaluate how well RAFT-LC, RAFT-QMC, and Hadoop+Push perform together. Notice that we use RAFT-QMC instead of RAFT-RC, because it recovers from more than one worker failure.

7.2 Cluster Setup

We run all our experiments on a 10-node cluster where we dedicate one node to run the JobTracker. The remaining nine nodes ran five virtual nodes each, using Xen virtualization, i.e. resulting in a total of 45 virtual nodes. Node virtualization is also used by Amazon to scale up its clusters [6]. However, we recently showed that Amazon EC2 suffers from high variance in performance [18]. Therefore, running the experiments on our cluster allows us to get more stable results. Each physical node in our cluster has one 2.66 GHz Quad Core Xeon running 64-bit platform Linux openSuse 11.1 OS, 4x4 GB main memory, 6x750 GB SATA hard disks, and three Gigabit network cards. We set each virtual node to have a physical 750 GB hard disk and physical 3.2 GB main memory. The physical nodes are connected with a Cisco Catalyst 3750E-48PD, which use three Gigabit Ethernet ports for each node in channel bonding mode. From now on, we refer to virtual nodes as *nodes* for clarity.

7.3 Data and Benchmarks

We use the same data generator proposed by Pavlo et al. [15] to create all datasets for our experiments. Briefly, this benchmark first generates a set of HTML documents, each of them having links to other pages following a Zipfian distribution. It then creates `Rankings` and `UserVisits` relations, using the set of HTML documents and some randomly generated attribute values. We use a total dataset size of 50 GB for `Rankings` ($\sim$900 M tuples) and of 1 TB for `UserVisits` ($\sim 7,750$ M tuples). We consider a subset of two tasks of the benchmark proposed in [15]: (1) the *selection task* and (2) the *simple aggregation task*. Additionally, we consider a third task that modifies the simple aggregation task by introducing a selection predicate on `visitDate`: (3) the selective aggregation task. In the following , we describe these three tasks in more detail.

Q1. SELECTION TASK:

```
SELECT pageURL, pageRank
FROM Rankings
WHERE pageRank > 10;
```

The MapReduce job for this task consists of a map function that parses the input key-value pairs and outputs those `pageURL`s whose page rank is above a certain threshold. As in [15], this threshold defaults to 10 in our experiments. Finally, a single *IdentityReducer* merges the results and store the results on HDFS. We consider this task as it is a very common analytical task in practice, but also because MapReduce jobs often filter out their input.

Q2. SIMPLE AGGREGATION TASK:

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits
GROUP BY sourceIP;
```

The MapReduce job for this task consists of a map function and reduce function. The map function projects the two relevant fields, `sourceIP` and `adRevenue`, as intermediate key-value pairs. The reduce function simply aggregates the `adRevenue` for each `sourceIP` group. We consider this task because, like in several MapReduce jobs, it produces large amounts of intermediate results. Hence, this task allows us to better understand the impact in performance of RAFT algorithms.

Q3. SELECTIVE AGGREGATION TASK:

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits
WHERE visitDate > 1990-01-01
GROUP BY sourceIP;
```

The MapReduce job for this task consists of a map function and reduce function. The map function projects the fields `sourceIP` and `adRevenue` of any input key-value pair whose `visitDate` is after January 1st 1990. The reduce function then aggregates the `adRevenue` for each `sourceIP` group. We consider this task because we can control the amount of shuffled data — varying the predicate on `visitDate`. Thereby, we can represent several MapReduce jobs used in practice.

7.4 Methodology

We proceed in three phases so as to have a deeper understanding of performance of the RAFT algorithms. In summary, we evaluate RAFT algorithms in the following scenarios:

(1.) *Dealing with task failures* (Section 7.5). In this series of experiments, we focus on evaluating how well RAFT-LC allows applications to recover from task failures.

(2.) *Dealing with worker failures* (Section 7.6). In these experiments we evaluate how well RAFT-RC and RAFT-QMC allow applications to perform under worker failures.

(3.) *Putting everything together* (Section 7.7). We focus on evaluating RAFT in a mixed failure scenario. In other words, we want to know how well RAFT performs under task and worker failures at the same time. Additionally, we measure how much overhead the RAFT algorithms generate.

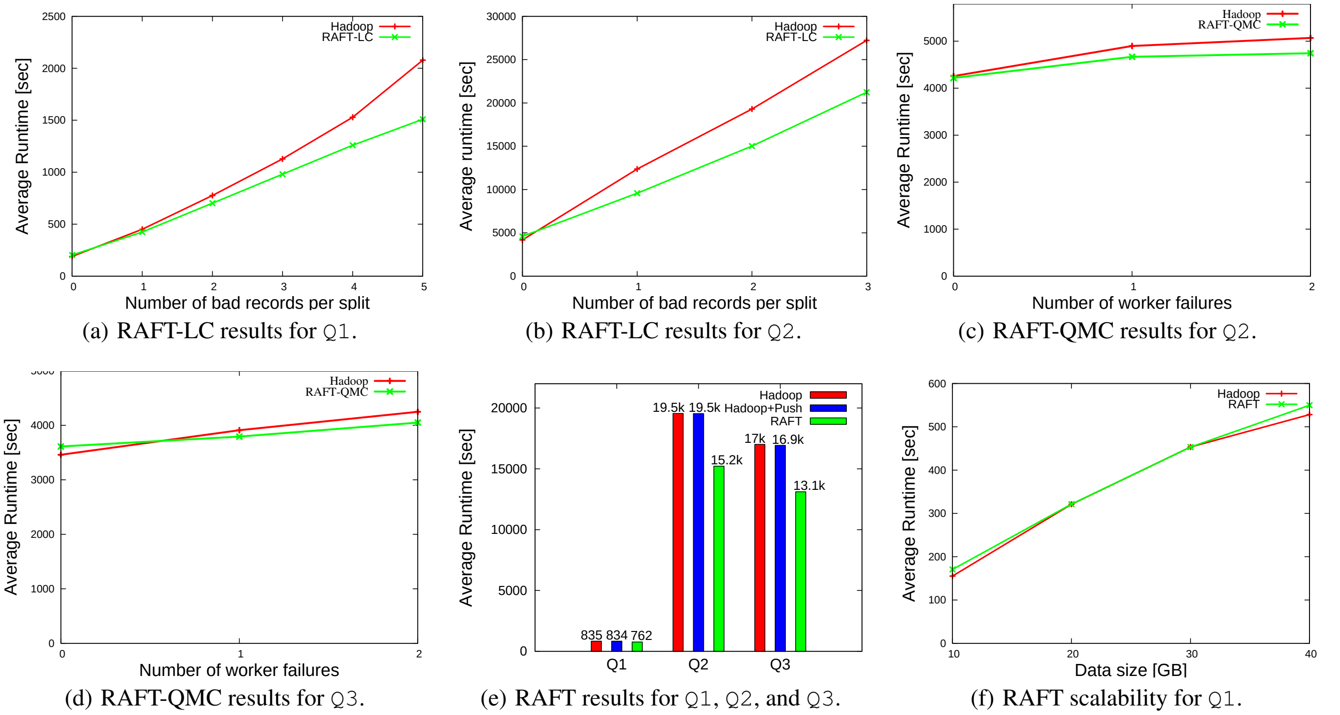
Notice that, for all results we present in this paper, we run each benchmark three times and report the averaged results.

7.5 RAFT-LC: Dealing with Task Failures

To evaluate RAFT-LC in the presence of task failures, we consider a "bad records scenario". That is, we introduce a varying number of bad records in input splits, which cause map tasks to fail. As mappers perform RAFT-LC when they spill to disk, we evaluate RAFT-LC in two different scenarios: (i) when mappers rarely spill to disk, and (ii) when mappers frequently spill to disk. Therefore, we only present results for `Q1` and `Q2` since they represent the two extremes in terms of the amount of intermediate results produced by mappers.

Figure 3(a) shows the runtime results for `Q1` with a varying number of bad records. We observe that RAFT-LC slightly outperforms Hadoop for one bad record per split. but significantly outperforms Hadoop by up to 27% for a higher number of bad records. RAFT-LC performs better than Hadoop as RAFT-LC allows failed map tasks to reuse the materialized results produced so far. Notice that the high selectivity of this task is not optimal for RAFT-LC, as mappers rarely spill intermediate results to disk — resulting in few local checkpoints.

Figure 3(b) shows the runtime results for `Q2`. In contrast to the results of `Q1`, we observe that RAFT-LC outperforms Hadoop by 25% already for one bad record — increasing to 27% for a higher number of bad records. This is because mappers do not filter any information from their input. Consequently, mappers spill much more intermediate results and thus RAFT-LC frequently creates local checkpoints. As a result, RAFT-LC can significantly speed-up map tasks by reusing more intermediate results. These results



(a) RAFT-LC results for Q1.

(b) RAFT-LC results for Q2.

(c) RAFT-QMC results for Q2.

(d) RAFT-QMC results for Q3.

(e) RAFT results for Q1, Q2, and Q3.

(f) RAFT scalability for Q1.

Figure 3: (a)-(b) RAFT-LC results for Q1 and Q2; (c)-(d): RAFT-QMC results for Q2 and Q3; (e) RAFT results for all three tasks; (f) RAFT scalability results.

Table 1: RAFT-RC and RAFT-QMC results for Q2 (runtime in seconds).

	RAFT-RC	RAFT-QMC
No worker failures	4601	4390
One worker failure	4699	4691

show the potential runtime improvement of RAFT-LC, as in practice MapReduce jobs typically produce large amounts of intermediate results. Notice that we do not present results for more than three bad records per input split, because we had to kill Hadoop since it was running for more than ten hours without finishing.

7.6 RAFT-RC & RAFT-QMC: Dealing with Worker Failures

To evaluate RAFT-RC and RAFT-QMC in the presence of worker failures, we kill a varying number of workers in the middle of the reduce phase. We present results only for Q2 and Q3, because the reduce phase for Q1 is too short (∼3 seconds) to kill the worker. We consider a worker that does not communicate with the master node during 60 seconds as failed. One of our goals in this section is to know whether RAFT-RC or RAFT-QMC performs better. Recall that these techniques differ in the way they recover from failures of local reducers. With this in mind, we evaluate both algorithms in a scenario with no worker failures and with one worker failure. Notice that we do not consider more than one worker failure in this experiment, because RAFT-RC cannot recover from more than one worker failure.

Table 1 presents the results for Q2 when comparing RAFT-RC with RAFT-QMC. The results show that RAFT-QMC runs 5% faster than RAFT-RC with no worker failures (we study this in detail in Section 7.7). We also observe that RAFT-QMC can still slightly outperform RAFT-RC with one worker failure. This is an interesting result as RAFT-QMC needs to locally recompute lost local partitions, while RAFT-RC only has to fetch such partitions from backup nodes. Our findings prove that RAFT-QMC can efficiently recover from worker failures with less network traffic and, at the same time, can deal with more than one worker failure.

Since RAFT-RC is limited to a single worker failure and RAFT-QMC performs better, we only compare RAFT-QMC with Hadoop for a larger number of worker failures. In these experiments, we do not consider more than two worker failures since such situations are not common in practice. Figure 3(c) shows the runtime results of these experiments for Q2. We observe that RAFT-QMC outperforms Hadoop by ∼7% on average. This is because RAFT-QMC makes less usage of the network by only replicating small query metadata checkpoint files, instead of important amounts of intermediate results required by local reducers. Furthermore, RAFT-QMC recomputes map tasks that were completed by failed workers for reproducing lost local partitions only. This does not occur in Hadoop, where map tasks process their whole input split again. Figure 3(d) shows the runtime results for Q3. These results show again that RAFT-QMC outperforms Hadoop for both one and two worker failures. However, we see that the runtime difference between RAFT-QMC and Hadoop was decreased with respect to Q2, because Q3 produces less intermediate results than Q2. As a result, the reduce phase run faster and thus a worker failure has less impact in performance.

7.7 RAFT: Putting Everything Together

We demonstrated in previous sections that, on average, RAFT algorithms significantly outperform Hadoop. We now evaluate how well they perform together, i.e. we evaluate our RAFT prototype. For this, we introduce two bad records per input split and kill only

Table 2: Overhead of RAFT algorithms.

	RAFT-LC	RAFT-RC	RAFT-QMC	RAFT
Q1	7.0%	13.7%	3.3%	5.7%
Q2	6.3%	9.5%	4.3%	6.6%
Q3	6.5%	9.8%	4.7%	8.7%

one worker as in Section 7.6.

Figure 3(e) illustrates these results for all three tasks. We observe that RAFT outperforms Hadoop by 23% on average for tasks Q2 and Q3, which are common tasks in practice. Moreover, we observe that RAFT outperforms Hadoop by 10% for Q1. As explained in Section 7.6, this is because Q1 is highly selective and thus RAFT produces few checkpoints. One can imagine that the good performance of RAFT is mainly due to the push model that RAFT uses. The results, however, prove that this is not the case. We observe that Hadoop+Push has about the same performance as Hadoop. This clearly demonstrates the high efficiency of RAFT algorithms.

We then evaluate well RAFT scales in terms of dataset size. For fairness reasons — regarding Hadoop —, we ran these experiments with no failures as Hadoop suffers from tasks and worker failures. Figure 3(f) shows these scalability results for Q1 — we do not present results for Q2 nor Q3, because we did not observe any different behavior from the results we present here. We observe that RAFT scales as well as Hadoop, which emphasizes the scalability of RAFT. Interestingly, we see that RAFT produces negligible overhead, while providing the same scalability as, and better failover performance than, Hadoop. Additionally, we analyzed how well RAFT reacts when new workers are added to the cluster (speed-up), but we do not show the results here due to space constraints. In those results, we observed that the speed-up of RAFT is the same as the speed-up of Hadoop and hence close to the optimal speed-up.

Finally, Table 2 shows the overhead of RAFT algorithms for tasks Q1, Q2, and Q3. Interestingly, we observe that RAFT-QMC generates an overhead of only 3.3% for Q1, of 4.3% for Q2, and of 4.7% for Q3, which is four times less than RAFT-RC for Q1 and two times less for Q2 and Q3. RAFT-QMC significantly outperforms RAFT-RC since, instead of replicating intermediate results (*more* than 150 MB per map task for Q2), it only replicates query metadata checkpoint files (*less* than 6 MB per map task for Q2). Still, RAFT-QMC recovers from worker failures slightly faster than RAFT-QMC, as shown in Table 1. On the other side, we observe that the overhead of RAFT-LC and RAFT is less than 8.8% which is also quite acceptable – especially for long-running jobs that take advantage of the recovery properties of using RAFT.

In summary, when compared to Hadoop, our results show that RAFT algorithms generate on average only ∼3% of runtime overhead, while they allow MapReduce jobs to run ∼23% faster in the presence of task and worker failures.

8. CONCLUSION

MapReduce is a popular programming model that allows non-expert users to easily run complex analytical tasks at an unprecedented very large-scale. At such scale, task and worker failures are no longer an exception, but rather a characteristic of these systems. In this context, MapReduce has gained a great popularity as it gracefully and automatically achieves fault-tolerance. In this paper, however, we showed that MapReduce has performance issues in the presence of task and worker failures.

To deal with this issue, we proposed a family of *R*ecovery *A*lgorithms for *F*ast-*T*racking (RAFT) MapReduce in the presence of these failures. The beauty of RAFT algorithms — namely, RAFT-LC, RAFT-RC, and RAFT-QMC — is that they exploit the fact that MapReduce persists intermediate results at several points in time, and even copies them through the network, in order to piggy-back checkpoints on tasks progress computation. In particular, besides a local checkpointing algorithm (RAFT-LC), we proposed a new query metadata checkpointing algorithm (RAFT-QMC) to deal with several worker failures at very low network cost. To achieve this, mappers produce query metadata checkpoints of task progress computation, containing all offsets of input key-value pairs that produce an intermediate result and the identifier of the reducers that will consume such results. As a result, failed map tasks (previously completed) only recompute intermediate results required by local reducers. This is not case in Hadoop, where these tasks must process again the whole input split.

To take full advantage of our recovery algorithms, we also introduced a scheduling strategy that differs from current MapReduce schedulers: (i) it delegates the responsibility to workers for rescheduling tasks failed due to task failures, and (ii) it pre-assigns reduce tasks to workers in order to allow mappers to push intermediate data to reducers.

We implemented RAFT algorithms in a prototype we built on top of Hadoop — a popular open source MapReduce implementation. We experimentally evaluated RAFT algorithms and compared their effectiveness with the original Hadoop. The results demonstrated that RAFT algorithms incur negligible runtime overhead and outperform Hadoop runtimes by 23% on average, and up to 27%, under task and worker failures. Another important result is that RAFT algorithms have the same scalability and speed-up as Hadoop, while they allow MapReduce jobs to recover faster from task and worker failures. Last but not least, we showed that RAFT-QMC produces much less runtime overhead than a straight-forward implementation of a checkpointing technique (RAFT-RC).

9. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Dist. Stream Process. Syst. *TODS*, 33(1), 2008.

[3] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *USENIX NSDI*, 2010.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[5] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, V. S. Yagiz Kargin, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[6] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[7] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A Survey of Rollback-Recovery Proto. in Message-Passing Syst. *CSUR*, 34(3), 2002.

[8] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a HighLev. Dataflow Syst. on Top of MapRed.: The Pig Experience. *PVLDB*, 2(2), 2009.

[9] Hadoop. http://hadoop.apache.org/mapreduce/.

[10] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A Cooperative, Self-Config. High-Avail. Solution for Stream Process. In *ICDE*, 2007.

[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Dist. Comp. Clusters. In *SOSP*, 2009.

[12] A.-P. Liedes and A. Wolski. SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algo. for in-Memory Databases. In *ICDE*, 2006.

[13] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Process. In *SIGMOD*, 2008.

[15] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, 2009.

[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analy. with Sawzall. *Scientific Programming*, 13(4), 2005.

[17] K. Salem and H. Garcia-Molina. Checkpointing Memory-Resident Databases. In *ICDE*, 1989.

[18] J. Schad, J. Dittrich, and J. Quiane-Ruiz. Runtime Measur. in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1), 2010.

[19] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *DSN*, 2006.

[20] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *ICDE*, 2010.

[21] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2), 2009.

[22] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *SIGMOD*, 2010.

[23] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[24] C. Yang, C. Yen, C. Tan, and S. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, 2010.

[25] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.