RAFT at Work: Speeding-Up MapReduce Applications under Task and Node Failures

Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, Jens Dittrich
Information Systems Group, Saarland University
Saarbrücken, Germany
http://infosys.cs.uni-saarland.de

ABSTRACT

The MapReduce framework is typically deployed on very large computing clusters where task and node failures are no longer an exception but the rule. Thus, fault-tolerance is an important aspect for the efficient operation of MapReduce jobs. However, currently MapReduce implementations fully recompute failed tasks (subparts of a job) from the beginning. This can significantly decrease the runtime performance of MapReduce applications. We present an alternative system that implements RAFT ideas [10]. RAFT is a family of powerful and inexpensive *Recovery Algorithms for Fast-Tracking* MapReduce jobs under task and node failures. To recover from task failures, RAFT exploits the intermediate results persisted by MapReduce at several points in time. RAFT piggybacks checkpoints on the task progress computation. To recover from node failures, RAFT maintains a per-map task list of all input key-value pairs producing intermediate results and pushes intermediate results to reducers. In this demo, we demonstrate that RAFT recovers efficiently from both task and node failures. Further, the audience can compare RAFT with Hadoop via an easy-to-use web interface.

Categories and Subject Descriptors

H.0 [**Information Systems**]: General

General Terms

Algorithms, Performance, Reliability

Keywords

MapReduce, Hadoop, Node Failures, Fault-Tolerance, Recovery

1. INTRODUCTION

MapReduce is a computing paradigm that has gained a lot of popularity as it allows non-expert users to easily run complex tasks at very large-scale. At such a scale, task and node failures are no longer an exception but rather a characteristic of these systems. This is confirmed by a 9-year study of node failures in large computing clusters [13]. Moreover, large datasets are often messy, containing data inconsistencies and missing values (bad records) [15]. Bad records can in turn cause a task, or even an entire application, to crash if not handled correctly.

MapReduce implementations, such as Hadoop[1] [1], make task and node failures invisible to users; Hadoop automatically reschedules failed tasks to available nodes, which in turn recompute such tasks from scratch. However, recomputing failed tasks from scratch can significantly decrease the performance of long-running applications by propagating and adding up delays [10, 16]. Thus, task and node failures can considerably decrease the runtime performance of MapReduce jobs [14, 16, 4].

A natural solution to solve this problem is to checkpoint task progress computation on stable storage and resume computation from the last checkpoint. Nevertheless, checkpointing ongoing computation is *challenging* as it would require Hadoop to replicate intermediate results to an additional node. Doing so would significantly decrease performance, because MapReduce jobs often produce large amounts of intermediate results. Furthermore, persisting checkpoints usually requires intensive use of network bandwidth, which is a scarce resource in MapReduce [4]. Therefore, we decided against a straight-forward implementation of traditional checkpointing techniques [3, 7, 9, 6, 11].

In this demo, we present an alternative system using the ideas of RAFT [10]. RAFT is part of the Hadoop++ project [2, 5], which aims at improving performance of Hadoop for analytical queries. In particular, in this demo, we show RAFT in action and demonstrate that it allows applications to significantly reduce delays caused by task and node failures. The RAFT demo comes with a lightweight and friendly web interface, whereby the audience can easily: (i) configure and compare RAFT with Hadoop, and (ii) observe the potential of each of the RAFT algorithms. This demo is interesting to everyone who uses MapReduce applications.

2. RAFT

To improve the performance of MapReduce applications under task and node failures, RAFT introduces two new checkpointing techniques: *local checkpointing* and *query metadata checkpointing*. To efficiently bring these ideas into the Hadoop framework, we introduce a new task scheduling strategy that takes advantage of these checkpoints. The beauty of RAFT is that it only produces a negligible overhead during normal operation, while it improves the performance of MapReduce applications under failures.

We implemented RAFT on top of Hadoop 0.20.1. We modified the mappers and reducers to enable them to create local and query metadata checkpoints. In the following, we briefly explain how RAFT creates and utilizes local and query metadata checkpoints. For further details please refer to [10].

2.1 Local Checkpointing

The main idea of local checkpointing is to utilize intermediate results (which are by default persisted by Hadoop) as checkpoints of ongoing task progress computation. As a result, the RAFT sched-

[1]Hereafter, we focus on Hadoop, but our ideas apply to other implementations of MapReduce as well.

SIGMOD'11, June 12–16, 2011, Athens, Greece.
ACM 978-1-4503-0661-4/11/06.

uler (see Section 2.3) can resume task computation from the last checkpoint in case of task failure.

Creation phase. By default, map tasks spill buffered intermediate results to local disk whenever the output buffer is on the verge to overflow. RAFT exploits this spilling phase to piggy-back checkpointing metadata on the latest spill of each map task. A simple triplet (*taskID*, *spillID*, *offset*) of 12 bytes length is sufficient to store such checkpointing metadata: *taskID* is a unique task identifier that remains invariant over several attempts of the same task; *spillID* is the local path to the spilled data; *offset* specifies the last byte of input data that was processed in that spill. As a result, local checkpointing comes almost for free.

Recovery phase. To recover from a task failure, the RAFT scheduler reallocates the failed task to the same node that was running the task. Then, the node resumes the task from the last checkpoint and reuses the spills previously produced for the same task. This simulates a situation where previous spills appear as if they were just produced by the task. In case that there is no local checkpoint available, the node recomputes the task from the beginning. This mainly happens when the node executes the task for the first time.

2.2 Query Metadata Checkpointing

The idea behind query metadata checkpointing is twofold. First, to push intermediate results to reducers as soon as map tasks are completed. Second, to keep track of those incoming key-value pairs that produce local partitions (i.e. intermediate results for local reducers) and hence that are not shipped to another node for processing. As a result, in case of a node failure, the RAFT scheduler can efficiently recompute local partitions.

Creation phase. By default, Hadoop reschedules the map tasks that were completed by mappers on a failed node to available nodes. These nodes then recompute such map tasks by processing the entire input again. This can significantly decrease the performance of MapReduce jobs, because Hadoop is recomputing all work already completed by failed nodes. To reduce such a negative impact in performance, RAFT inverts the way in which reduce tasks obtain their input from completed map tasks. 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 finish performing a map task. Additionally, when computing map tasks, RAFT maintains a list of the positions (offset in the input files) of those incoming key-value pairs that produce local partitions. In other words, RAFT creates a so-called *query metadata checkpoint file* per map task [10]. Then, RAFT replicates these checkpoint files to preassigned backup nodes. Typically, only a tuple (*offset*, *reducerID*) of 8 bytes is necessary per record: the *offset* of the input key-value pairs producing intermediate results for local partitions and the identifier (*reducerID*) of the reduce task that will consume such partitions.

Recovery phase. To recover from a node failure, the RAFT scheduler reallocates the failed tasks to available nodes. If the failed tasks contain a reduce task, the task scheduler also reallocates all map tasks completed by failed nodes. This is done for computing the local partitions again. To speed-up the computation of local partitions, mappers compute map tasks by processing the key-value pairs that produce relevant results for the lost local partitions only.

2.3 The Task Scheduler

Like the Hadoop scheduler, the RAFT scheduler assigns tasks to available nodes by maximizing data locality. However, the RAFT scheduler differs significantly from Hadoop and other MapReduce schedulers proposed in the literature [17, 8, 4] when reallocating tasks after task or node failures. We describe this reallocation behavior in the following.

Scheduling map tasks. In the presence of failures, the RAFT scheduler reallocates failed map tasks as follows:

(1.) *Task failures*. The RAFT scheduler strives to allocate a failed map task to the same computing node right after its failure. This is with the goal of reusing the existing local checkpoints produced so far. This allows us to significantly reduce the waiting time for rescheduling failed map tasks. After the reallocation, a map task then has to restart from the last local checkpoint.

(2.) *Node failures*. The RAFT scheduler puts failed map tasks into its queue and thus they become again eligible for scheduling to available nodes. Notice that, in case that a reduce task was running on a failed node, the RAFT scheduler reallocates the completed map tasks to compute local partitions again. To do so, map tasks process input splits by considering only relevant key-value pairs, i.e. based on the query metadata checkpoint files.

Scheduling reduce tasks. So far, we saw that RAFT pushes intermediate results to all reduce tasks, even if they are not scheduled yet. To recover from node failures efficiently, the RAFT scheduler preassigns all reduce tasks to nodes when launching a MapReduce job; then, it informs map tasks of the pre-scheduling decision. Some nodes, however, complete tasks faster (fast nodes) than others (slow nodes). Thus, as soon as a fast node finishes its preassigned reduce task set, the RAFT scheduler starts picking tasks from other reduce task sets (belonging to slow nodes). In these cases, RAFT falls back to the standard pull model used by MapReduce; fast nodes have to pull the required intermediate results from slow nodes. In case of failures, the RAFT scheduler allocates failed reduce tasks as follows:

(1.) *Task failures*. As for map tasks, the RAFT scheduler allocates a failed reduce task to the same computing node right after its failure in order to take full advantage of the local checkpoints produce so far. Reduce tasks can then start from the last checkpoint.

(2.) *Node failures*. In these cases, the RAFT scheduler falls back to the original MapReduce: it first puts failed reduce tasks back into its queue and reallocates them when one node is available to perform one reduce task. When a reduce task is rescheduled to a new node, it pulls all the intermediate results it requires from all map tasks containing part of such results. Notice that, in contrast to map tasks, the RAFT scheduler reallocates running reduce tasks only. This is because completed reduce tasks store their output into stable storage (HDFS) and thus do not need to be rescheduled.

3. DEMONSTRATION

Our main goal in this demo is to demonstrate the effectiveness of RAFT algprithms [10]. We evaluate the performance of RAFT under both task and nodes failures.

3.1 Demo Setup

In the demo, we compare the performance of RAFT with that of Hadoop in order to better understand the benefits of using RAFT. We use our local 10-node cluster at Saarland University. Each physical node runs five virtual nodes using Xen virtualization, resulting in a total of 50 virtual nodes. We execute RAFT and Hadoop simultaneously by splitting our cluster into two 25-node clusters (noted as RAFT and Hadoop clusters). Additionally, we consider to run the demo on a large Amazon EC2 cluster. However, Amazon EC2 suffers from high variance in performance [12].

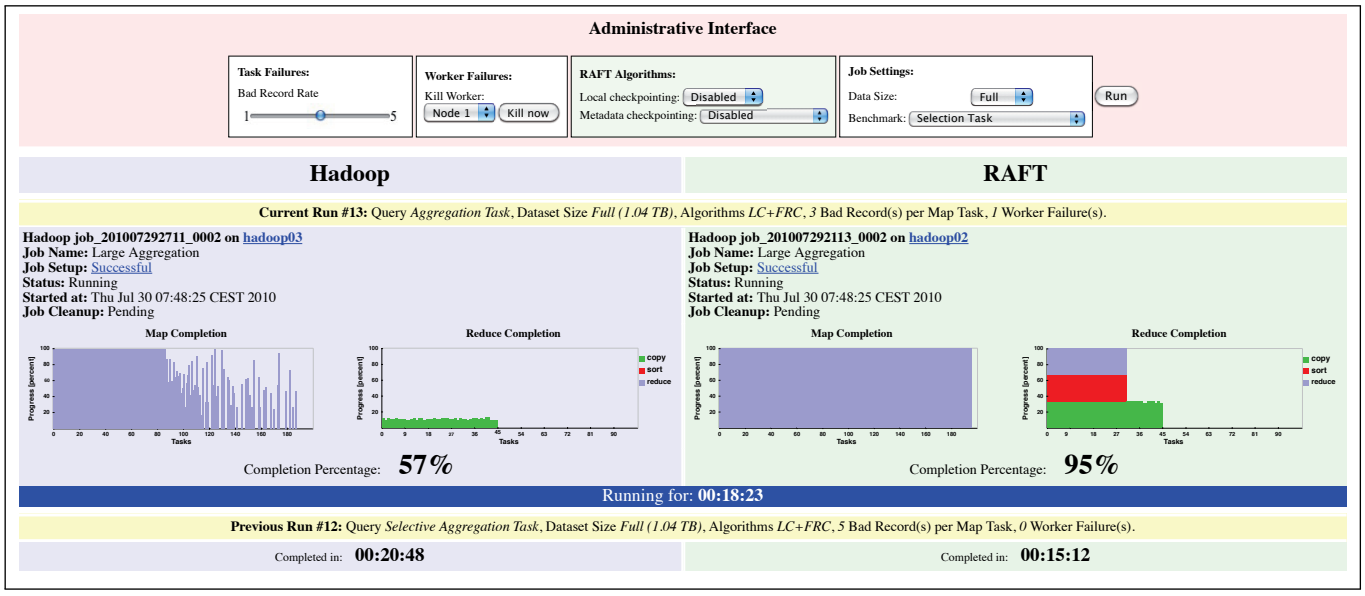

Figure 1: RAFT demo interface.

3.2 Demo Details

The RAFT demo offers a friendly web-interface (see Figure 1), whereby the audience can evaluate the different aspects of RAFT. For each of the following scenarios the user can vary (i) the data-size and (ii) the MapReduce job to be run. The audience can freely run one of the following MapReduce jobs: Selection, Simple Aggregation or Selective Aggregation jobs. For details on these jobs, refer to [10]. Similarly, the audience can run the local and query metadata checkpointing algorithms of RAFT individually.

(1.) Fast recovery from task failures. We consider a bad record scenario to demonstrate that RAFT significantly outperforms Hadoop under task failures using local checkpoints. For this, we insert incorrectly formatted fields into the input datasets, which causes a map task to fail. The audience is able to control the number of bad records per input split via the administrative interface (top area in Figure 1). The interface then allows the audience to compare the RAFT and Hadoop job progress.

(2.) Fast recovery from node failures. We also demonstrate the efficiency of RAFT when recovering from node failures. To do so, we provide a script to interactively stop any node in the RAFT and Hadoop clusters. Thus, the audience is invited to stop some nodes while a MapReduce job is running. Again, the audience can easily configure the job and control the node failure via the web interface.

(3.) Inexpensive checkpointing (low overhead). We show that RAFT incurs only little overhead compared to Hadoop. To do so, the audience is invited to run any benchmark query on both clusters with neither task nor node failures. Furthermore, we allow the audience to see the break down of the overhead generated by RAFT so as to better study the cost of each of the recovery algorithms that RAFT provides. For this, the audience can choose the individual RAFT algorithm they want to evaluate.

Notice that in all above scenarios, the audience is able to interactively compare the performance of both Hadoop and RAFT (blue and green areas in Figure 1).

4. REFERENCES

[1] Hadoop, http://hadoop.apache.org/.

[2] Hadoop++, http://infosys.cs.uni-saarland.de/hadoop++.php.

[3] M. Balazinska et al. Fault-Tolerance in the Borealis Distributed Stream Processing System. *TODS*, 33(1), 2008.

[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, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[6] M. Elnozahy et al. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *CSUR*, 34(3), 2002.

[7] J.-H. Hwang et al. A Cooperative, Self-Configurable High-Available Solution for Stream Processing. In *ICDE*, 2007.

[8] M. Isard et al. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

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

[10] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. RAFTing MapReduce: Fast Recovery on the Raft. In *ICDE*, 2011.

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

[12] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1), 2010.

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

[14] S. Subramanian et al. Impact of Disk Corruption on Open-Source DBMS. In *ICDE*, 2010.

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

[16] C. Yang et al. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE*, 2010.

[17] M. Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.