RCMP: Enabling Efficient Recomputation Based Failure Resilience for Big Data Analytics

Florin Dinu
EPFL, Lausanne, Switzerland

T. S. Eugene Ng
Rice University, Houston, TX, USA

***Abstract*—Data replication, the main failure resilience strategy used for big data analytics jobs, can be unnecessarily inefficient. It can cause serious performance degradation when applied to intermediate job outputs in multi-job computations. For instance, for I/O-intensive big data jobs, data replication is especially expensive because very large datasets need to be replicated. Reducing the number of replicas is not a satisfactory solution as it only aggravates a fundamental limitation of data replication: its failure resilience guarantees are limited by the number of available replicas. When all replicas of some piece of intermediate job output are lost, cascading job recomputations may be required for recovery.**

In this paper we show how job recomputation can be made a first-order failure resilience strategy for big data analytics. The need for data replication can thus be significantly reduced. We present RCMP, a system that performs efficient job recomputation. RCMP can persist task outputs across jobs and leverage them to minimize the work performed during job recomputations. More importantly, RCMP addresses two important challenges that appear during job recomputations. The first is efficiently utilizing the available compute node parallelism. The second is dealing with hot-spots. RCMP handles both by switching to a finer-grained task scheduling granularity for recomputations. Our experiments show that RCMP's benefits hold across two different clusters, for job inputs as small as 40GB or as large as 1.2TB. Compared to RCMP, data replication is 30%-100% worse during failure-free periods. More importantly, by efficiently performing recomputations, RCMP is comparable or better even under single and double data loss events.

I. Introduction

Data replication is the main failure resilience strategy used for big data analytics jobs today. It consists of writing several replicas (copies) of the same piece of data in different locations in the hope that on failures at least one replica survives. Unfortunately, when applied to intermediate job outputs in multi-job computations (series of jobs with the output of one job being the input of another), data replication can be greatly inefficient. This is important because multi-job computations are very popular. The primitives provided by big data processing systems (e.g. Hadoop, MapReduce) constrain the amount of work possible in a job. As a result, users need to divide their algorithms into multiple jobs [16], [15] or rely on higher level languages (e.g. Hive [23], Pig [19]) which usually also get compiled into sequences of jobs. We are aware of one computation requiring as many as 150 jobs to complete [1].

Even writing relatively few replicas (3 is common today [12]) can be an expensive operation in the context of big data analytics because the large data transfers required put significant stress on the network and the storage. Today's clusters are especially inefficient at handling large transfers due to economical constraints and architectural bottlenecks (e.g. oversubscribed networks [8], poor disk throughput [22]). For instance, in our evaluation we show that in the absence of failures, an I/O-intensive multi-job computation can double its running time when the replication factor is increased from 1 to 3. Importantly, the large performance penalty induced by data replication is paid on *every* use of replication, even during failure-free periods.

However, reducing the number of replicas is unsafe. This only aggravates the inherent limitation of data replication: its failure resilience guarantees are fundamentally limited by the number of replicas. Having insufficient replicas leaves computations exposed to failures and this can severely impact performance. The reason is that without the use of data replication, failures can easily cause data loss which can trigger cascading job recomputations: several jobs need to be recomputed to regenerate the lost data. In the worst case, the recomputation may have to revert all the way to the beginning of the multi-job computation. This suggests the need for devising efficient approaches to job recomputation.

Unfortunately, efficient recomputation support is noticeably absent in today's big data processing systems. The jobs affected by failures can be resubmitted but the system treats the resubmissions identically to the initial runs: it computes the jobs entirely. In this paper we show that efficient job recomputation can be made a first order failure resilience strategy for big data analytics. If done right, recomputation can be very efficient when failures do occur while bearing no cost during failure-free periods. Thus, the need for data replication can be greatly reduced. We present RCMP (name derived from the word recomputation) a system that performs efficient job recomputations in the context of the popular MapReduce paradigm. While extending the MapReduce model with support for efficient job recomputations is important and practically relevant given its popularity, we believe that our work on the importance and challenges of job recomputation transcends the MapReduce paradigm. Our work should apply to any big data parallel processing computation model based on DAGs of tasks. We view recomputation not as a replacement for replication but rather as a complement. Our position is that enabling efficient recomputation will in turn enable judicious use of replication thus facilitating improvements in overall computation performance.

RCMP is efficient. It recomputes only the minimum number

This research was sponsored by the NSF under CNS-1305379, CNS-1018807 and CNS-1162270, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp.

of tasks necessary for each recomputed job. For this, RCMP persists across jobs mapper outputs as well as reducer outputs that are part of successfully completed intermediate jobs. On failures that cause data loss, RCMP decides which jobs must be recomputed and based on the persisted data it also determines the minimum number of tasks to recompute for each recomputed job. RCMP's capability to maximize data reuse is shared by previous work in programming languages [20], [17] or cloud computing (Nectar [14], RDD [27]).

However, determining *what* to recompute is not RCMP's main contribution. RCMP goes beyond that. Its uniqueness stems from improving *how* a job is recomputed. In fact, recomputing the minimum number of tasks introduces challenges that would not be encountered otherwise. In this respect we identified and tackled in RCMP two fundamental challenges that greatly limit the efficiency of recomputation runs: the difficulty in fully leveraging the available compute-node parallelism and the presence of hot-spots. The first challenge is that during job recomputation, the recomputed tasks are unlikely to be numerous enough to efficiently utilize the available compute node parallelism. In other words, the task scheduling granularity used during the initial run is insufficient for efficient job recomputation. This results in underutilized compute nodes and consequently inefficient recomputation. The second challenge is that hot-spots appear during the recomputation of a job's mappers. In the initial run of the job, mapper accesses to input are essentially balanced over all nodes. The number of concurrent accesses on one node is limited. We find that during recomputation, mapper accesses can concurrently concentrate on one or a few storage locations. The resulting contention significantly increases mapper running time and consequently the whole job recomputation time. RCMP's approach to both challenges is to switch to a more fine-grained task scheduling granularity only during recomputation by splitting recomputed tasks. This better utilizes the available compute nodes and mitigates hot-spots by distributing computation and data accesses over all the nodes used for recomputation.

In our evaluation we quantitatively describe the magnitude of the overheads that data replication can introduce as well as the benefits of efficient recomputation. RCMP's benefits hold across two different clusters, with job inputs of either 40GB or 1.2TB. RCMP is implemented on top of Hadoop. In our experiments, during failure-free periods replication is 30% to 100% worse compared to RCMP. More importantly, by being efficient under recomputation, even under single and double data loss events, RCMP yields better or comparable total multi-job running time.

II. Background

The MapReduce paradigm A MapReduce job applies in a distributed fashion user-defined functions (UDF) to input datasets. The job input and output are composed of key-value pairs and are stored in a distributed file system. Jobs are further subdivided into tasks. A task is a small portion of the work and is assigned to only one compute node.

MapReduce has mapper and reducer tasks. Mappers run first and process the job input by applying the same map UDF to each record (i.e. key-value pair) in the mapper input. The union of all the mapper outputs comprises the input for the reducers. Mapper outputs are stored outside of the distributed file system, on the node that computed the mapper. Each reducer processes a separate set of keys. It applies a reducer UDF that takes as input one key at a time along with all the values corresponding to it. Mappers and reducers exchange data in the shuffle phase when each reducer copies from the completed mappers the key-value pairs that correspond to the keys it needs to process. Commonly, this shuffle phase results in an all-to-all traffic pattern between the nodes running the tasks. Finally, the union of all reducer outputs is the job output. A compute node can only run a limited number of tasks concurrently. This is enforced by the concept of mapper and reducer slots. A job runs in multiple waves when the number of tasks is greater than the number of slots.

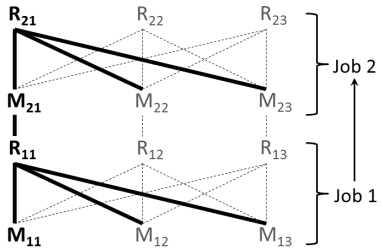


Fig. 1. The set of tasks and data transfers (both in bold) that are part of the recomputation of a MapReduce job under RCMP. M = mapper task, R=reducer task. A failure occurs just before Job 2 completes. The outputs of tasks M_{11}, R_{11}, M_{21}, R_{21} are lost due to the failure and need to be recomputed.

Cascading recomputations This paper focuses on collocated data center environments because of their popularity. In this case each node performs computation and also stores data that is part of the distributed file system (i.e. each node is both a storage and a compute node). Our contributions directly apply also to the non-collocated case where storage and computation are separated.

Collocation is more challenging because node failures impact both computation (tasks cannot finish) and storage (data is lost). For the job running at the time of the failure, part of the job input and some of the persisted mapper outputs are lost. Thus, the affected job cannot continue if its input has been insufficiently replicated. To recover using recomputation, it is necessary to cascade back to previous jobs to regenerate the lost data. In MapReduce this is especially important because the computation DAG always has local components. A failure can easily lead to the loss of data from all jobs already finished. Thus, recomputation needs to be efficient. It may need to cascade all the way to the beginning of the computation.

As an example, consider Figure 1 which illustrates recomputation in RCMP. The failure occurs just before Job 2 finishes. R_{21} is lost and needs to be recomputed. But R_{21} requires the output of M_{21} which is also lost. In turn, M_{21} is based on the output of R_{11} which was also on the failed node and was lost. Thus, Job 2 cannot continue before R_{11} is recomputed. Therefore, RCMP has to cascade back to Job 1 to recompute the lost part of Job 2's input. RCMP recomputes only the tasks that had outputs on the failed compute node (M_{11}, R_{11},

M_{21}, R_{21}) as well as the data transfers that are required for these recomputed tasks (bold lines in Figure 1). Note that the recomputation work performed by RCMP is a fraction compared to recomputing an entire job.

Notations and clarifications We refer to the first execution of a job as the *initial run* of that job. During failure recovery, parts of the job (some of its tasks) may have to be re-executed. We call such a re-execution a *recomputation run* of that job. This job-level recomputation should not be confused with speculative execution in MapReduce. Speculative execution is a task-level mechanisms useful only when the input to the job is available. Speculative execution detects slow tasks and duplicates or restarts them on other nodes in the hope that in the new location they will progress faster.

This paper is about data replication (i.e. writing multiple copies of the same data on multiple nodes). This should not be confused with task replication, a completely different mechanism outside the scope of this paper. For simplicity, we use the term ”replication” to refer to data replication.

III. Why replication is problematic

In this section we provide detailed arguments to support our claim that replication is too costly to be the only failure resilience strategy used in big data analytics. Despite the failure resilience guarantees and the performance benefits that replication can offer in a few narrow cases, there are simply too many practically relevant situations in which replication costs far outweigh the benefits. This suggests the need for devising more efficient failure resilience strategies.

Part of the overhead of replication stems from inefficiencies in current systems. For example, data center networks are often oversubscribed [8] and the disk throughput obtained by applications can fall well short of the disk hardware capabilities [22], [21]. A number of proposals improve I/O performance and could also decrease the absolute replication costs. Batching optimizations can mitigate the detrimental effect of excessive seeks caused by concurrent disk accesses [22], [21]. Leveraging raw access to disk [18] mitigates inefficiencies resulting from layering a distributed file system on top of general purpose file systems which are not optimized for big data workloads [22]. While such solutions incrementally improve performance, the fundamental limitation remains. Replication adds extra I/O work to the system. Thus, the relative overhead of replication is expected to persist.

A. Overrated benefits of replication

Failures are not an ubiquitous threat Replication does provide some useful failure resilience guarantees. Current replication strategies [12] protect against the simultaneous failure of two nodes or against single rack-level failures. This is particularly useful when a job has a high probability of encountering a failure. One example are large-scale, long-running jobs spanning thousands of nodes. However, most data analytics users do not run such large scale jobs and few companies have extremely large clusters. In 2011, Cloudera reported that the median size of a data analytics cluster was less than 30 nodes [3] while the average was around 200 nodes.

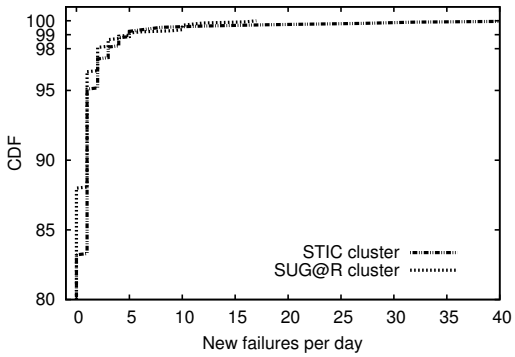


Fig. 2. CDF of new failures per day for two clusters at Rice University.

At this moderate scale node failures are expected only at an interval of days [21].

Figure 2 depicts the rate at which machines become unavailable for the STIC (218 nodes) and SUG@R (121 nodes) clusters at Rice University. The traces have been made publicly available [2]. The traces are based on daily, automated checks of node unavailability, end in Sept 2012 and start in Sept 2009 for STIC and Jan 2009 for SUG@R. Only 12% of days for SUG@R and 17% of days for STIC show new failures. Discussions with IT revealed that most failure events reported in Figure 2 are likely hardware issues that take at least a day to solve. The few days with many nodes becoming unavailable are unplanned situations (scheduler and file system outages or performance degradation). Our numbers corroborate with estimates from other studies [21] and suggest that for the popular moderate-sized clusters occasional failures should be expected but are not an ubiquitous threat. Therefore, in these situations continuous use of replication for failure resilience is unwarranted.

Data locality is oftentimes inconsequential If a node storing a piece of data also processes it, then the computation is said to be data-local. Increased data locality can lead to improved job running times when it is more efficient to process data locally (e.g. cluster with a highly oversubscribed network). Replication can improve the chance of scheduling data-local tasks. More replicas result in better chances that a node having a replica of a task’s input data will be selected by the scheduler to run the task.

However, there are many situations in which data locality is either not applicable, is inconsequential or easily obtainable without replication. First, data locality is not even applicable [18] to non-collocated environments. All transfers are remote in this case. Second, data locality is inconsequential when the network is not the bottleneck. Such systems are often proposed today even for the large scale [13], [4] and have long been economically viable at moderate scale. Future trends point to advances in networking that will outpace advancements in disk drive technologies thus eventually making data locality completely irrelevant [5]. Third, oftentimes data locality is easily achievable without replication. In the collocated case, data locality is trivially obtained by distributing data evenly across exactly the same set of nodes that perform computations. Thus, each node will have plenty of local data to compute on and little or no remote access is

required. Moreover, researchers have also proposed improving data locality with smart scheduling decisions [26]. Even when none of the above applies, the benefits of data locality may not necessarily offset the overhead of replication.

Benefits to speculative execution are limited Replication may also benefit the speculative execution of mappers. If a slow mapper needs to be duplicated (or restarted), the duplicate can read its input from another replica, potentially bypassing the problem that hampered the initial task. This benefit only applies when the slowness is caused by inefficiencies in reading input data (bad drives, slow network transfers). If the slowness is computation-related, then speculative execution may succeed even in a single-replicated system. Still, the benefits of speculative execution should not be overestimated. Studies show that up to 90% of speculatively executed tasks provide no benefits [10] because it is hard to gather enough information to understand the causes of stragglers [6].

B. Indirect costs of replication

Apart from increased job running time, replication also has several less obvious disadvantages. First, replication in one job indirectly affects concurrently running jobs by increasing disk and network contention. Second, replication increases the costs necessary for provisioning a cluster that can sustain a given job execution rate because extra compute nodes or disks are necessary to compensate for the overhead of replication. Third, replication makes scaling difficult in collocated environments. Future projections show that the number of cores in a commodity compute node will increase significantly but this trend will not be matched by a similar increase in the throughput of commodity disk drives [5]. The only way to increase local I/O throughput will be increasing the number of disk drives. This trend is already challenging current solutions for cooling and chassis design with as many as 24 disks being installed in one compute node [18]. The extra overhead of replication further aggravates this unsustainable trend.

IV. Achieving efficient recomputations with RCMP

Next, we detail the design and capabilities of RCMP. To provide a theoretical quantification of the magnitude of the challenges and of RCMP's benefits this section uses a simple model of the environment and of the MapReduce paradigm. We assume N compute nodes each having S mapper and S reducer slots. Each node runs W_M waves of mappers and W_R waves of reducers. Each compute node runs the same number of tasks and each task performs the same amount of work. We make these assumptions only to simplify illustration. RCMP does not need them. We further assume that the key-value pairs lost on failure can be traced back to the reducer that created them. Today this is easily achieved by dividing the job output file into separate partitions with one partition per reducer [25].

A. Overall system design

We now present the design of RCMP using Figure 3 as the illustration. RCMP extends Hadoop's design with advanced functionality necessary for efficient recomputation.

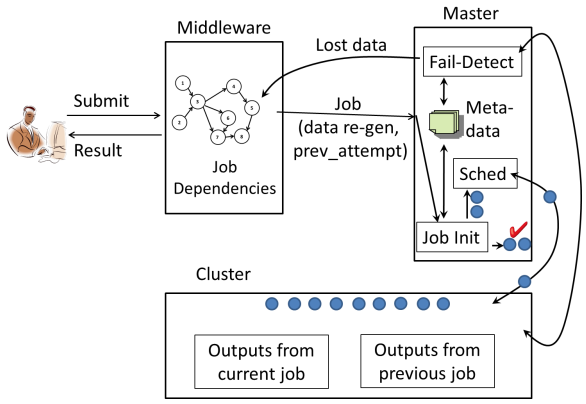


Fig. 3. RCMP system overview.

The initial job submission is similar to Hadoop. We describe it here for completeness. The user submits the multi-job computation and describes the job dependencies. A middleware program uses the dependencies to decide the order of job submission. A job is submitted only after the jobs that it depends upon are successfully computed. The jobs are submitted to the Master one by one. The Master possesses no knowledge of job dependencies and knows only how to run individual jobs to completion. Upon receiving a regular job (not recomputation), a job initialization component (JobInit) in the Master creates the tasks (circles in Figure 3) that need to be executed and the scheduler assigns them to cluster nodes.

Job execution is modified in RCMP. RCMP recognizes that during the computation of a job a significant amount of data (map and reducer outputs) needs to be materialized anyway for the job to complete. RCMP persists this data across jobs to benefit potential future recomputation, effectively trading-off storage space for recomputation speed-up. The rationale is that in the common case, failures are likely to lead to the loss of only a small portion of a job's persisted data. Therefore, most of the data persisted on an initial run can be reused to minimize the work performed on recomputation.

Upon failure detection, RCMP is much more advanced.If those failures cause irreversible data loss (all replicas of some data are lost), then the Master informs the middleware which files (job outputs) were affected and also which specific reducer outputs were affected. The middleware then immediately cancels the currently running job since it cannot complete without the lost data. The middleware uses the job dependency information and the affected files to infer which jobs need to be recomputed and in which order so that the lost data is regenerated. When submitting a recomputation job, the middleware tags it with the reducer outputs that need to be recomputed and with the job IDs of any previous successful attempts to compute this job. If a new failure occurs while RCMP is recovering from a previous one, RCMP's behavior remains unchanged. It interrupts the currently running job and starts recomputation. RCMP need not recover from each failure separately. A recomputation job can service any number of data loss events. RCMP only needs to be careful and tag the submitted recomputation job with the reducer outputs damaged by all failures.

JobInit uses the tagged information to decide what persisted data to consider for job recomputation. JobInit checks the metadata on the list of already persisted map outputs and readies for execution only the minimum necessary number of mappers. Most persisted mappers are reused. They are treated as if they had already finished. See end of §IV-B1 for one subtle exception. Concerning reducers, JobInit readies for execution only the reducers for which the outputs were affected. Note that on recomputation, RCMP significantly departs from Hadoop. Hadoop does not decide which lost data actually needs to be recomputed because it does not understand the notion of a recomputation job. It treats each job submitted to the system as a brand new job and re-executes it entirely.

RCMP performs job recomputations at the granularity of tasks. It is possible to optimize further and use a per-record granularity but we believe that this makes the system unnecessarily complex. For example, under a single failure, recomputed mappers would ideally only do a small portion of the initial records (1/N for a balanced computation), strictly the amount necessary for the 1/N recomputed reducers. However, it is difficult to make mappers skip specific input records because the reducer destination of each map output record is only decided after the map function is applied to the record.

B. RCMP during recomputation

To give a sense of the benefits that RCMP provides during recomputation by reusing persisted data, consider that after a single node failure, RCMP only needs to recompute 1/N of the mappers and 1/N of the reducers. This also results in 1/N of the shuffle traffic compared to the initial run of the job. If the 1/N mappers took W_M waves in the initial run, they can now be recomputed in $ceil((W_M * S)/((N-1)*S)) = ceil(W_M/(N-1))$ waves if they can be distributed over all compute nodes. The same holds for the W_R waves of reducers.

While these are important performance benefits, RCMP's uniqueness stems from how it efficiently executes recomputation jobs. This provides additional, significant performance benefits. This subsection details the challenges in providing efficient recomputation and the way RCMP tackles them.

1) Maximizing the use of compute nodes for recomputation: Ideally, all available nodes will be used for recomputations. RCMP's case is challenging. Because RCMP may end up recomputing a fraction of a job's tasks there is a real danger that these tasks may be too few to fully utilize all available nodes.

One may attempt to configure a job so that it is efficient on recomputation. However, this is bound to be inefficient in the failure-free case. For example, to efficiently recompute after single failures, each of the N nodes should run N-1 reducers so that each of the surviving N-1 nodes helps with recomputation. This results in $W_R = (N-1)/S$ reducer waves in the failure-free case. If W_R is high (e.g $N = 100, S = 10$) then, in failure-free runs, time is wasted performing a shuffle for each wave of reducers. One can keep W_R small by increasing S (e.g. $N = 10, S = 10$) but then performance is impacted because too many reducers are running concurrently on each node and contend for resources.

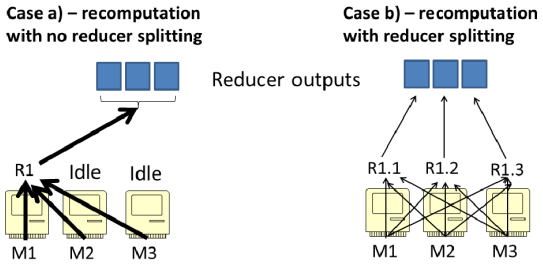


Fig. 4. Maximizing resource use for recomputation using reducer splitting.

The root of the problem thus lies in the task scheduling granularity. A coarser granularity is often desired for the initial run because it simplifies scheduling, offsets task start-up and shut-down costs and can improve performance while using resources efficiently because many tasks need to be executed. Unfortunately, the same coarse granularity can severely under-utilize nodes under recomputation when only a few tasks need to be recomputed. Consider the case when $W_R * S << N$ (i.e the total number of reducers ran by a node for a job is smaller than the total number of nodes used). In this case, reducer slots will be severely under-utilized during recomputations. After a single failure, most nodes (i.e. $N - W_R * S$) will run no reducers. This case is the norm today because it is more efficient to set the number of reducers so that W_R is 1. This allows the shuffle phase to overlap with the map computation [25]. Thus, using the task granularity from the initial run for recomputations has profound negative implications. The job recomputation time, instead of being bounded by the number of available nodes, ends up being bounded by the impact of the failure (i.e. the number of tasks affected by failure) and the job configuration (which dictates the number of tasks per node).

RCMP addresses this resource usage challenge by switching to a finer-grained task scheduling granularity only during recomputations. RCMP effectively balances the benefits of the two types of task granularities using each when it is most efficient. Note that RCMP differs from most big data processing systems today which use a single, static task scheduling granularity defined at job configuration time. RCMP's approach is to split tasks that belong to recomputation jobs. We focus on reducer splitting because mappers are less likely to under-utilize resources since they are usually far more numerous and there is no negative side-effect to having $W_M >> 1$. Nevertheless, mappers can be trivially split since each record is usually processed individually. Reducer splitting works as follows. An initial reducer is responsible for a number of keys. During recomputation the keys are simply divided among the multiple splits of the reducer. Each split still is responsible for all the value belonging to one key and this ensures computations correctness. Users should configure RCMP to split reducers only if the application logic allows it. For example, a reducer performing a top-k computation may not be split. Fortunately, such cases are rare.

Figure 4 illustrates the benefits of task splitting using a recomputation job during which one single reducer (R1) needs to be recomputed. The mappers M1, M2 and M3 have already been recomputed. In case a) reducer splitting is not used and

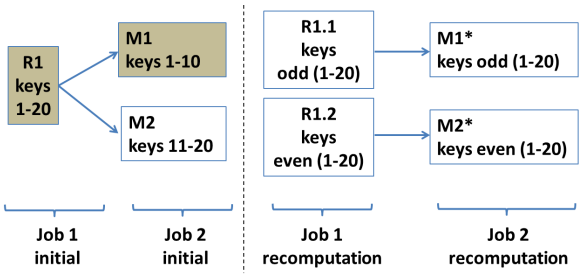


Fig. 5. Splitting correctly is not trivial. Grayed boxes represent failed tasks.

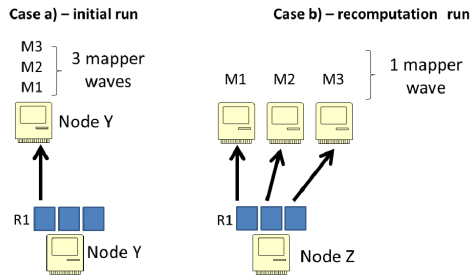


Fig. 6. Increase contention (hot-spots) on storage during recomputation.

2 compute nodes have idle reducer slots. One node has to recompute R1 entirely. With splitting (case b), the reducer work is divided among all available nodes and each split reducer contributes a portion of R1's output data. Note that reducer splitting helps not only because it better uses the available compute node parallelism but because it also load balances data transfers across more disks and network links.

Correctly performing splitting may seem trivial but is not. Figure 5 presents a subtle challenge resulting from the interaction between non-locality, splitting and data partitioning. In the initial run, mappers M1 and M2 each process half the keys from R1's output. A failure occurs and the outputs of the grayed tasks (R1 and M1) are lost and need to be recomputed. The output of M2 survived because M2 was a non-local task. During recomputation R1 is split in two (R1.1 and R1.2). The keys initially processed by R1 are now partitioned between the two splits using hash-partitioning. R1.1 processes the odd keys and R1.2 the even. It may seem that RCMP could reuse the output of M2 and not even recompute R1.2 and M2*. This would be incorrect! M2 and M2* are not the same because of the hash partitioning. Re-using M2 would cause keys 11,13,15,17,19 to appear twice in the job output, and keys 2,4,6,8,10 to never appear. RCMP solves this problem by not re-using the map outputs (such as M2) for which the reducer they depend on has been split during recomputation.

2) Avoiding hot-spots: Under recomputation there is also the danger of hot-spots when many mappers concurrently converge on one storage location to obtain their input. Consider Figure 6. Case a) illustrates an initial job run in which node Y computes reducer R1 and in the subsequent job it computes 3 mappers in 3 different waves, because it has just one mapper slot. These 3 mappers are based on R1's output. Suddenly, node Y dies and R1 and the 3 mappers (M1, M2, M3) need to be recomputed. During recomputation (case b) node Z recomputes reducer R1 but the 3 mappers based on R1's output are recomputed in 1 single wave because they are distributed over 3 surviving nodes. Since they run in one single wave, all mappers will attempt to simultaneously access node Z to get their input, thus severely increasing contention on Z.

To quantify the magnitude of the contention, consider that during the map phase of the initial run, the average number of concurrent mapper accesses on node Y's local storage is on the order of S, which is the number of mapper slots on a node. Under recomputation, the contention on node Z can be as high as $S*N$ which is the number of mapper slots over all available nodes. A network bottleneck may also appear because of the large number of simultaneous transfers.

RCMP can also use reducer splitting to mitigate the hot-spots. This works because reducer splitting distributes the reducer computation over many or all available nodes. Thus, this also implicitly distributes reducer output data over the nodes, mitigating the contention in the subsequent map phase. In effect, reducer splitting helps speed-up both the current recomputation job as well as the subsequent one.

We have also analyzed an alternative solution for mitigating hot-spots. Specifically, instead of splitting, RCMP can tell the reducers belonging to recomputed jobs to spread their output over many nodes. This solution also balances the mapper accesses in the subsequent job but compared to reducer splitting, it does not have the added benefit of dividing reducer output writing or shuffle work among several nodes. As a result, its capability to lower job running times is reduced, especially when the shuffle phase is significantly more expensive than the map phase. In this case, speeding up just the map phase may not improve overall job running time because the shuffle will still be the bottleneck. This shuffle-bottleneck can appear when only a small fraction of the mappers need to be recomputed or when the network is slow. In both cases the cause is that the recomputed reducers need to shuffle data from *all* mappers, including the persisted ones.

C. Bounding recomputation time with replication

Combining recomputation with replication can ensure that under common failure scenarios, cascading recomputations revert only to the last replication point and not all the way to the start of the computation. We have also implemented this hybrid failure resilience approach in RCMP by replicating the output of a job if its ID modulo a statically chosen value equals 0. While this hybrid approach is not the focus of this paper we also briefly evaluated it and the results are promising. As future work we are considering a dynamic approach that intelligently chooses between replication and recomputation using job and environment-related information.

A second benefit of this hybrid approach is allowing RCMP to reclaim storage space. After replication, RCMP could reclaim the space used for persisting outputs for the jobs that finished before the replication. While RCMP does not currently implement this feature, it is a straightforward addition to the system. In storage-constrained environments, RCMP may need to more aggressively reclaim storage space even in-between replications to make room for data required

for the job to finish. As future work we are considering an eviction policy that maximizes the speed-up that the remaining persisted outputs bring on recomputation. We plan to start by analyzing the benefits of deleting persisted outputs at the granularity of waves.

V. Evaluation

A. Methodology and computing environment

This section analyzes RCMP's benefits using experiments on two different clusters. In addition, numerical analysis is used to make extrapolations starting from the experiments.

What we compare We compare Hadoop 1.0.3 with RCMP (also based on Hadoop 1.0.3) and an intuitive strategy which we call OPTIMISTIC. Hadoop uses a replication factor of 2 or 3 (called REPL-2 and REPL-3). With a factor of 1, Hadoop cannot survive any failure. RCMP uses a factor of 1 (writes one HDFS replica locally) since it can recover by recomputing. OPTIMISTIC assumes that failures never happen so it also uses a factor of 1. On failure, OPTIMISTIC discards everything and re-starts from the beginning. The numbers for RCMP and Hadoop are from real experiments. The numbers for OPTIMISTIC are obtained using numerical analysis by combining the average job running time before and after the failures for RCMP without splitting.

The computing environment We use two clusters, STIC from Rice University and DCO from Zurich, Switzerland. STIC nodes have 8-core 2.76GHz Intel Xeon CPUs, a 10GbE interconnect and 24GB RAM. Each STIC node has only one 100GB S-ATA HDD. DCO nodes have 16-core AMD Opteron 6212 CPUs, 128GB RAM, use 10GbE and are distributed in 3 different racks. On each DCO node, a 2TB S-ATA HDD is dedicated to RCMP. All compute nodes are non-virtualized and we had exclusive access to them. In §V-D we also emulate an environment with a much slower network speed.

The multi-job computation used We built a custom 7-job, I/O-intensive, chain computation. Each mapper and reducer, for every input record, performs two computations which help us check correctness. One is based on the MD5 hash of a record's value while the other is based on the sum of all bytes in a record value. In addition, each mapper randomizes the key of each record to ensure load balancing of data across tasks for every job. RCMP is geared towards I/O-intensive computations. The exact computation performed by the tasks is inconsequential for the message of the paper as long as the I/O-intensive nature is preserved.

Our job has a ratio of input/shuffle/output size of 1/1/1. This ratio is in between the range of ratios encountered in practice [7], [11]. It is the same ratio used for sorting, a popular barometer of cluster performance. The relative benefits of RCMP vs Hadoop are expected to increase when the job output is relatively larger compared to the input and shuffle (i.e. ratios of the form $x : y : z$ where $z > y$ and/or $z > x$ encountered in jobs like Pig Cogroup or creating a web index [7]).

The 7-job computation uses randomly generated, triple replicated, binary input data. The HDFS block size is 256MB. On STIC *each node* processes 4GB (16 mappers of 256MB). On DCO *each node* processes 20GB(roughly 80 mappers).

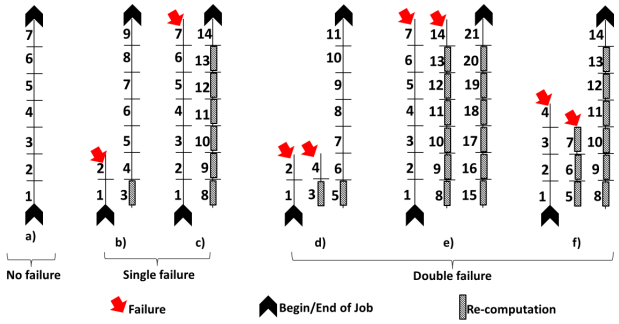


Fig. 7. The different moments at which failures are injected and their effects on the multi-job computation when using RCMP.

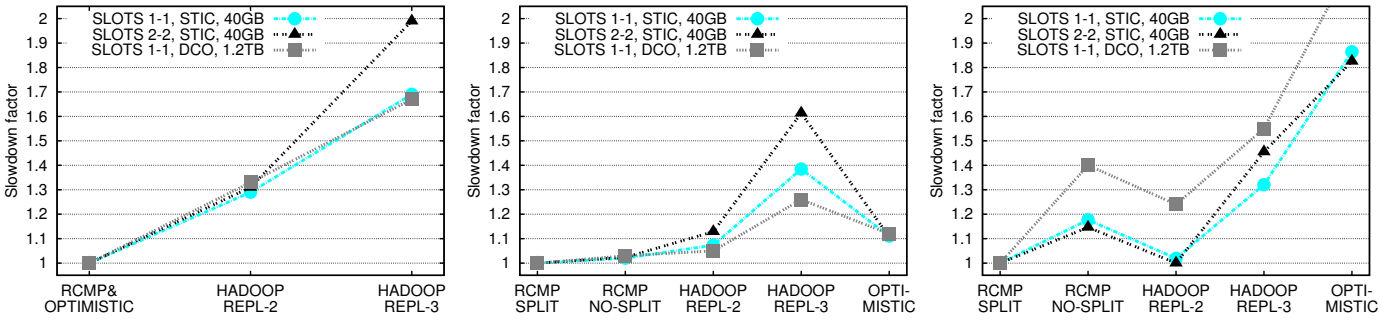
For STIC the results are averages over 5 runs of the 7-job computation. For DCO we performed 3 runs. The reducer splitting ratio is chosen to use efficiently the available compute nodes under recomputation. For DCO we enabled JVM reuse in Hadoop and RCMP as disabling it unnecessarily penalizes job performance.

How the jobs are numbered Each job (initial or recomputation) that starts running, receives as an unique ID the next available integer number starting with 1. Re-computations increase the total number of jobs ran. For an illustration consider Figure 7, case c). A failure occurred during the 7th job. As a result, RCMP recomputes the first 6 jobs and then restarts the 7th. In this case RCMP started a total of 14 jobs; each of the different 7 jobs was started twice. On the other hand, since Hadoop uses replication for failure resilience it always starts a total of 7 jobs.

How failures are injected We inject failures by killing both the Hadoop TaskTracker and DataNode processes on a randomly chosen compute node. We injected failures 15s after the start of some job. The only exception is when we inject two failures in the same job. Then, the second failure is injected 15s after the first one. Both Hadoop and RCMP have been configured with failure detection timeouts of 30s. Thus, a first failure is detected roughly 45s after the job start.

For RCMP, we chose the moments to inject failures as follows. We do not inject failures during the first job since its input is replicated. Case b) in Figure 7 represents a single failure impacting the computation early. RCMP recomputes 1 job. In case c), the failure impacts the computation when it is close to completion. RCMP recomputes 6 jobs. Case d) shows double failures injected early while in case e) the failures are injected when the multi-job approaches completion. Case f) shows a nested failure: the second failure occurs while recomputation is still being performed to address the first failure. For Hadoop we inject failures at jobs 2 or 7.

On the efficiency of RCMP's implementation For simplicity, for the job during which the failure occurs, RCMP currently discards the partial results computed before the failure. Thus, the 45s taken by RCMP to react to one failure are pure overhead. Ideally, RCMP would freeze the affected job, recompute and then reuse the partial results after restarting the frozen job. Hadoop does not suffer from this inefficiency as it uses replication. Thus, if we had set the failure detection timeout to more than 30s, or if we injected failures later in a



(a) No failure. (b) Single failure early (at job 2). (c) Single failure late (at job 7).

Fig. 8. RCMP vs Hadoop vs OPTIMISTIC. SLOTS X-Y means X mapper and Y reducer slots per node. DCO uses 60 nodes for a total of 1.2TB job input. STIC uses 10 nodes for 40GB job input. The split ratio is 59 for DCO and 8 for STIC. Results are normalized to the fastest run in each experiment.

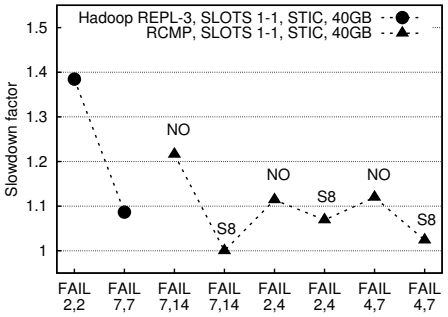


Fig. 9. Hadoop vs RCMP under double failures. NO = no reducer splitting. S8 = split in 8.

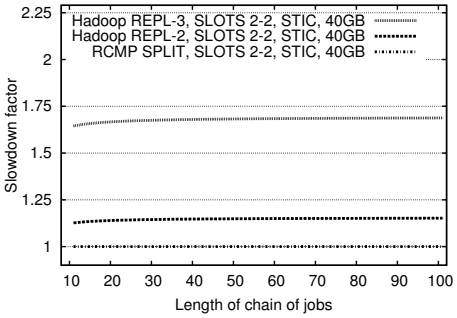


Fig. 10. The impact of a larger chain length (failure at job 2). Numerical analysis.

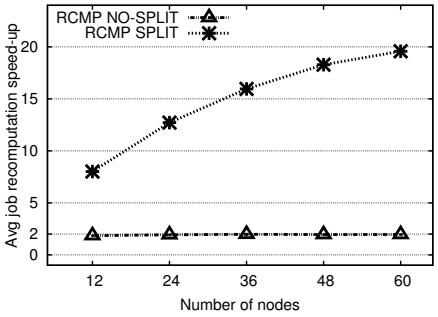


Fig. 11. Reducer splitting efficiently uses the available compute nodes for recomputation.

job, then RCMP would be at an even greater disadvantage.

B. Overall system comparisons

No failure Figure 8a shows that RCMP provides significant benefits in the failure-free case across both clusters. Hadoop REPL-2 is 30% slower while REPL-3 is 65%-100% slower. OPTIMISTIC is on par with RCMP since neither uses replication. Combining REPL-3 with 2 mapper and 2 reducer slots per node (SLOTS 2-2) causes too much contention on STIC and leads to performance degradation.

Single failure Figures 8b and 8c describe single failures. Even under failure RCMP is still fastest. The gap between RCMP SPLIT and NO-SPLIT is larger when the failure is injected at job 7 because more job recomputations are performed. For each of these recomputed jobs RCMP NO-SPLIT uses one node for the reduce phase while RCMP SPLIT distributes reducer work over many nodes. OPTIMISTIC is very inefficient when the failure occurs late (2.23x slower) because it nearly runs the same job twice. For the case STIC SLOTS 1-1 in Figure 8c we can also showcase the benefits of RCMP using the hybrid strategy that combines replication (factor of 2 once every 5 jobs) and recomputation. Though not plotted this would appear as 0.93 in the figure.

Double failures Figure 9 shows the results for double failures using 10 nodes on STIC. FAIL X,Y means two failure are injected one at job X and one at job Y. Here we only compare RCMP against Hadoop REPL-3 because REPL-2 cannot protect against all double failures. Hadoop performs better when the failures are injected late since only a small portion of the computation needs to be executed with the fewer remaining nodes. However, it is challenging to assess under which sequence of double failures is RCMP most efficient. If the failures occur late (e.g. FAIL7,14), then RCMP needs to recompute many jobs but after the recomputation is finished few jobs will have to be fully completed with fewer nodes. If the failures occur early (e.g. FAIL 2,4), RCMP recomputes few jobs but after that many jobs will have to be completed with the fewer surviving nodes. Thus, deciding the best and worst cases depends on the speed of recomputation compared to the overhead of using fewer compute nodes.

In all runs RCMP performs well, consistently beating Hadoop REPL-3 when reducer splitting is used. Splitting benefits case FAIL7,14 the most because most recomputations occur then. RCMP successfully and efficiently handled a nested double failure (FAIL 4,7) where the second failure occurs while the RCMP is still recovering from the first.

More failures To protect against F failures with replication, $F+1$ replicas are needed. If $F+1$ replicas exist but fewer than F failures occur, then replication was unnecessarily inefficient. If more than F failures occur, the computation has to be restarted. Thus, setting the right replication factor requires guesswork. In contrast, RCMP can recover from any number of failures while performing the minimum necessary amount of recomputation work.

Longer chains We now use numerical analysis to extrapolate RCMP's speed-up when the computation has more than 7 jobs. Figure 10 extrapolates based on Figure 8b for the STIC cluster, case SLOTS 2-2. The value 1 is RCMP with splitting 8-wise. The extrapolation works as follows. For any chain length, for RCMP, the running time is a combination of

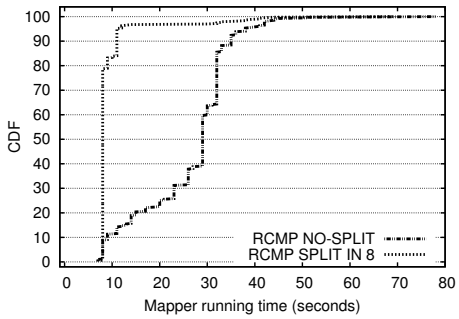


Fig. 12. Reducer splitting mitigates hot-spots and accelerates mappers. STIC - SLOTS 2-2.

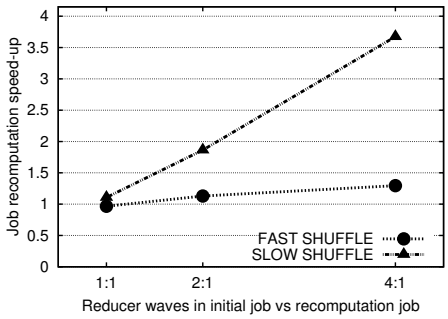


Fig. 13. Speed-up from having fewer reducers waves during recomputation.

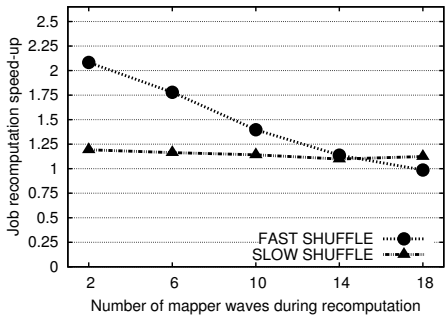


Fig. 14. Speed-up as a function of the number of mapper waves during recomputation.

jobs running with 10 nodes before the failure, with 9 nodes for recomputation and with 9 nodes after the recomputation finishes. All these jobs appear in the experiments with the 7-job chain computation and we use the averages from those. For Hadoop the extrapolation is similar only there is no recomputation to consider. In both cases we also account for the job during which the failure occurred.

RCMP's benefits are stable regardless of the chain length and match well the values in Figure 8b. This is because when the failure occurs early, the speed-up provided by RCMP is basically the ratio of how fast Hadoop runs a job with 9 nodes and how fast RCMP does the same. Similarly (not pictured) the speed-up for longer chains when failures occurs at the last job is also very stable and matches the values in Figure 8c. In that case, the speed-up is defined by the ratio of how fast Hadoop runs a job with 10 nodes and how fast RCMP does the same plus recomputing the same job with 9 nodes.

C. Breakdown of RCMP specific benefits

Recomputing using all available nodes Next, we vary the number of DCO nodes while keeping per-node work constant (20GB of data). After a failure, the 20GB on the failed node are recomputed. We want to quantify the benefits that RCMP can extract from recomputing using more nodes. The reducer split ratio is N-1 where N is the number of nodes.

Figure 11 shows how fast a job is recomputed compared to the initial run of that same job. Without reducer splitting (RCMP NO-SPLIT), there is little benefit to having more nodes because one compute node needs to fully recompute the reducer that was on the failed node. The rest of the nodes have idle reducer slots. Small benefits may be obtained when increasing the number of nodes even in the RCMP NO-SPLIT case, because the map phase is recomputed in fewer waves. Splitting provides significant benefits. Recomputation is able to benefit much more from an increase in the number of nodes, as each node performs a diminishing amount of reducer work.

Mitigating hot-spots Figure 12 shows the negative effects of hot-spots in the recomputation runs from Figure 8c when RCMP uses 2 map and 2 reduce slots (SLOTS 2-2) per node on STIC. All nodes used for recomputation attempt simultaneously to read the map input from one node. This significantly increases mapper running time. Reducer splitting mitigates contention and in the process also improves reducer running time. At the median a reducer took 103s without splitting and 53s with splitting.

D. Speed-up from recomputing with fewer waves

Having analyzed splitting we turn to the other important source of speed-up for RCMP: the reduction in the number of waves during recomputation compared to an initial job. We present two cases. One is the high-bandwidth STIC environment used so far (we call it FAST SHUFFLE). In the second, we emulate a bottlenecked-network by introducing a 10s delay at the end of each shuffle transfer (SLOW SHUFFLE). Splitting is not used. We inject a single failure at job 7.

For reducers To isolate the benefits of the reducer phase recomputation, no map outputs are reused. All mappers are recomputed. We vary the number of reducer waves in the initial run (1, 2, 4) by varying the total number of computed reducers (10, 20, 40) and keeping the number of reducer slots to 1. For the recomputed jobs all recomputed reducers (1, 2 or 4) fit in 1 wave.

Figure 13 shows the results. Recall that a shuffle phase is performed for every reducer wave but only the first reducer wave overlaps with the map phase. For SLOW SHUFFLE, the speed-up increases linearly with the decrease in the number of reducer waves recomputed. This is because the map phase is insignificant compared to the bottlenecked shuffle phase and thus each reducer wave in the initial run takes roughly the same amount of time to complete. In comparison, for FAST SHUFFLE, the speed-up increases sub-linearly because the first reducer wave is more time-consuming that the rest.

For mappers We now isolate the impact of the map phase by having 1 reducer wave during both the initial run and recomputation. Figure 14 shows that for SLOW SHUFFLE, no matter how fewer mapper waves execute during recomputation the speed-up barely increases. Finishing the map phase faster does not decrease the time necessary to complete the network-bottlenecked shuffle. On the other hand, for FAST SHUFFLE, the shuffle finishes shortly after the last map output is computed. This results in a near-linear increase in speed-up with a decrease in the number of mapper waves recomputed.

VI. Related Work

Failure resilience for big data RDDs [27] are a general purpose, distributed memory abstraction for sharing data between applications. RDDs provide fault-tolerance by logging

the transformations used to build a dataset and using this lineage information for recovery. There are several important differences between RDDs and RCMP. First, RDDs are geared towards applications that can fit most of their data in memory while RCMP focuses on the general case where data may not fit in memory and needs to be written to stable storage. Second, the lineage information allows RDDs to determine *what* to recompute on failure. RCMP also determines *what* to recompute but goes beyond that by focusing on *how* to recompute. That is, RCMP is designed to address specific challenges faced when performing recomputations: maximizing resource use and mitigating hot-spots. RDD does not deal with such challenges. Third, we quantitatively analyze the overhead of replication and the benefits of recomputation, while the work on RDDs only briefly mentions that replication may be expensive. Note that RDD also mentions the term "efficient fault tolerance", but does so when comparing against solutions that use shared-memory as a distributed memory abstraction which are expensive to provide failure resilience for. For RCMP, "efficient" means recomputing as fast as possible.

FTopt [24] is a cost-based fault-tolerance optimizer for parallel data processing systems. FTopt automatically selects the best strategy for each operator in a query plan in a manner that minimizes the expected processing time with failures for the entire query. FTopt focuses on three failure resilience strategies: NONE, MATERIALIZE (akin to replication) and CHCKPT (checkpoint operator state). FTopt does not provide insights into the benefits and challenges of recomputation.

Re-using previously computed results There is also related work on optimizing computations by leveraging previously computed results. Some big data computations are amenable to such optimizations because they have similarities in computation (shared sub-computations) [14] and similarities in input (same input or a sliding window of the input data) [9]. The challenge faced by this related work is determining and maximizing the opportunities for data reuse. While RCMP also reuses previously computed task outputs it does not face the same challenges because under failures it needs to perform the same computation on the same input. RCMP goes beyond data reuse and focuses on how to best recompute data that cannot be reused. At a higher level, RCMP's focus is also different. RCMP deals with the problem of providing failure resilience for applications while prior work in this area focuses on improvements in performance and storage utilization.

In this space, Nectar [14] automates and unifies the management of data and computation in data centers. Data and computation are treated interchangeably by associating the data with the computation that produced it. Thus, duplicate computations can be avoided by reusing cached results. Nectar uses fingerprints of the computation and the input to determine similarity to previous runs. A cache server allows the lookup of stored entries based on the fingerprints. Nectar automatically and transparently rewrites programs to cache intermediate results and to take advantage of the cached results.

VII. Conclusion

RCMP is a system that uses efficient recomputation as a first-order failure resilience strategy for big data analytics. RCMP is geared towards multi-job, I/O-intensive computations. It leverages previously persisted outputs to speed-up recomputed jobs but more importantly, during recomputations, it ensures that compute node parallelism is maximized and hot-spots are mitigated. RCMP's benefits hold across two different clusters, and for job inputs as small as 40GB or as large as 1.2TB. Not using data replication makes RCMP significantly faster during failure-free periods. More importantly, by efficiently performing recomputations, RCMP is competitive even under single and double data loss events.

References

[1] Contrail. http://sourceforge.net/apps/mediawiki/contrail-bio.

[2] Failure traces from Rice University clusters. http://riceclusterfailure traces.weebly.com/.

[3] Petabyte-scale Hadoop clusters. http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS 2011*.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI 2010*.

[7] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *SOCC 2013*.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC 2010*.

[9] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *SOCC 2011*.

[10] E. Bortnikov, A. Frank, E. Hillel, and S. Rao. Predicting execution bottlenecks in map-reduce clusters. In *HotCloud 2012*.

[11] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Eurosys 2012*.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP 2003*.

[13] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[14] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI 2010*.

[15] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters.

[16] H. Lim, H. Herodotou, and S. Babu. Stubby: a transformation-based optimizer for mapreduce workflows. In *VLDB 2012*.

[17] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3), May 1998.

[18] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI 2012*.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD 2008*.

[20] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL 1989*.

[21] A. Rasmussen, M. Conley, R. Kapoor, V. T. Lam, G. Porter, and A. Vahdat. Themis: An i/o efficient mapreduce. In *SOCC 2010*.

[22] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *ISPASS 2010*.

[23] 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. In *VLDB 2009*.

[24] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD 2011*.

[25] T. White. Hadoop - the definitive guide. O'Reilly Media, 3rd ed., 2012.

[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.