Technical Report

UCAM-CL-TR-888
ISSN 1476-2986

Number 888

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Recomputation-based data reliability for MapReduce using lineage

Sherif Akoush, Ripduman Sohan,
Andy Hopper

May 2016

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Recomputation-Based Data Reliability For MapReduce using Lineage

Sherif Akoush, Ripduman Sohan and Andy Hopper

Abstract

Ensuring block-level reliability of MapReduce datasets is expensive due to the spatial overheads of replicating or erasure coding data. As the amount of data processed with MapReduce continues to increase, this cost will increase proportionally. In this paper we introduce *Recomputation-Based Reliability in MapReduce (RMR)*, a system for mitigating the cost of maintaining reliable MapReduce datasets. RMR leverages record-level lineage of the relationships between input and output records in the job for the purposes of supporting block-level recovery. We show that collecting this lineage imposes low temporal overhead. We further show that the collected lineage is a fraction of the size of the output dataset for many MapReduce jobs. Finally, we show that lineage can be used to deterministically reproduce any block in the output. We quantitatively demonstrate that, by ensuring the reliability of the lineage rather than the output, we can achieve data reliability guarantees with a small storage requirement.

1 Introduction

Large-scale data warehouse architectures maintain multiple petabytes of data stored in distributed file systems (e.g., HDFS [1] and GFS [2]). The reliability of data in these architectures is asserted by data redundancy techniques such as *block replication* or *erasure codes*.

Block replication makes identical copies of data blocks onto other devices, operating on the premise that if the device or host fails the data can be retrieved from a replica. Typically production systems will employ at least 3-way replication to mitigate against data loss. Alternatively, erasure codes leverage polynomial interpolation to provide equivalent reliability guarantees to block replication but with significantly lower storage requirements [3, 4, 5].

The choice of whether to use block replication or erasure codes can be reduced to trading off space overhead for recovery overhead. While block replication incurs a much higher spatial footprint than erasure codes, recovery overheads in block replication are far lower. Creating a new copy of a data block in the replication case consists of a single block read and write whereas in the erasure code case multiple blocks need to be read and decoded for a new block to be generated. In fact, the data and network transfer overheads of a (10,4) Reed-Solomon (RS) erasure code [6] are 10 times more than for block replication.

Recently, the idea of *data recomputation* has been introduced as an alternative technique for ensuring data recovery [7]. Recomputation trades off storage for compute; outputs are not duplicated on the basis that they can be reproduced by rerunning the original computation in the event of data loss. In cases where output datasets dominate inputs (e.g., data transformation and

	3-way Rep.	Encoding RS(10,4)[a]	Rerun	RMR (Sort Job)[b]	
Storage Overhead	200%	40%	0%	11%[c]	5%[d]
Block Recovery Overhead					
Disk I/O	1x	10x	High	1x	8x
Network I/O	1x	10x	High	1x	1x
Time (64 MB Block)	Low	20 s[e]	High	26 s	
Lazy[f]	No	No	Yes	Yes[g]	
Job Runtime Overhead	No	No	No	8%[h]	
Generic	Yes	Yes	No	No[i]	

[a] Reed-Solomon Encoding of 10 Blocks into 4 Parity Blocks.
[b] For a Sorting (1:1 Data Input-to-Output Ratio) MapReduce Job.
[c] Record-Level Lineage. Lineage is Encoded using RS(10,4).
[d] Sector-Level (256 Bytes) Lineage. Lineage is Encoded using RS(10,4).
[e] From Previous Studies [5].
[f] Without Affecting Data Reliability.
[g] In the Case of Cold Data Output.
[h] The Runtime Overhead to Capture Lineage during the Main Job Execution.
[i] Suitable for Data Transformations and Joins.

Table 1: First-Order Comparison of Data Reliability Techniques.

expansion jobs) [8, 9], this technique can realize huge storage savings as only a small fraction of the overall data footprint requires redundancy (the inputs). We observe, however, that naive recomputation-based recovery has an inordinately large overhead as recovering even a single block of the output requires a re-execution of the entire computation. Additionally, recovery time is in the order of the *total* running time for *all* the computations needed to recover blocks contained in failed nodes [10].

In this paper we introduce *Recomputation-Based Reliability in MapReduce*(RMR), a recomputation-based data recovery system specialized for MapReduce. We focus on MapReduce due to its popularity as a data processing platform in many compute workflows [11, 12, 13]. RMR (*i*) provides storage efficiency close to naive recomputation and (*ii*) recovers failed blocks with manageable network and disk I/O resource requirements.

RMR is based on the principle that, by collecting precise fine-grained lineage (i.e., links between input and output records) during the original MapReduce job execution, we can speed up data recovery by only re-executing the computation with a subset of the input that is needed to reproduce a missing output block. RMR imposes low spatial and temporal overheads for lineage capture – less than 10% for many MapReduce jobs.

Table 1 summaries the advantages of RMR as compared to block replication and erasure codes. RMR has a lower spatial overhead, and lower network and disk resource requirements for block recovery. While recovery time is large compared to block replication, it is comparable to the erasure code case.

The rest of this paper is organized as follows. Section 2 gives an overview on common data reliabilities techniques. We then illustrate the design of RMR in Section 3 and its suitability in Section 4. We describe our prototype implementation for Hadoop MapReduce in Section 5. Section 6 quantifies RMR overheads for 10 MapReduce jobs and Section 7 presents associated trade-offs. Finally we discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section we outline the details of block replication, erasure codes and data lineage pertinent to the understanding of the design of RMR.

2.1 Data Reliability And Recovery

Block replication is the most prevalent data reliability technique used in modern datacentres. In particular, 3-way replication is commonly employed in production systems [] making it possible to tolerate up to two failures before there is underlying data loss. Of the three copies two are usually co-located on the same rack while the third is placed off-rack thereby providing tolerance to rack failure. Block replication also possesses the attractive property of supporting read-parallelism. Data accesses may be routed to any one of the replicas. Many services will increase the replication factor if higher tolerance to failure or increased read-parallelism is required [14].

With increasing dataset size replication is becoming infeasible due to the fact that storage overhead is proportional to the product of the dataset size and the number of replicas required. To this end erasure codes have gained popularity as a mechanism to provide equivalent data reliability guarantees as block replication but with lower storage overhead.

Erasure codes leverage polynomial interpolation to encode k blocks into n extra parity blocks with the property that the loss of up to *any* n blocks of the entire (k + n) set (or *stripe* as it is commonly known) can be tolerated without incurring fundamental data loss. Acceptable values of k and n are dependent on the type of erasure code used, the design of the storage system, and the level of reliability required. In general k is varied according to the number of distinct hosts on which blocks belonging to a stripe will be stored while n is varied according to the maximum number of simultaneous failures tolerable.

Current systems are usually based on RS erasure codes as they are well understood for encoding data blocks [6]. Taking the example configuration of RS (10,4), the system can tolerate up to four block failures per stripe without incurring any underlying data loss at the cost of 40% additional storage overhead.

While erasure codes provide significant storage savings, data recovery overheads are significantly higher than block replication. Recovering a single data block requires carrying out a calculation over k blocks from the stripe. In practice it has been shown that normal data recovery activity in a compute cluster can place a significant burden on resources. Facebook, for example, has reported that it incurs 180 TB of cross-rack data recovery related traffic per day [15].

It has also been observed that more than 98% of all recoveries are for single blocks in a stripe [15]. To optimize this case *local repair codes* specialize parity blocks into *local* and *global* parity blocks. For a given stripe, local parity blocks are co-located near and encode a subset of its data blocks while global parity blocks are calculated across all its data blocks. Recovering from a single block failure in any subset is reduced to carrying out a calculation across the other blocks contained in the subset and the local parity block. This requires significantly fewer disk and network resources compared to standard RS codes. However, calculations across the entire stripe are still required to tolerate multiple failures [3, 4]. Local repair codes possess a higher storage footprint than standard RS codes as they maintains both local and global parity blocks.

When optimizing storage overhead is the top priority, piggyback erasure codes may be an attractive option. Piggyback erasure codes incur the same storage overheads as standard RS codes but use carefully designed functions to combine parity blocks over multiple stripes. Multiple blocks in disjoint stripes can be recovered with a subset of the parity blocks that would be required for standard RS codes. While this optimization is especially useful for reducing the resource requirements for multi-block recovery (e.g. in the event of node failure), it incurs a

higher processing overhead than standard RS codes [5].

We observe that data recovery overheads are of the same order of magnitude regardless of optimizations. Some researchers have advocated delaying recovery until multiple failures occur in the same stripe. In this case more data can be recovered per-byte of data transferred over the network thereby reducing the network overhead of coalesced repairs [16]. However, this optimization reduces the overall system reliability. Nevertheless, the idea of lazy recovery illustrates how we can manage recovery overheads dynamically. This is a useful property when the data is not expected to be accessed frequently.

In recomputation-based recovery on the other hand, only the input datasets have to be reliably stored in order to guarantee the overall data reliability of the system. However as we discussed before, naive recomputation is wasteful as it requires a significant temporal overhead to reproduce an output data block. In this paper we show how we employ data lineage to efficiently recompute these missing data blocks.

2.2 Lineage

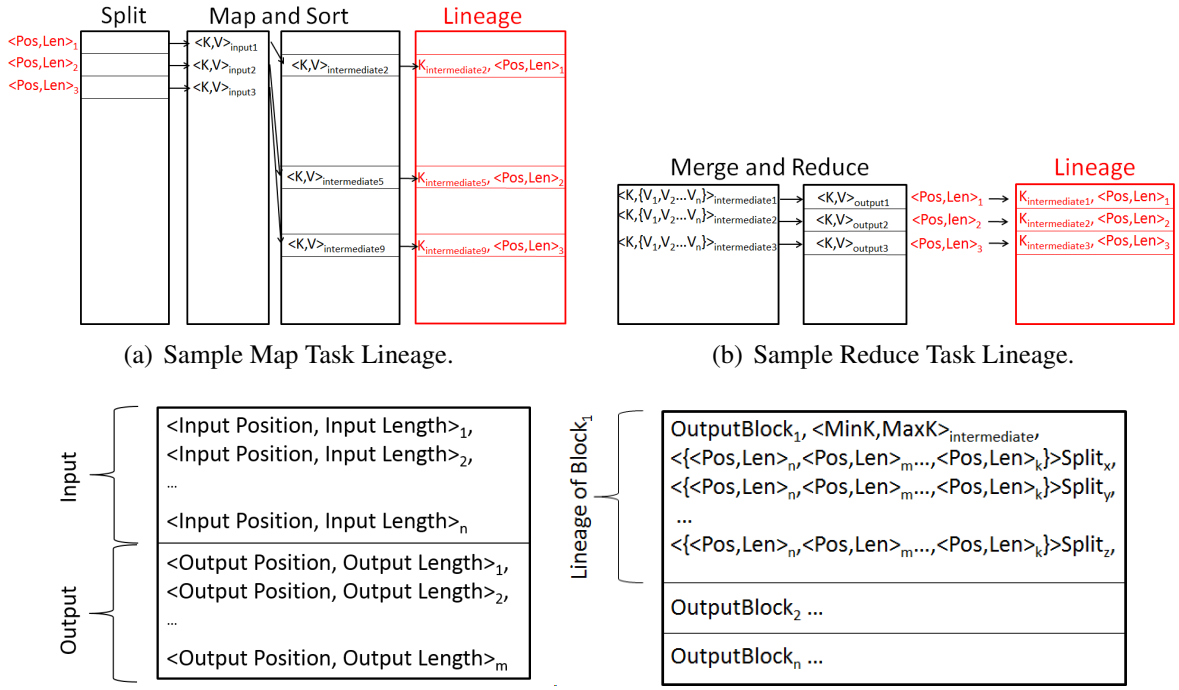
Data lineage is the causal relationship between input and output records. Given a set of input I, a transformation P, and a set of output O such as $O = P(I)$, lineage associates $i \in I$ with $o \in O$. Fine-grained record-level lineage is achieved when there is a precise association between each $o \in O$ with the subset of input $I' \subseteq I$ that affected its creation: (o, I'). This level of lineage is useful for many applications such as such as data debugging and drill-down scenarios [17, 18].

RMR leverages lineage to support data recovery via recomputation. For example, if a string is transformed from upper to lower case, there is a lineage record connecting the input record to the output record. If, at any stage, we lose the output record, we can reproduce it by reading the associated input record and reapplying the original transformation. In the next section we outline the detailed design of RMR and illustrate how it achieves recomputation-based block recovery in the MapReduce framework.

3 RMR Design

MapReduce [19] consists of 2 user-defined functions: `map` and `reduce`. Map tasks sequentially read, filter and transform key-value pair records from an input file outputting the results of the transformation as a set of zero or more intermediate key-value pair records. Intermediate records are split into equally sized blocks and sorted according to key. Finally, a reduce task combines and further transforms all intermediate records with identical keys into final key-value pair records which are written out to persistent storage. Modern MapReduce tasks are usually expressed in a high-level language such as Pig [20] or Hive [21] which are translated into low-level operations supported by the MapReduce framework.

At a high-level, RMR records fine-grained lineage connecting input and output key-value records for Map and Reduce tasks at runtime. This lineage is further encoded into an efficient and compressed form for long-term storage. When a data block requires recovery, RMR restarts a computation job based on the original Map and Reduce functions to reproduce the required block. By leveraging stored lineage RMR is able to limit their computation to only operating on the subset of records required to recreate the output block. In this manner RMR is able to provide efficient block-level data recovery.



(a) Sample Map Task Lineage.

(b) Sample Reduce Task Lineage.

(c) Record-Level Lineage.

(d) Compacted Lineage: Output Block to Input Locations.

Figure 1: Lineage in RMR

3.1 Lineage Capture

As explained in Section 2, lineage is an association between an output record and the relevant input records that affected its creation. In RMR lineage at the record-level is defined by its file offset and length as indicated in Figure 1(c). RMR is able to succinctly represent associations between input and output records as a list of input and output record offset and length tuples. This maps well onto the append-only distributed storage system such as HDFS as offsets remain stable.

Previous approaches to providing key-value pair lineage granularity to MapReduce have relied on propagating lineage information along with the record across the entire job [22]. However, this approach has an unacceptably high temporal (up to 75%) overhead and is thus unacceptable for use in production systems. Instead, RMR records the lineage for every Map and Reduce job independently linking it together via a post-processing job [23, 17]. This design choice results in low lineage capture overhead as we demonstrate in the evaluation section.

Every Map task reads key-value records from an input file, applying a user-defined Map function to emit intermediate key-value records typically sorted on the intermediate key. RMR records the mapping between each intermediate key, and the offset and length of every input key-value pairs affecting its creation. Figure 1(a) illustrates this association for an example 1:1 Map transformation.

MapReduce guarantees that all values associated with a given intermediate key are processed by the same Reduce task. When the user-defined Reduce function is applied on intermediate key-value records to produce final output key-value records, RMR stores links between the offset and lengths of output records and the associated intermediate key. Figure 1(b) shows this association for a typical Reduce task.

3.2 Job Behavior

Map-Only Jobs: RMR supports seamlessly Map-only jobs [11] with no Reduce tasks. The associations between input and intermediate records during Map tasks are materialized as lineage files for the job.

Map-Side Combination Jobs: Some MapReduce jobs contain a Map-side `combine` that carries out a local reduction on intermediate key-value records. The purpose of this phase is to minimize the amount of data shuffled to Reduce tasks. RMR is able to precisely track lineage during Combine tasks as part of Map tasks.

Multi-Stage Jobs: Additionally RMR tracks lineage seamlessly for multi-stage MapReduce jobs. If the input is itself derived from a previous computation, RMR can follow this link by matching file offsets between the Map lineage of the subsequent job with the Reduce lineage of the previous job thereby enabling lineage tracking between the source and final output in a multi-stage job.

Stateful and Non-Deterministic Jobs: While MapReduce jobs were originally designed to be stateless, the framework can be extended to include primitive support for state sharing across `map` and `reduce` functions [12]. The current design of RMR is unable to capture precise lineage for stateful jobs. As RMR places no restrictions on and has no observability into the nature of the `map` and `reduce` functions it conservatively assumes that a given output record can depend on all input records consumed prior to its emission. We highlight that the issue of observability can be mitigated by statically analyzing and augmenting user-defined functions to precisely capture state-based lineage. Non-deterministic jobs can be treated as a special case of stateful jobs. We are planning to add support for this functionality in future work.

3.3 Lineage Compaction

"Temporary" lineage captured at Map and Reduce tasks is compacted by joining on matching intermediate keys, which produces a direct association between input and output key-value records as illustrated in Figure 1(c). RMR achieves this goal by scheduling it as an *offline* post-processing task which can be executed when there are spare resources available depending on user requirements. Lineage compaction also allows the storage overhead to be reduced with no loss in the overall accuracy or precision of record-level lineage.

Lineage compaction is achieved by creating a union list linking a given output block to *all* the input key-value records required to reproduce the records contained in it as illustrated in Figure 1(d). This approach is transparent to the original job implementation.

The MapReduce framework sorts intermediate records based on their keys. RMR relies on this support to quickly $(\mathrm{O}(\log n))$ join Map and Reduce lineage files. RMR sorts the input positions associated to a given output block, which reduces lineage size by storing relative positions. Moreover this optimization enables fast processing of input records during block recovery.

During lineage compaction, RMR observes the intermediate key range of all intermediate records used to produce a given output block. It then augments the lineage of each output block with the Minimum and Maximum of this range (Figure 1(d)). In Section 3.4, we will show how RMR leverages this extra information to optimize network transfer during block recovery.

Lineage compaction is easily extended to multi-stage jobs, the only constraint being that this compaction has to be applied at the end of the entire workflow in order to be able construct

precise dependencies to source records. We expect that these workflows are expressed in Hive or Pig, which we plan to support in the future.

3.3.1 Sector-Level Lineage Compaction

By default RMR records lineage at record-level granularity meaning that lineage storage overhead is proportional to the number of input record consumed by the computation. We observe that it is possible to reduce the lineage storage footprint by maintaining pointers to records at *sector* rather than record level. For example, by recording lineage information for records in a 64 MB block at 4 KB sector boundaries as, we can reduce the number of bits required to store position offsets from 26 bits to 12 bits at the cost of reduced precision. Sector-level lineage may increase savings at the compaction stage as records within the same sector can be presented by a single entry.

The reduced precision introduced by sector-level lineage recording increases the resource requirements for block recovery. When a failed block needs to be reproduced *all* the records contained in the associated input sectors require processing.

Varying sector size trades off lineage storage overhead against block recovery overhead. Larger sector sizes reduce lineage storage overhead but increase block recovery overheads.

3.3.2 Range-Level Temporary Lineage

RMR can also be configured on the way it captures temporary lineage information during the execution of the main job. Our design choice is to minimize the time overhead on the main job, that is why we try to lift any heavy-weight lineage processing off the critical path. However this results in more temporary lineage that is stored on the distributed storage system.

We can still carry out optimizations to reduce the amount of lineage information stored during the main job execution. One possible solution is to group the lineage of successive records together. This might be useful in some situations where we know beforehand that this will not hurt lineage precision.

For example if we know that we are not expecting further jobs on the chain of computations in a workflow (i.e., last or single MapReduce job). We can coalesce Reduce side temporary lineage information for a few records. The intuition here is that the process of lineage compaction will group records encapsulated in one output block similarly. This reduces the amount of temporary storage requirement in this specific case.

However we cannot apply the same intuition for Map side temporary lineage information without losing precision. Therefore if we coalesce Map side temporary lineage information, we will end up having to read unnecessary records form the input when recover output blocks.

3.4 Block Recovery

Data blocks are recovered by replaying the original computation that resulted in their creation. RMR uses the previously collected lineage information to recreate every output record contained in the failed block. For each output record, reproduction is achieved by retrieving all associated input records followed by reapplying the original Map and Reduce functions. This mechanism is trivially extended to multiple blocks of the output datasets. By pinpointing the precise input positions relevant to the creation of the failed output blocks we optimize disk and network resource consumption during recomputation.

If lineage collection was carried out at the sector-level, it is necessary to discard output records that are not present in the block being recovered. RMR uses the intermediate key range records collected at compaction to provide this functionality. As the Map computation is replayed and the intermediate output block recreated, every intermediate key is compared against the range of keys recorded against the block at compaction. If the intermediate record has a key that is within the recorded range, this record is emitted during the Map phase, otherwise it is silently discarded. In other words, only relevant intermediate records are allowed to proceed to the Reduce task.

It has previously been reported that network requirements dominate MapReduce jobs with the major bottleneck being the throughput of top-of-rack switches [3, 15]. While increasing lineage sector size will increase the amount of data read during block recovery, RMR can still maintain the same network transfer overhead regardless of lineage sector size by employing the range check on intermediate records.

4 Workload Suitability

As the amount of lineage generated by RMR is dominated by the connectivity of inputs records to output records we highlight that the ratio of the size of the input dataset to the output dataset is a good heuristic that indicates how much additional storage will be required for maintaining lineage information.

For jobs where the input-to-output ratios is less than or equal to 1, as is the case for jobs that expand or transform input data (e.g. sort or joins), the amount of lineage information required to express relationships between input and output records is expected to be an order of magnitude smaller than the output. This is due to the fact that the low connectivity between input and output records means lineage can be recorded as a mapping between a single output block and a small set of input records.

In contrast, when the input-to-output ratio is greater than 1, lineage overheads increase in proportion to the connectivity between input and output records due to the requirement for RMR to record mappings between a large number of input records for every emitted output record. Very large input-to-output connectivity can lead to the lineage information being of equivalent or larger size than the output dataset.

Previous work has shown, however, that many MapReduce computations have low input-to-output ratios. For example, traces from a Yahoo cluster showed that up to 91% of MapReduce jobs are Map-only or Map-mostly jobs [11]. Similar results were obtained on an analysis of Facebook workloads [13].

These studies indicate that many jobs that run in current production setups can benefit from the use of RMR. Data transformations, sort and join jobs are seemingly common and because they maintain low input-to-output ratios, RMR can capture lineage with little storage overhead. Our practical evaluation in Section 6 illustrates that, in the many case, this overhead is usually an order of magnitude lower than the output datasets they augment.

RMR is designed to enable reliable and efficient recomputation-based recovery of failed blocks. Ensuring this goal requires that the input dataset and lineage information are stored reliably as we can reproduce any data block from its corresponding input using lineage. When lineage has a reasonably small footprint we use RMR to achieve similar reliability guarantees to data redundancy techniques but at a fraction of their storage costs.

	Input	Shuffle	Output	Input-to-Output	Description
WordCount (WordCnt)	280 GB	8 GB	250 MB	1,107	Count Words in Wikipedia Documents
HistogramMovies (HistMs)	280 GB	841 KB	96 B	$3x10^9$	Generate a Histogram of Average Movie Rating
SelfJoin (SelfJoin)	76 GB	74 GB	722 MB	106	Generate Associations among Fields
DBQuery (Query)	177 GB	40 GB	17 GB	11	Filter, Join and Aggregation Workload
DBJoin (Join)	177 GB	125 GB	133 GB	1.3	Join Workload
DBAgg (Agg)	162 GB	59 GB	7 GB	24	Data Projection and Summary Workload
AdjacencyList (AdjList)	145 GB	153 GB	152 GB	0.9	Generate Adjacency Lists of Nodes
SequenceCount (SeqCnt)	280 GB	260 GB	28 GB	10	Count all Unique Sets of three Consecutive Words
DBCogroup (CoGroup)	177 GB	225 GB	218 GB	0.8	Cogroup Workload
Sort (Sort)	14 GB	14 GB	14 GB	1	Sort Workload using IdentityMap and IdentityReduce

Table 2: Workload Information.

5 Implementation

We have implemented RMR in Hadoop Mapreduce v2. We augmented the core of the framework to capture record-level lineage at all the different stages of a MapReduce job. As RMR is implemented at the framework level it is transparent to user jobs; they are able to run seamlessly without modification.

Record readers and writers have to be extended to expose the position of the current record being consumed and produced respectively. In our implementation we augmented the standard text and binary (SequenceFile) readers and writers.

Although we have not yet implemented this feature, RMR has to synchronize lineage information for any modification to files stored in HDFS. For example if a user renames a file, we have to update its lineage accordingly. However we believe that it is easy to handle these changes for an append-only filesystem such as HDFS.

We further implemented the post-processing lineage compaction functionality as a number of separate jobs that construct the direct lineage graph from task-level lineage files. The recovery of a subset of the output data is achieved by the system rerunning the original MapReduce job limited to the subset of the input required to reconstruct blocks being recovered. RMR stores lineage information in plain HDFS sequence files co-located with the output data generated by the MapReduce job.

We have added around 3,000 lines of code to different parts of the Hadoop MapReduce codebase to support lineage tracking. A further 2,000 lines of code provide compaction, recomputation-based block recovery support.

6 Evaluation

In this section we provide a quantitative evaluation of the time and space overheads of RMR. We further provide a detailed discussion on the major tunable parameters and trade-off of the system. Results indicate that, in cases where jobs have low data input-to-output ratio, the prototype implementation is able to significantly reduce the storage footprint required to ensure reliable recovery of the output dataset as compared to block replication and erasure codes. The results also show that RMR is able to achieve block-level recovery with network and disk resource requirements comparable to block replication. Finally, the results provide evidence that recovery time of failed blocks are comparable to erasure codes.

We focus our evaluation on: (*i*) the runtime cost of capturing lineage on MapReduce jobs, (*ii*) the storage overheads associated with lineage capture, (*iii*) the lineage compaction over-

heads of RMR, and (*iv*) the network, disk and computation overheads of recovering a single 64 MB output block. We also compare RMR with state-of-the-art erasure codes.

6.1 Evaluation Setup

As outlined in Section 4, the effectiveness of RMR is dependent largely on the connectivity between input and output records. To gauge its effectiveness on real-world workloads we evaluate RMR on 10 MapReduce jobs selected from existing big-data benchmarks [24, 25]. We choose a number of jobs representative of the types of operations for which users employ MapReduce and of varying data input-to-output ratio. Table 2 details the complete set of workloads used to evaluate RMR in this work.

Evaluation was carried out on 21 Google Compute Engine (GCE) `n1-standard-4` Instances. These nodes possess 4 virtual cores, 15 GB RAM and 2 TB storage each. In some tests we use SSD-based storage to characterize the behavior of RMR as the performance of the underlying storage substrate varies. In all cases the results shown are the average of 5 runs. Confidence intervals and standard deviations are highlighted where appropriate. We guard against variations in the GCE environment by verifying that normalized results are consistent with results observed in a local, dedicated 7-node cluster.

6.2 Lineage Capture Overhead

As the fundamental block-level recovery functionality offered by RMR is dependent on fine-grained lineage it is important that the time and space overheads for capturing lineage are small enough to justify the use of the system. In this section we quantify the temporal and spatial overheads of lineage capture obtained by running the jobs on Hadoop with and without support for RMR.

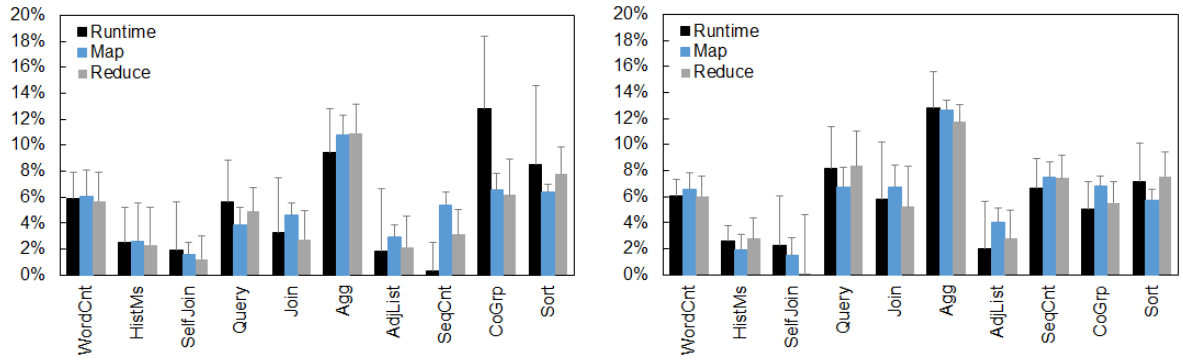
6.2.1 Temporal Overheads

Real-time lineage capture imposes an unavoidable temporal overhead. We quantify the temporal overhead of collecting record-level temporary lineage information for input to the compaction stage.

Figure 2 details the increase in MapReduce job times, for jobs executing in RMR with respect to baseline. We evaluate both HDD and SSD based configurations. In most cases the job runs a maximum of 8% slower. Both Map and Reduce tasks exhibit similar runtime overheads. We only observe one workload (Agg) that incurs more than 10% runtime overhead. This is due to the fact that the Combine stage of this workload is not able to combine any intermediate records due to the nature of the data. Tracking lineage in this workload therefore requires twice the compute resources (at Combine and Reduce stages) as other workloads.

The similar (overall) runtime overheads exhibited by both HDD and SSD-based configurations suggests that lineage collection is not I/O bound. However, the 95% confidence interval is smaller on SSD-based configurations, indicating that a faster storage substrate can be helpful in mitigating the temporal overhead of lineage capture.

A complete quantification of the temporal overhead of lineage collection includes the time required for compaction, which we discuss in Section 6.3.



(a) Using HDD Disks.

(b) Using SSD Disks.

Figure 2: Time Overheads of Lineage Capture (Average and 95% Confidence Interval) Relative to Standard Job Execution. Map and Reduce Represent Time Spent by All Map and Reduce Tasks in Occupied Slots Respectively.

6.2.2 Spatial Overheads

As explained in Section 3.1, RMR persistently stores Map and Reduce lineage for use in block recovery after compaction. In this section we quantify the spatial overhead of collecting temporary lineage.

Table 3 details the spatial overhead of temporary lineage, collected at record-level relative to the output dataset size. The results illustrate that, for aggregation jobs that reduce the data significantly (WordCnt and HistMs), the amount of storage required to store temporary lineage is unacceptably high. This is because, as explained in Section 4, lineage size is proportional to the connectivity between input and output records and these workloads exhibit very high connectivity between input and output records. For all other workloads the amount of temporary lineage created is between 19–636% of the output dataset size depending on workload.

We also evaluate the effectiveness of applying the record coalescing optimization discussed in Section 3.3.2. As the results in Table 3 show, this optimization is extremely effective at reducing the size of temporary lineage. For our sample workloads it reduces the temporary lineage space overheads of all workloads to 15–60% depending on workload.

Note that, as explained in Section 3.3, the spatial footprint of temporary lineage is reduced even more when post-process compaction is carried out. For example, of the 10 workloads in our evaluation set, 5 have a compacted lineage that is $\leq 20\%$ of the output dataset size. A full evaluation on the spatial overheads of compacted lineage is presented in the next section.

6.3 Lineage Compaction Overheads

Table 3 and Table 4 detail the spatial and temporal overheads of lineage compaction respectively for the sample workloads in the evaluation set. Compaction is evaluated on temporary lineage collected at record-level. Temporal overhead ranges between 6–34% of main MapReduce job runtime. Of the 10 workloads in our evaluation set, the temporary lineage of 4 workloads can be compacted within 15% runtime overhead while the other workloads incur below 34%. Recall, however, that lineage compaction is a one-time computation that can be scheduled at-will. It does not affect the critical path of the original MapReduce job.

	Temporary Lineage	Compacted Lineage
WordCnt	↑	↑
HistMs	↑	↑
SelfJoin	303%(144%)	63%
Query	103% (38%)	18%
Join	38% (16%)	8%
Agg	636% (266%)	104%
AdjList	19% (16%)	11%
SeqCnt	368% (263%)	132%
CoGroup	24% (10%)	5%
Sort	70% (18%)	7%

Table 3: Lineage Storage Overheads. Percentages in Parentheses Represent Range-Level Lineage as Discussed in Section 3.3.2.

	Runtime Overhead
WordCnt	32%
HistMs	11%
SelfJoin	20%
Query	6%
Join	22%
Agg	26%
AdjList	12%
SeqCnt	15%
CoGroup	22%
Sort	34%

Table 4: Lineage Compaction Runtime Overhead.

	Record-Level	64 Bytes	256 Bytes	1 KB	4 KB
WordCnt	↑	↑	↑	↑	↑
HistMs	↑	177%	148%	117%	94%
SelfJoin	63%	39%	27%	25%	22%
Query	18%	15%	11%	7%	6%
Join	8%	8%	6%	4%	3%
Agg	104%	62%	47%	44%	34%
AdjList	11%	9%	6%	4%	4%
SeqCnt	132%	82%	58%	53%	43%
CoGroup	5%	5%	4%	3%	2%
Sort	7.41%	4.56%	3%	3%	2%

Table 5: Sector-Level Lineage Overheard Relative to Output Dataset.

Table 3 also details the spatial overheads of the compacted lineage for all 10 workloads in our evaluation set. As the table shows, for most workloads lineage storage overhead is a fraction of the output dataset. The smallest spatial overhead (5%) is attributable to the CoGroup job (a workload that has a 0.8 data input-to-output ratio). Other workloads with low data input-to-output ratios (Sort, AdjList, Join) exhibit similar low overheads with respect to spatial overhead.

Conversely, workloads with high data input-to-output ratio (WordCnt, HistMs) require orders of magnitude more storage space than the output dataset size. This is expected as jobs that summarize data typically output a small amount of data highly connected to a large set of input records.

There exist workloads (Agg, SeqCnt) that do not aggressively reduce the amount of emitted data with respect to the input. Per output record, these jobs aggregate a relatively small number of input records (in the range of 10-20). Lineage for this type of job is expected to be of the same order of magnitude as the output dataset.

The Query workload is also of interest. It combines filter, join and aggregation and although its data input-to-output ratio is high, compacted lineage overhead for the job is less than 20% of the output dataset size. This is due to the fact that for jobs that do data filtering, RMR records lineage only for records that pass the filter condition. Therefore the amount of lineage associations that are required to be tracked is small.

6.3.1 Sector-Level Lineage Storage

Section 3.3.1 describes sector-level lineage compaction, an optimization that reduces the storage footprint of lineage at the expense of lower precision. In this section we evaluate the effect of increasing sector size on storage requirement.

For many workloads (Query, Join, AdjList, CoGroup and Sort), record-level lineage has a low ($\leq 10\%$) spatial footprint. However, for other workloads (e.g. SeqCnt), it can be as high as 142% of the output dataset size.

We quantified the effect of larger (64, 256, 1024 and 4096 bytes) sector-sizes on compacted lineage output for all the workloads in our evaluation set. Table 5 provides details on the lineage footprint as compared to output dataset size. Three clear observations emerge from the measured results:

(*i*) Larger lineage sector sizes are effective at reducing compacted lineage size: For all workloads in the evaluation set, larger lineage collection sector sizes resulted in a smaller overall lineage footprint. In general, the amount of lineage stored using a 4 KB sector size is approximately 33% of the lineage collected at record-level granularity. This result highlights the resource savings afforded by sector-level lineage storage.

(*ii*) Lineage size does not reduce linearly with increasing sector size: The results further show that for many workloads the reduction in lineage overhead plateaus as sector size increases. For example, with the Adjlist workload, there is no difference in compacted lineage size when lineage collection sector size increases from 1 KB to 4 KB. This result indicates that selecting lineage sector size depends on workload characteristics.

(*iii*) Larger sector-sizes are ineffective at enabling tractable lineage collection for highly connected workloads: As explained in Section 6.2, WordCnt and HistMs display very high connectivity between input and output records. While intuitively it may appear that larger lineage sector-sizes will reduce the size of the collected lineage, our results show this is not true. Even with 4 KB sector sizes lineage overhead is at least as big as the output dataset. Further evaluation showed that with a 16 KB lineage sector size, HistMs still has a lineage footprint in excess of 50% of output dataset size. This result confirms that RMR is unsuitable for efficiently supporting high data aggregation workloads.

As outlined in Section 3.3.1, larger sector sizes increase resource overheads for block recovery. We discuss the effect of larger lineage sector size on block recovery overheads in the next section.

6.4 Data Block Recovery

This section characterizes the time and resource overheads associated with reproducing failed data blocks in RMR.

6.4.1 Runtime Overhead

We measure the real-time required to reproduce a single 64 MB block of the output. Our definition of real-time includes the time required for record lookup from the compacted lineage and is measured as the time between which RMR is instructed to reproduce a block and it confirming successful reproduction. Unless stated otherwise all lineage is compacted to record-level.

Figure 3 provides the average block recovery time for all workloads in the evaluation set.[a] We first evaluated recovery time in equivalent HDD and SSD-based configurations. Results show that 5 (HistMs, SelfJoin, Add, AdjList and Sort) of the evaluated workloads are able to recover a missing block in less than 300 seconds. The rest are able to recover the block

[a] except for Query due to a miss configuration at that time.

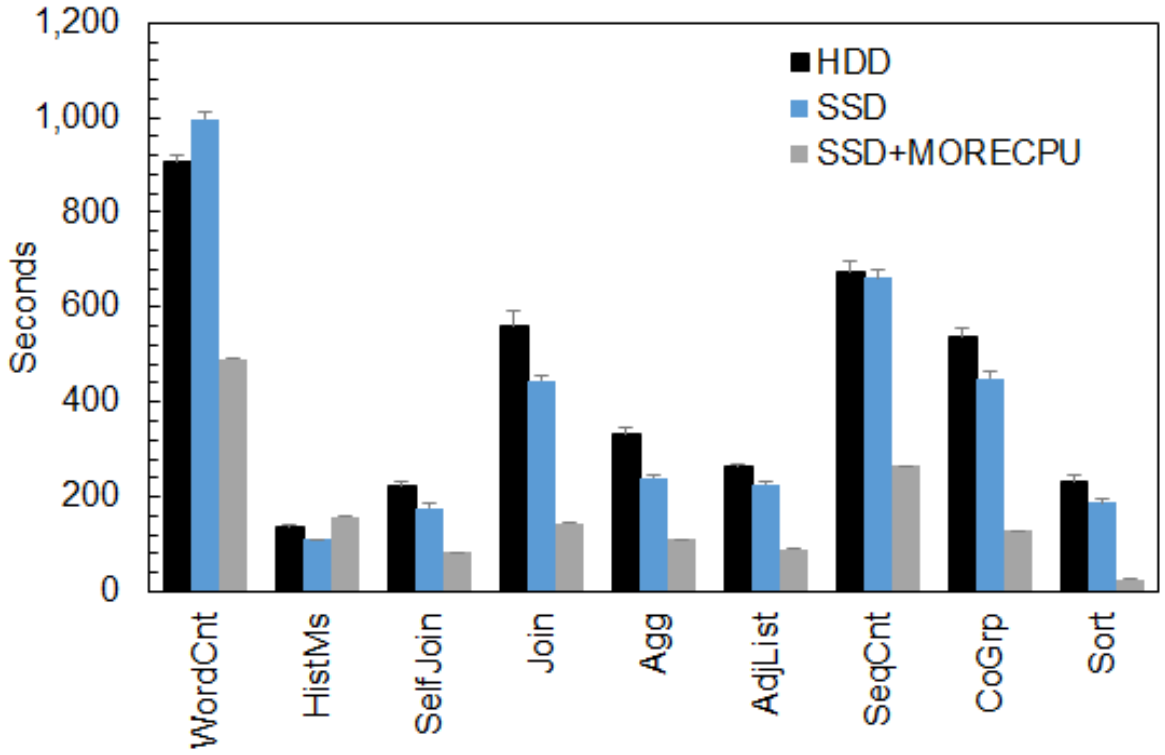


Figure 3: Time to Recover a Data Block (64 MB) under Different Cluster Configurations.

within 600 seconds with the exception of WordCnt, which requires approximately 850 seconds. Due to the fact that WordCnt has a very high connectivity between input and output records, reproducing a missing block requires reading in more than 70% of the initial input dataset. Results also show that an SSD-based configuration does not significantly speed up re-execution thereby providing evidence that reproduction is compute bound.

We quantify the benefit of increased compute capacity for block recovery by rerunning the evaluation on SSD-based `n1-highcpu-16` instances (16 cores and 14.4 GB RAM). Results show that providing more compute capacity significantly accelerates recovery, reducing the time required to reproduce the block by 70% regardless of workload. With this setup 3, workloads can recover a block in less than 90 seconds; the fastest is Sort taking 26 seconds to complete. All workloads can recover a block in less than 300 seconds with the exception of WordCnt. By proving that reproduction is compute bound we identify that block recovery time can be minimized simply by allocating the recovery job more compute capacity.

6.4.2 Disk and Network Overheads

Given that compacted lineage is the fundamental conceptual primitive that provides support for block-level reproduction, the granularity of this lineage directly affects the network and disk I/O resource overheads during recovery. As quantified in Section 6.3.1, lineage stored at record-level has the highest spatial overhead. This overhead decreases as sector-size is increased. Conversely, the network and disk I/O resource requirements for recovering a block are the smallest when lineage is at record-level, and increase as sector size increases.

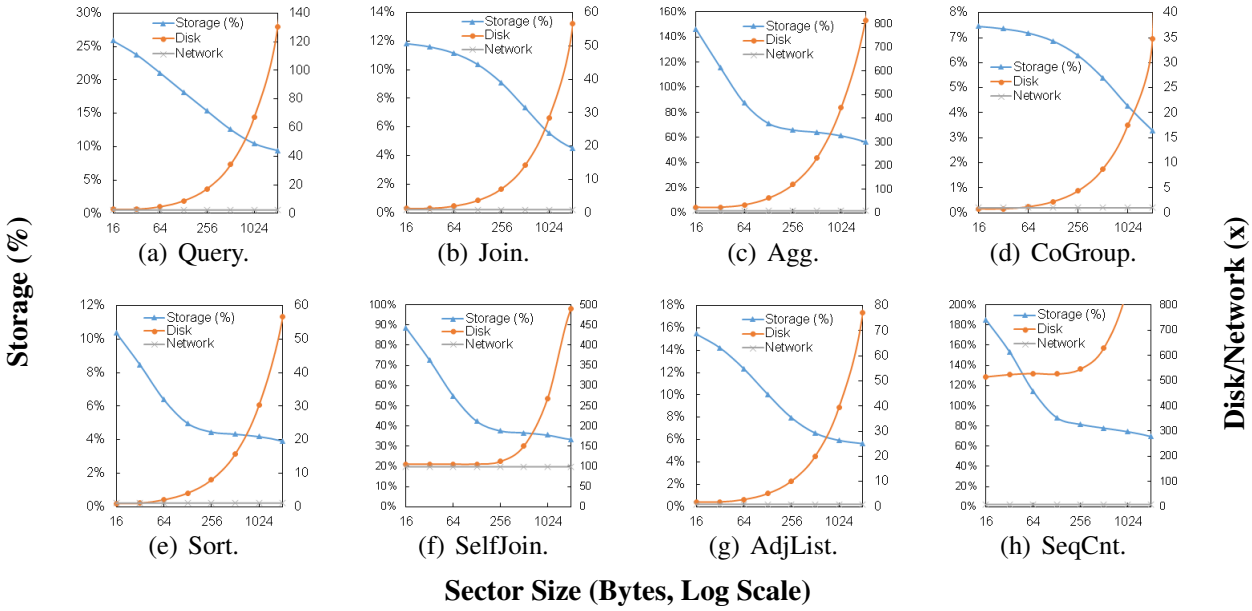


Figure 4: The Effect of Sector Size on Lineage Storage (Left Vertical Axis), Disk Read and Network Transfer (Right Vertical Axis) Overheads.

Figure 4 presents the quantitative relationship between compacted lineage storage overhead, and the network and disk I/O resources required to carry out a block recovery for eight out of the ten workloads in the evaluation set. WordCnt and HistMs are omitted as lineage collection is infeasible as explained in Section 6.2.

Each graph represents a unique workload. On every graph is illustrated the compacted lineage (left vertical axis), relative to the output dataset size for lineage collection sector sizes of 16 to 2048 bytes. Every graph also illustrates, normalized to the output size, the average amount of data that needs to be read from the input (right vertical axis) to successfully recover the failed block. Finally, every graph illustrates, normalized to the output size, the amount of data that is sent over the network (right vertical axis) for the purposes of supporting recovery of the failed block.

We highlight the following notable results:

(*i*) For workloads that have low data input-to-output ratio, the amount of disk read required is proportional to the size of the data recovered. However data summarization jobs (e.g., Agg) require an infeasible amount of input data to be read on recovery.

(*ii*) As expected, as sector size increases lineage size decreases but the average amount of data read required to service a block during recovery increases.

(*iii*) The SelfJoin workload (Figure 4(f)) can achieve reductions in lineage storage by increasing lineage sector size while maintaining constant data read overhead. This is due to the fact that input records in SelfJoin are relatively large; by extension, increasing lineage sector size does not affect lineage preciseness. For sector sizes of 512 bytes or larger, however, disk read overhead begins to increase but at this point lineage storage overhead is constant.

(*iv*) The network overheads during recovery are negligible for all but one workload. They achieve an overhead $\leq$ 10x of the output data recovered. In fact workloads that have low data input-to-output ratio incur just 1x as in the case of Query, Join, CoGroup, Sort and Adjlist. In RMR, we are able to shuffle only the relevant intermediate records that are need to reproduce

	Disk I/O	Network I/O
WordCnt	↑	↑
HistMs	↑	↑
SelfJoin	↑	↑
Query	2.8x	2.4x
Join	1.3x	0.9x
Agg	↑	↑
AdjList	1.9x	1x
SeqCnt	↑	8.5x
CoGroup	0.8x	1x
Sort	1x	1x

Table 6: RMR I/O Overheads During Recovery for $\leq$40% Storage Overhead.

	Storage Overhead
WordCnt	↑
HistMs	↑
SelfJoin	↑
Query	21%
Join	10%
Agg	↑
AdjList	10%
SeqCnt	↑
CoGroup	5%
Sort	5%

Table 7: RMR Storage Overhead for $\leq$5x Data Read During Recovery.

a given output block as discussed in Section 3.3.1.

The results in this section highlight the importance of tuning lineage sector size on a per-workload basis. They also show that RMR incurs a low network overhead during recovery comparable to output data.

6.5 Comparison with Erasure Codes

In this section we provide a first-order comparison between RMR and two state-of-the-art erasure codes in production use. We compare the storage and block recovery overheads of RMR in relation to Local Parity [3, 4] and Piggyback Codes [5].

For example, a (10,6,5) local parity code [3] configuration has 5x repair traffic (disk read and network transfer) overhead at the expense of 60% storage overhead. We test how much storage is required in RMR to achieve similar repair traffic.

Table 7 shows that 5 of the 10 workloads in the evaluation set we are able to meet this 5x repair traffic with significantly lower storage footprint. Of particular interest, note that the maximum compacted lineage overhead is 21% of the output size, compared to 60% in the(10,6,5) Local Parity Code case. This result shows that for some workloads RMR can result in significantly lower additional storage overhead than Local Parity Codes while still achieving the low (recovery) resource overheads exhibited by the codes. We omit a discussion of network costs as the results in Section 6.4.2 have already shown that network transfer overhead is 1x in these workloads for which RMR shows an advantage.

Piggyback codes are an optimization to RS codes designed to reduce the amount of data required to recover a block. For example a (10,4) Hitchhiker-XOR code has 40% storage overhead while blocks can be recovered by reading around 6.5 blocks on average from the stripe. We compare RMR against piggyback codes by evaluating the disk and network resources required to recover a block in the output dataset if the compacted lineage overhead is capped to a maximum of 40% of the output dataset size.

Table 6 shows that RMR is able to achieve recovery with significantly fewer disk and network I/O resources than a (10,4) Hitchhiker-XOR code in 5 of the 10 workloads in the evaluation set. In particular, it requires 2–8 times less disk read and transfers 1–7.2 times less network traffic during block reproduction.

This result shows that in specific cases RMR is significantly more resource efficient than erasure codes especially on storage requirement. This first-order comparison illustrates the potential benefits of employing RMR in practice.

7 Discussion

Although RMR is limited to the subset of MapReduce jobs that exhibit low input-to-output record connectivity, the potential storage and data recovery savings it offers makes it attractive for some environments. Storage capacity has already been identified as a bottleneck in big data environments [5, 15, 8]. We provide a tool that is effective in aggressively reducing the footprint of output datasets without compromising reliability.

As the effectiveness of RMR is workflow dependent, we argue that it is complementary to and can be used in conjunction with general purpose data redundancy mechanisms (data replication and erasure codes). It is simple and cheap to identify jobs that may potentially benefit from RMR and the lineage capture overheads are low.

While the recovery time in RMR is longer than traditional replication techniques we believe it can be lowered. In particular, we are exploring optimizations to HDFS for accelerating segmented reads and for increasing the degree of parallelism when carrying out recomputation based block recovery.

Jobs that exhibit high input-to-output connectivity do not fare well in RMR. We argue, however, that in situations where the data is cold it may be simpler to just delete intermediate outputs and recompute the job from scratch when outputs are required.

The fact that RMR's post-processing compaction and block recovery functionality can be scheduled arbitrarily greatly increases overall system flexibility. We are further exploring the advantages of aggregating post-processing for multi-stage jobs and working on techniques to identify and leverage data input and output commonalities between distinct jobs.

8 Related Work

Using lineage to recompute missing data has previously been explored by a few systems. For example Tachyon [7], a caching layer on top of HDFS, incorporates file-level lineage to rebuild in-memory data upon failure. Similarly, workloads in this architecture are not required to synchronously write out outputs. This flexibility significantly speeds up computation as I/O wait times are minimized.

Spark [26], an in-memory data processing framework, captures lineage for its in-memory "RDD" data abstraction. It is able to recompute data when a node fails. Dependencies between RDDs are specified using parallel operators in the API.

Nectar [8], an on-disk caching system, applies lineage-based recomputation of old datasets on-demand. In this case cold data can be effectively deleted to reclaim storage space in data warehouse systems. Nectar, however, relies on data replication to ensure data reliability.

We note that all these systems use coarse-grained lineage for recomputation. While this is sufficient for their respective use-cases, coarse-grained lineage leads to imprecision (e.g., wide dependencies in join, sort and multi-stage workloads [26, 7]). Consequently, MapReduce jobs incur high recomputation costs even if a small subset of the output is required. We argue that support for more targeted recomputation is necessary in scaling up data reuse and reliability.

The concept of lazy recovery of cold data has been proposed by a few studies [16, 8]. As cold data is accessed infrequently, these systems can effectively optimize storage and rebuild overheads significantly. We employ similar strategies in RMR; however we recover lineage files eagerly towards the goal of being able to reproduce blocks more efficiently.

9 Conclusion and Future Work

This paper has introduced RMR, a system that supports recomputation-based block-level data recovery for MapReduce. RMR works by recording fine-grained record-level lineage during the job execution, leveraging this lineage to recompute MapReduce jobs on only the subset of inputs required to reproduce a failed output block.

We have evaluated RMR on 10 common types of stateless MapReduce jobs. We show that, for many jobs, RMR imposes a temporal overhead of $\leq 10\%$ of the job runtime while providing data reliability guarantees with a smaller storage footprint. We further show that RMR can recover data blocks with manageable disk and network I/O requirements.

In future work we plan to extend the system to support stateful and multi-stage jobs. We are also extending the codebase and plan to release it as open-source.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST'10*.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP'03*.

[3] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: novel erasure codes for big data," in *VLDB'13*.

[4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *USENIX ATC'12*.

[5] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers," in *SIGCOMM '14*.

[6] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of SIAM*, 1960.

[7] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica, "Tachyon: Memory Throughput I/O for Cluster Computing Frameworks," in *LADIS'13*.

[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in *OSDI'10*.

[9] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy Efficiency for Large-scale MapReduce Workloads with Significant Interactive Analysis," in *EuroSys '12*.

[10] F. Dinu and T. S. E. Ng, "RCMP: Enabling Efficient Recomputation Based Failure Resilience for Big Data Analytics," in *IPDPS '14*.

[11] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *CCGrid'10*.

[12] Z. Xu, M. Hirzel, and G. Rothermel, "Semantic characterization of MapReduce workloads," in *IISWC'13*.

[13] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *MASCOTS '11*.

[14] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littleeld, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A Distributed SQL Database That Scales," in *VLDB'13*.

[15] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *HotStorage'13*.

[16] M. Silberstein, L. Ganesh, Y. Wang, L. Alvizi, and M. Dahlin, "Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage," in *SYSTOR'14*.

[17] D. Logothetis, S. De, and K. Yocum, "Scalable Lineage Capture for Debugging DISC Analytics," in *SOCC '13*.

[18] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom, "Provenance-Based Debugging and Drill-Down in Data-Oriented Workflows," in *ICDE'12*.

[19] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04*.

[20] "Apache Pig!." http://pig.apache.org/.

[21] "Apache HIVE." http://hive.apache.org/.

[22] H. Park, R. Ikeda, and J. Widom, "RAMP: a system for capturing and tracing provenance in MapReduce workflows," *Proc. VLDB Endow.*, 2011.

[23] S. Akoush, R. Sohan, and A. Hopper, "HadoopProv: Towards Provenance As a First Class Citizen in MapReduce," in *TaPP'13*.

[24] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "PUMA: Purdue MapReduce Benchmarks Suite," Tech. Rep. TR-ECE-12-11, Purdue University, 2012.

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

[26] 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'12*.