MR+: Knocking Down the MapReduce Brick-wall

Ahmad Humayun
University College London

* * *
*

Abstract

This paper presents a modified architecture of MapReduce, dubbed MR+, which advocates a radical departure from the fixed two-staged architecture of MapReduce to a flexible, multi-staged implementation. MR+ has several inherent advantages over traditional MapReduce: (1) it is resilient to skew in intermediate results; (2) it avoids the wholesale copying of intermediate data at the end of the map phase which may otherwise paralyze the entire cluster while reduce workers are being loaded with data en masse, (3) it enables early estimation of results for very large datasets, (4) it naturally avoids the reduce straggler problem due to a heterogeneous cluster, (5) it may be used to prioritize the processing of large datasets by detecting clusters of useful information in the input data. Our improvements over the original architecture still maintain the clean, convenient programming model of MapReduce. Our initial experiments show that MR+ outperforms Hadoop MapReduce, Hadoop Online Prototype and LATE by up to a factor of 7 for our target applications and datasets.

1. Introduction

In the last few years, MapReduce [Dean 2004] has emerged as a popular platform for data intensive computation on commodity clusters. Developed at Google, MapReduce is now also used by other search engines like Yahoo! Search, Amazon's A9, and Microsoft Bing [Had, Ananthanarayanan 2010]. Apart from building web-indexes, MapReduce has found use in analyzing images, clustering news articles, computing various rankings, analyzing seismic data, parallelizing large-scale graph computations and machine learning problems [Dean 2004]. Social networks such as Facebook [Thusoo 2010], Twitter and LinkedIn use it to store and process log files and for reporting analytics [Had]. As we move towards petascale computations, the popularity of MapReduce is expected to increase [Dikaiakos 2009].

The popularity of MapReduce is largely attributed to two key factors: (1) MapReduce affords a great deal of flexibility for processing unstructured or "arbitrary" data that does not follow a strict SQL-like schema, such as log-files, collection of news articles, satellite images, product descriptions on the web, links from webpages etc. Since most data found in, say, user logs of web applications may not follow a strict schema, MapReduce has found growing use for mining, analyzing and processing of large unstructured data sets. (2) MapReduce affords a neat, conceptually simple model – derived from functional languages like Lisp – in which the complexity of parallel processing is hidden underneath the abstraction of a two-staged map-reduce computational model.

MapReduce, and its open-source implementation by Apache called Hadoop [Had], is popularly seen as a clean, convenient framework to parallelize the processing of large amounts of arbitrary data.

However, in this paper we argue that while MapReduce affords flexibility for processing unstructured data, it implicitly assumes certain properties of input data, intermediate results and application semantics which limit its performance and utility.

Above all, the performance of MapReduce goes down sharply when there is skew in the intermediate results from the map phase [Kwon 2010, Lin 2009]. Unfortunately, a large number of real-world applications are prone to producing intermediate results that follow a highly skewed Zipf-like distribution [Adamic 2002]. To understand skew, consider the vanilla *word count* example from the original MapReduce paper [Dean 2004] which could produce highly skewed intermediate results due to word frequencies. In this example, the map function emits the count of occurrences of each word and the reduce function sums up all the counts for each word. As shown in Figure 1, text corpora have a Zipfian skew, i.e. a very small number of words account for most occurrences, leading to a long tail distribution of intermediate data; out of 242,758 words, the 10, 100 and 1000 most frequent words account for 22%, 43% and 64% of the entire set. Such *skewed* intermediate results lead to uneven distribution of workload across reduce workers. Therefore, reduce tasks that process high frequency words automatically become "stragglers" and dictate the overall completion time.

Surprisingly even Google's own (iterative) MapReduce implementation of its core PageRank algorithm is plagued by skewed intermediate data that follows a Zipfian distribution [Lin 2009]. Google uses PageRank to calculate a webpage's relevance for a given search query[1]. In the MapReduce implementation, the map function emits the outlinks for pages and the reduce function calculates the rank per

[1] Google has developed another system called *Pregel* [Malewicz 2010] for inhouse large-scale graph computing. Pregel may have replaced MapReduce for PageRank calculation at Google.

page. The skew in the intermediate data – due to a huge disparity in the incoming links across pages on WWW – follows a power law [Fortunato 2008, Volkovich 2007]. The scale of the problem is evident when we consider the fact that Google currently indexes more than 25 billion webpages [Web] – with skewed incoming links. For example Facebook has 49,376,609 incoming links while the personal webpage of one of the authors of this paper has only 4[2]. The websites on the higher end of ingress links dominate the running time of PageRank MapReduce jobs.

Indeed, data skew is a widely recognized problem in the parallel databases community [DeWitt 1992, Xu 2008]. Data skew affects the performance of Inverted Indexing [Lin 2009], FP growth [Chuang 2006], Publish/Subscribe systems [Demers 2006], fraud detection systems [Huang 2008] and various clustering algorithms [Kerdprasop 2005]. We cite more instances of the skew problem in MapReduce in Section 1.4.

Reduce-heavy Applications: The original MapReduce paper, and most subsequent work, assumes the reduce phase to be a simple aggregation function. In contrast, in recent years, MapReduce has been used in a broad range of applications that are reduce intensive i.e. a good portion of the computation is carried out during the reduce phase. In MapReduce, the performance degradation due to intermediate data skew is exacerbated if the reduce function has an equal or higher complexity compared to the map function. These applications include image and speech correlation, backpropagation in neural networks [Chu 2006], linear regression [Ranger 2007], co-clustering [Papadimitriou 2008], tree learning [Panda 2009] and computation of node diameter and radii in Terabyte-scale graphs [Tsourakakis 2010].

An important contribution of our work is to highlight and evaluate the performance of MapReduce for reduce-heavy tasks. Our implementation is specifically designed for this class of applications.

1.1 Contributions

In this paper, we present the design, implementation and evaluation of a modified MapReduce system that performs well even when intermediate results (from the map stage) have skew. To achieve this, we make an important distinction between the programming model of MapReduce and the underlying architecture used to implement this programming model. The traditional implementation of MapReduce by Google [Dean 2004], and the open source project Hadoop [Had], implement the two-staged MapReduce programming model using a fixed two-staged architecture, in which reduce tasks run after all the map tasks have finished execution. We argue that this sequential two-staged implementation of the MapReduce programming model, with a "brickwall" between the Map and Reduce stages, is overly inflexible and suffers from several shortcomings, including skew.

[2] Found using Yahoo! Site Explorer.

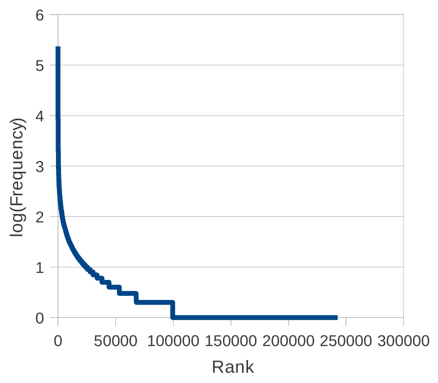


Figure 1. Zipfian distribution of frequency of words from the 50 most downloaded books from the Gutenberg library. A small number of words account for most occurences.

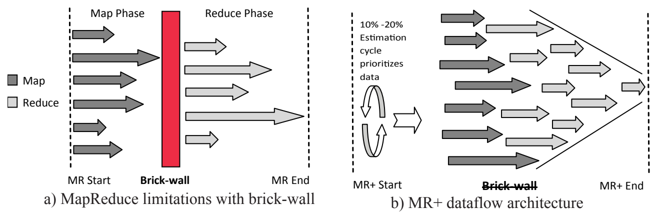
Therefore, the basic concept of our modified architecture of MapReduce is simple: we maintain the simple MapReduce programming model, but instead of implementing MapReduce as a sequential two-staged architecture, our system, dubbed MR+ allows map and reduce stages to interleave and iterate over intermediate results (shown in Figure 2). That is, MR+ changes the underlying implementation of MapReduce, but retains its clean functional API (in contrast to systems such as Dryad [Isard 2007]).

The multi-staged architecture of MR+ enables flexibility in two dimensions: a) Instead of waiting for all maps to finish before scheduling a reduce task, MR+ permits a model where a reduce task can be scheduled for every *n* invocations of the map function. b) MapReduce mandates that a specific key must be reduced completely by a single reduce task. MR+'s flexible scheduling enables a model where a densely populated key can be recursively reduced by repeated invocation of the reduce function at multiple reduce workers.

Therefore, the multi-staged architecture of MR+ ensures that no single node is overwhelmed with an unfair workload at one instance. Instead, map and reduce workers recursively process small shards of intermediate data, materializing and refining results as map and reduce stages are interleaved over time.

1.1.1 Multi-staged MapReduce

The primary goal of our work is to make MapReduce resistant to skew. However, our multi-staged architecture makes several tertiary contributions. MR+'s flexible MapReduce implementation has several natural advantages over the traditional two-staged implementation: (1) The interleaving model of MR+ leads to early materialization of partial results. Early estimation of results is very useful when processing petascale datasets since common queries with thresholds or confidence intervals may be answered efficiently without



a) MapReduce limitations with brick-wall b) MR+ dataflow architecture

Figure 2. Architectural comparison of MapReduce and MR+. Intermediate results in MR+ flow through an inverted tree-like structure of reduce workers, where each level reduces a fraction of the keys and emits the results for the next level.

processing the entire dataset. (2) The concurrent scheduling and interleaving of maps and reduces automatically removes the overwhelming network spike, network congestion [Chen 2009] and possible livelocking due to the wholesale copying of petabytes of intermediate data at the end of the map stage. (3) Assigning equi-sized shards of intermediate data to iterative reduce workers naturally avoids the reduce task straggler problem that degrades the performance of MapReduce in heterogeneous clusters [Zaharia 2008].

Our multi-staged MapReduce architecture can be extended to prioritize the processing of large datasets by detecting structure in input data. To accomplish this, we have built an extension of MR+ that runs an interleaved map-reduce sampling cycling on the input data to learn the distribution of information in the input dataset. This enables the MR+ runtime to prioritize those pieces of input data that may be clusters of useful information, such as logs of an intrusion detection system.

To summarize, MR+ architecture has the following inherent properties: (1) it is resilient to skew in intermediate results; (2) it avoids the wholesale copying of intermediate data at the end of the map phase which may otherwise paralyze the entire cluster while reduce workers are being loaded with data en masse, (3) it enables early estimation of results for very large datasets, (4) it naturally avoids the reduce straggler problem due to a heterogenous cluster, (5) it may be used to prioritize the processing of large datasets by detecting clusters of useful information in the input data.

It is important to highlight that the multi-staged architecture of MapReduce naturally leads to repeated invocations of the reduce function. This iterative implementation of reduces in MR+ can be viewed as a distributed *combiner*. Unlike the localized combiner in vanilla MapReduce, our implementation causes each reduce to combine and pass results on to the next level, until all the results are reduced. However, this flexibility comes at a cost; the iterative refinement of results characteristically restricts the reduce function from performing any non-associative operation such as a vector cross product. At the same time it is noteworthy that our architecture does not impose any additional constraints on the MapReduce computation model since MapReduce already mandates associativity for the combiner function.

1.2 Prior Work in Skew in MapReduce

Before we explain the architecture of MR+, it is useful to compare our approach to other recent systems that recognize the problem of skew in MapReduce.

SkewReduce [Kwon 2010] is a parallel data processing system for extraction of features from scientific datasets which exhibit a high degree of computational skew. It provides a three-function API to express feature-extraction algorithms. An inbuilt static optimizer partitions the data into a *partition plan* and an *execution schedule* to minimize computational skew and query execution time. The optimizer makes use of user defined cost functions that estimate processing times by relying on prior runs on samples of the original dataset. The execution engine converts the partition plan and execution schedule into a graph of Hadoop jobs which is submitted to a Hadoop cluster for execution. Our approach is fundamentally different from SkewReduce. SkewReduce is designed as a wrapper around MapReduce for addressing the problem of skew for a specific class of applications and exposes a specialized programming API targeted at feature extraction applications. On the other hand, our system, is designed as a generic solution to the skew problem and retains the generic MapReduce programming model. MR+ addresses the skew problem by moving away from the fixed two-staged implementation of MapReduce to a flexible implementation, where map and reduce may interleave and iterate.

Another recent system, Mantri [Ananthanarayanan 2010] is designed to address the problem of skew due to outliers in the intermediate results of MapReduce. Mantri monitors input data to a reduce worker at runtime, and if it detects a straggler, it restarts or replicates a task at another node in an attempt to mitigate both failed nodes and slow stragglers. In contrast, MR+ does not introduce the overhead of runtime monitoring, speculative scheduling or restarting of tasks. Instead, the multi-staged architecture of MR+ naturally leads to workload balancing at runtime to mitigate the effect of skew.

Likewise, MR+ is fundamentally different from PIG [Olston 2008], Sawzall [Pike 2005], Dryad [Isard 2007] and DryadLINQ [Yu 2008] that introduce a completely new programming model to implement a dataflow computation model. MR+ retains the simple MapReduce programming model but uses a more flexible architecture to implement MapReduce. Importantly, PIG, Sawzall, Dryad and DryadLINQ do not explicitly address the problem of data skew. In fact, PIG, which is designed as a layer on top of Hadoop MapReduce, also suffers from skew during Join operations [Gates 2009]. Similarly, Dryad also faces significant performance degradation when the input dataset is skewed in nature, due to its static non-global scheduler [Qiu 2009]. We summarize other related work in Section 6.

1.3 Defining Skew

The *skewness* metric of any distribution can be defined in terms of entropy [Cheung 2002]. The entropy $H(X)$ for any random variable X is a measurement of the uncertainty in its value. Put differently, entropy is a measurement of the eveness (or uneveness) of the probability distribution over the values of the random variable X. For a dataset with k keys, the skewness $S(k)$ is given by:

$$S(k) = \frac{H_{max} - H(k)}{H_{max}} \quad (1)$$

where $H(k) = -\sum_{i=1}^{n}(p(k_i)log(p(k_i)))$, $H_{max} = log(n)$ and $p(k_i)$ is the probability of key i.

$S(k)$ has the following properties:

- $S(k) = 0$, when all $p(k_i)(1 \leq i \leq n)$ are equal. Skewness of an input dataset or intermediate key-value pairs is zero when the keys are evenly distributed.
- $S(k) = 1$, when a single $p(k_i) = 1$ and all others are zero. Skewness of an input dataset or intermediate key-value pairs is one (maximum) when all keys are present in a single partition.
- $0 < S(k) < 1$, otherwise.

The above properties also show that high frequency terms in case of applications such as, say word count lead to low entropy [Ananthanarayanan 2010, Elsayed 2008] and hence more skewness. For instance, the distribution in Figure 1 has a skewness of 0.622.

1.4 Prevalence of Data Skew

Data skew is a fundamental problem in parallel computing. This section highlights examples of data skew identified by previous work on MapReduce.

Kavulya et al. [Kavulya 2010] have presented a detailed study of the MapReduce logs from Yahoo!'s M45 Supercomputing cluster to understand the characteristics of workloads. The study has shown that there are some jobs which have a very high Gini-coefficient [Gini 1971] (a ratio of actual cumulative job durations to the ideal equal cumulative job durations). This is attributed to either performance problems or data skews. Xu et al. [Xu 2008] make a very compelling argument for handling data skew in parallel systems:

> "Data skew occurs naturally in many applications. A query processing skewed data not only slows down its response time, but generates hot nodes, which become a bottleneck throttling the overall system performance."

The architects of PIG [Olston 2008], in a subsequent paper [Gates 2009] have highlighted the performance problems faced due to data skew while using PIG at Yahoo!. According to the paper, when a few of the join keys have a large number of matching tuples as compared to the rest of the keys, performing a join at a single node sometimes causes PIG to spill to disk. The designers of Hive, the open-source data warehousing solution built on top of Hadoop by Facebook, have also enumerated design of better techniques for handling skews for a given key as a future optimization thread [Thusoo 2010]. Performance issues due to stragglers in MapReduce have been explored by Lin [Lin 2009] in detail. Lin makes the following observations: 1) The large amount of skew in the running times for map and reduce tasks is a result of Zipfian distribution [Adamic 2002] of input or intermediate data. In fact, Lin argues that speculatively executing tasks in such a situation does little to address skewed running times. 2) In most cases the effect of Zipfian distributions can be minimized by breaking long input data lists into same-sized smaller shards of work while skew in intermediate data can be tackled by sampling strategies. 3) Better load distribution across reducers can be achieved once the distribution of the intermediate data has been generated.

The architecture of MR+ is inspired by experimental analysis and characterization of the skew problem in MapReduce by Lin [Lin 2009], Kavulya [Kavulya 2010] and designers of Hive [Thusoo 2010] and PIG [Gates 2009].

1.5 Structure of the Paper

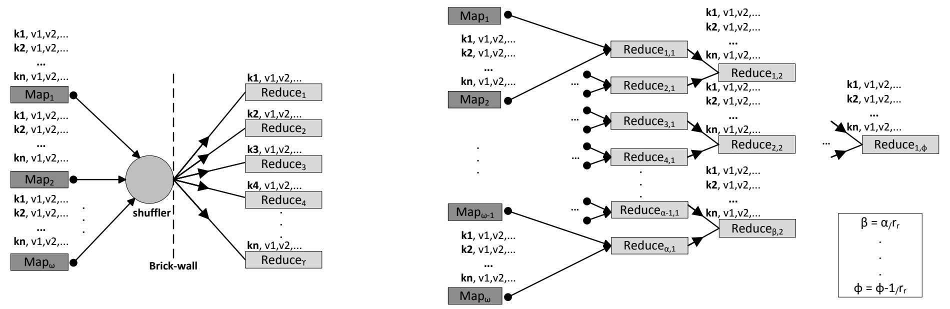
The rest of the paper is structured as follows. We describe the MapReduce computational model in Section 2. In Section 3 and 4 we present the architecture and implementation of our system. In Section 5 we present an evaluation of MR+ and compare it with Hadoop [Had], Hadoop Online Prototype [Condie 2010] and LATE [Zaharia 2008]. Related work is summarized in Section 6. We conclude the paper in Section 7.

2. Overview of MapReduce

In this section, we present the MapReduce computational model and discuss its fault-tolerance mechanism.

2.1 MapReduce Computational Model

In MapReduce, a programmer divides the computation into two stages, such that the first stage *maps* each logical record in the input data – stored as files on the Hadoop Distributed File System (HDFS) – and computes a set of intermediate key/value pairs. The second stage applies a *reduce* operation to all the values that share the same key that combines, merges and aggregates the intermediate key/value pairs. The use of a functional model with programmer-specified map and reduce operations naturally supports a framework in which both processing (map) and aggregation (reduce) of data could be parallelized. Likewise, MapReduce's functional model conveniently lends to a simple model of fault-tolerance in which failed or slow "straggler" [Dean 2004] tasks may be re-executed with a different set of workers to mask failed or slow machines.



a) MapReduce b) MR+

Figure 3. Distribution of keys to reduce tasks in MapReduce and MR+. In MapReduce all values for one key are crunched by the same reduce task after being passed through the shuffler. The shuffler groups key-value pairs based on each key. MR+ in comparison, does not require a shuffler. It permits one key to be reduced by different reduce tasks with further consolidation performed by tasks in subsequent levels.

3. MR+

In this section, we describe the architecture of MR+ in detail (Section 3.1) followed by a discussion on implementation challenges (Section 3.2).

3.1 Architecture

MR+ transforms MapReduce from a fixed two-staged process to a flexible multi-staged architecture, in which input data passes through several levels of interleaved map and reduce stages (shown in Figure 2).

To achieve this, MR+ relaxes two constraints in the vanilla MapReduce implementation:

- Map and Reduce stages run once sequentially on the entire data.
- All the values for a particular key are reduced at a single reduce worker.

By relaxing these two constraints, MR+ enables flexibility in processing of parallel tasks in two different, but related, ways: (1) Since map and reduce functions can be scheduled repeatedly, they can be interleaved to process input data in multiple stages, and (2) a key can be reduced iteratively by scheduling the reduce function repeatedly, possibly on different nodes.

In MR+, intermediate results, in effect, flow through an inverted tree-like organization of map and reduce workers, where at each level, a fraction of the intermediate keys are processed and results are emitted for the next level of workers. This naturally permits both early estimation of results and the reduction of a single key in parallel on different nodes.

It is important to highlight that given the applicative programming model of MapReduce, both Map and Reduce tasks are free of side-effects, and hence naturally permit repeated invocation of map and reduce functions on smaller chunks of data. The only additional requirement introduced by iterative scheduling of the reduce task in MR+ is that the input and output data for the reduce task must be in the same domain. This in effect makes it possible for a reduce to get input from both maps and reduces in parallel.

3.2 Implementation Challenges

Transforming MapReduce from a two-staged architecture to a multi-staged architecture presents a number of major challenges.

1. **Scheduling Complexity:** The multi-staged architecture of MR+ could result in increased complexity for the scheduler. In MR+, the scheduler has to both make more scheduling decisions and maintain additional state when interleaving map and reduce workers.

   We address this by limiting the "fan-in" of maps to reduce and reduces to reduce. This puts an upper bound on the additional state that must be kept at the scheduler and spreads the work load of the scheduler to the entire timeline of a MapReduce application. Therefore, the scheduler does not become a performance bottleneck in our architecture. We explain this further in Section 4.

   As a side, to circumvent the continuous involvement of the scheduler in our architecture, we are exploring a masterless implementation of MapReduce. However, such an implementation necessarily involves use of advisory locks, such as Chubby [Burrows 2006] or lockfree data structures [Herlihy 1993], both of which could impose a substantial overhead for massively parallel tasks.

2. **I/O Overhead:** With its iterative architecture, MR+ runs many more (smaller) reduce workers compared to vanilla MapReduce. Since intermediate results must be transferred between reduce workers, this could lead to a substantial I/O overhead in a naive implementation.

   MR+ addresses this challenge by paying special attention to keep intermediate results localized. We explain this further in Section 4.4.1 and evaluate the impact of reduce-locality.

3. **Fault Tolerance:** In MR+, data flows through several levels of reduce workers. This has a downside that if one

of the non-leaf reduce worker fails, the intermediate state could be lost, precluding any subsequent processing.

To mitigate the challenge of failed intermediary reduce workers, an MR+ reduce worker always maintains a backup node while it is processing intermediate results. This node serves as a cold-backup and may be scheduled by the scheduler to restart the task if it detects a failed node. In order to control the I/O overhead, a cold-backup is always maintained on a node which shares the same subnet mask (typically on the same rack). Indeed, the number of replicas keep shrinking as we can go down the reduce tree, as number of reduce tasks decrease. Replicas can be deleted as soon as a dependent reduce in a higher reduce level replicates its input

4. **Reduce Key Implosion:** As mentioned earlier, the iterative nature of the MR+ architecture results in an inverted tree-like organization of reduce workers. In this model, all intermediate results "converge" on the final reduce worker. However, this could potentially lead to a scenario where intermediate keys, which have already been completely reduced, still *ride* reduce levels; a key that has been fully reduced is still copied between levels all the way to the last level of reduce. This could be problematic in case of peta or terascale computations in which the last few reduce levels may be overwhelmed by the influx of a huge number of keys.

   This problem requires mechanism to identify keys that have been fully reduced early and taking them out from the set of intermediate reduce keys.

   Section 4.4.4 explains the mechanisms in MR+ to avoid the key implosion problem.

5. **Resource Aware Scheduling:** MR+ assigns a small number of intermediate results to each node to avoid burdening one node with an unfair workload due to skewed intermediate results. However, often the mere count of the intermediate keys is not a good measure of the ensuing processing load on the node.

   Therefore, MR+ implements a scheduling scheme where nodes constantly monitor their resource utilization state and report it back to the master. The master takes this state into account during task allocation. This state only adds a moderate overhead since these resource usage reports are piggybacked on heartbeat signals. This is explained further in Section 4.4.3.

4. Implementation

In this section, we describe the implementation details of MR+ and compare it with the key implementation details of Hadoop.

No. of Input Files :	25	50	100	250	500	800	1200
% Node Local	25	50	25	25	26.52	28.30	30.06
% Switch Local	50	35.71	53.57	61.76	67.42	66.98	62.34
% Non Local	25	14.29	21.43	13.24	6.06	4.72	7.59

Table 1. MR+ Reduce Locality

4.1 Hadoop MapReduce Implementation

Before describing the implementation details of MR+, we first outline the key implementation details of Hadoop in this section.

In Hadoop, a central *JobTracker* (running on the master) slices up the *job* into smaller *tasks*. This JobTracker orchestrates the assignment of tasks to *TaskTrackers* (running on each worker node) which in turn handle intra-node task execution. The maximum number (slots) of maps and reduces that are run by a TaskTracker at a time, can be configured at the time of job submission but remains fixed during job execution. The JobTracker first fills up all the slots on each TaskTracker with map tasks. During execution, every TaskTracker sends a periodic heartbeat signal to the JobTracker indicating that it is alive. Through this heartbeat signal the TaskTrackers also declare any free slots. If any TaskTracker declares a free slot, the JobTracker assigns it a new task through the heartbeat return signal. After all map tasks have been scheduled, reduce tasks are scheduled in a sequential order. Note that reduce tasks can be assigned before the brick-wall, but can only start processing once they have received input from all maps. As reduce tasks pull key-value pairs from across the cluster, there can be no locality considerations.

4.2 MR+ Implementation

The implementation of MR+ is derived from Hadoop MapReduce. However, unlike Hadoop, MR+ enables the same key to be reduced by different reduce workers in parallel, and in turn, permits multi-level reduces (Figure 3). To achieve this, MR+ changes the implementation of Hadoop such that the values for the same key in the intermediate results can be split between different reduce workers. In turn, reduce workers are scheduled repeatedly, each consolidating (reducing) some results from a previous stage, up until a stage when the set of values from each key is small enough to be reduced at one node.

In the MR+ implementation, although the role of the JobTracker and TaskTracker is identical to that in Hadoop, the details of task scheduling are completely different. To achieve interleaving of maps and reduces, MR+ tasks are allocated according to a configurable *map_to_reduce_schedule_ratio* parameter. The first reduce task is scheduled as soon as a certain number of map tasks complete. For example, if *map_to_reduce_schedule_ratio* is 4 then the first reduce task – corresponding to a reduce level of 1 – is scheduled as soon as the first 4 map tasks complete. Each reduce task is as-

No. of Reduces:	1	5	10	15	20
Hadoop	16.34	10.74	7.86	7.02	6.05
MR+	61.57	57.84	53.17	47.43	46.28

Table 2. Throughput in Mbps - Hadoop vs. MR+

signed[3] the output of *map_to_reduce_ratio* number of maps. To enable the TaskTracker of the newly scheduled reduce task to fetch *map_to_reduce_ratio* inputs, tuples containing TaskTracker IDs and Map IDs are communicated by the JobTracker. The reduce task pulls the output of the maps through a TCP socket and then sorts the values key-wise. Finally, it applies the user-provided reduce function to each key. After all keys have been processed, the output is written to the local filesystem – which is then pulled by a reduce task of a greater level. To ensure consistency, MR+ imposes object-type agreement on the input and output of reduces.

Level-two reduces are scheduled as soon as the first level-one reduce task writes its output to the local filesystem. Every level-two and greater reduce task is assigned the output of *reduce_input_ratio* number of reduce tasks. This pattern of reduce task assignment and writing output to the local filesystem continues until the final reduce level. Due to the decreasing number of reduce tasks per level, the final reduce level naturally consists of just a single reduce task. At this point all values are small enough to be reduced by a single worker. This final reduce can also be made to use non-associative functions such as exponentiation on the final result; which are then written to the HDFS. The minimum number of reduce levels is decided by the formula:

$$L = \log_{r_r} \frac{M \times r_r}{m_r} \tag{2}$$

where M are the number of map tasks, r_r is the reduce to reduce ratio and m_r is the map to reduce ratio. The ability to control the *map_to_reduce_ratio* and *reduce_input_ratio* is key to minimize the additional complexity required in the MR+ scheduler, as highlighted in Section 4.4.1.

4.3 Evaluation Framework

MR+ implementation builds upon Hadoop. The implementation of MR+ took less than ~10k lines of Python code[4]. Like Hadoop, MR+ uses the *Hadoop Distributed File System (HDFS)* to store the input datasets for map tasks and the job results at the completion of the final reduce level. We use the Hadoop Thrift Server [Had] as middleware between the HDFS (in Java) and MR+.

For evaluation we used a cluster of 11 machines with 1 exclusive master node. The machines in our cluster range between 1 GHz P4 with 1 GB RAM to 64-bit Linux machines with two 3.2 GHz Xeon processors with Hyper-Threading and 4 GB of RAM. This heterogeneity in our cluster is deliberate since we want to evaluate the performance of MR+ for a realistic, moderate-sized cluster. The heterogeneity also enables us to compare our multi-staged architecture to speculative scheduling techniques like LATE. The nodes in our cluster are connected via 1 Gb Ethernet. We compare the performance of MR+ to Hadoop 0.21.0 [Had], Hadoop Online Prototype (HOP) 0.2 [Condie 2010], and LATE [Zaharia 2008]. HOP is a modification of MapReduce that enables inter-task and inter-job pipelining of data to achieve better cluster utilization, increase parallelism and decrease response time. On the other hand, LATE is a speculative scheduler for straggler tasks. For a fair comparison between MR+ which in Python and Hadoop, HOP, and LATE which are in Java, we compiled the Python code to bytecode.

The rest of this section describes how MR+ addresses the implementation challenges highlighted in Section 3.2 and evaluates the performance of MR+ using a canonical wordcount example. We use different datasets from the online Gutenberg [Gut] library for our experiments.

4.4 Implementation Details

4.4.1 Reduce Locality

In MapReduce, since a single reduce worker is solely responsible for reducing all the intermediate values of a specific key, it must copy key-value pairs from all map tasks located across the cluster. Therefore, MapReduce cannot localize reduce input.

On the other hand, due to its interleaved architecture and a controlled *map_to_reduce_ratio* and *reduce_input_ratio*, MR+ attempts to minimize the data to be copied across nodes. To achieve this, the MR+ scheduler attempts to schedule subsequent invocations of a task on any one of the nodes that completed the previous level of processing, if possible. In a condition when the same node may not be available for a subsequent invocation (due to unavailability of slots), MR+ prioritizes the available nodes based on network co-location by comparing their subnet masks. As a result, each worker in MR+ typically accepts input from other nodes in its own rack i.e. I/O is typically localized within a rack (behind the same subnet domain). We call such traffic switch-local traffic.

Table 1 shows the percentage of node-local, switch-local and non-local traffic due to our locality-aware scheduling scheme in our canonical wordcount example. On average, more than 85% of reduce task allocation is localized within the same subnet for different number of input files.

For completeness, it is important to highlight that there is often a trade-off between fairness and data localization in a system like MapReduce [Isard 2009, Zaharia 2010a]. Fairness dictates that tasks be allocated resources as soon as they are requested but the locality principle delays this allocation till the computation can be moved closer to the data. Giving preference to fairness over locality can lead to an expensive transfer of large swathes of data across the

[3] All reduce task input assignments are made on the basis of locality which we discuss in the next section.

[4] As MR+ differs substantially from Hadoop in certain aspects, we felt the need to take a fresh start and code the entire architecture from scratch.

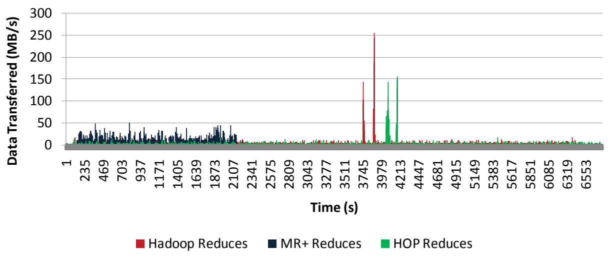


Figure 4. Network I/O comparison between Hadoop, HOP and MR+. Hadoop and HOP show a significant spike at the end of the Map phase. Whereas, the interleaving nature of MR+ allows it to transfer data from maps to reduces as soon as it becomes available.

cluster [Dean 2004]. The MR+ runtime tries to find a sweet spot between fairness and localization by incorporating both into its scheduling decision.

4.4.2 Avoiding Traffic Spike and TCP Incast

Unlike MapReduce where reduce tasks pull cross-cluster key-value pairs, MR+ implements a model in which each level 1 reduce task only pulls data (regardless of key constitution) from a fixed number of map tasks defined by a parameter *map_to_reduce_ratio*. Likewise, each level > 1 reduce task pulls data from a fixed number of reduce tasks defined by a parameter *reduce_input_ratio*. This enables MR+ to avoid the "TCP incast problem" [Chen 2009] in which datacenter applications experience sub-optimal utilization of link capacity. This performance bottleneck is attributed to the overfill of switch buffers by high amounts of incoming data. According to [Chen 2009] this flood of data leads to TCP timeouts which in some cases last hundreds of milliseconds resulting in a drop in throughput by up to 90%. Figure 4 plots the network traffic of MR+ during the multi-staged reduce phase and compares it with both Hadoop and Hadoop Online Prototype (HOP) for our word count example. The multi-staged architecture of MR+ avoids the network traffic spike in Hadoop since all reduce workers copy keys from map workers en masse at the end of Map cycle. Interestingly, HOP, which also implements pipelining in Hadoop, generates a traffic spike (though almost half in its magnitude) when the reduce tasks are unable to keep up with the map tasks and HOP defaults to the brick-wall architecture of Hadoop. MR+, on the other hand, is able to limit the network traffic over time since it can constrain the reduce input ratio and transfer intermediate data to the reduce tasks as soon as it becomes available. For comparison, for our word count example, Table 2 MR+ shows that MR+ achieves close to 5 times better throughput than Hadoop on average.

4.4.3 Resource Aware Scheduler

In order to avoid burdening a node with an unfair load, MR+ implements resource-aware scheduling. As opposed to the native Hadoop scheduler, the resource utilization aware

```
map(String key, String value):
// key: camera id
// value: frames for the camera
for each frame f in value:
    if brightness(f) > brightness_threshold:
        EmitIntermediate(key, f);

reduce(String key, Iterator values):
// key: camera id
// values: the frames output by the map function
distinct_frames = [];
for each frame in values:
    correlation_val = correlation_coeff(frame, frame+1);
    if correlation_val < correlation_thresh:
        distinct_frames.append(frame);
Emit(distinct_frames);
```

Figure 5. Pseudo code of map-reduce image correlation. The map function acts as a filter to emit images that cross a certain threshold of brightness. The reduce function performs image correlation. The same code is applicable to both Hadoop and MR+.

scheduler configures every TaskTracker to monitor its rate of page flushing, swap space usage and memory overcommit[5]. If any of these values exceeds configurable thresholds, the TaskTracker stops asking the JobTracker for any more tasks. To implement this, every TaskTracker in MR+ piggybacks a variable *give_task* when it sends its periodic heartbeat signal to the JobTracker for task allocation. The JobTracker assigns a task to a node only if *give_task* is true and the TaskTracker is not already running tasks to its full capacity (*task_capacity*). If *give_task* is false then the scheduling decision is deferred to the next heartbeat. A TaskTracker in MR+ can set the *give_task* false, if, for instance, its swap space fills up more than 90%.

4.4.4 Reduce Key Implosion

In order to avoid implosion of keys at the reduce workers in later iterations (as outlined in Section 3.2), MR+ implements two mechanisms:

1. In order to identify keys that may have been fully reduced, the TaskTracker in MR+ sifts out keys that only have a single value at any reduce worker.
2. MR+ system includes a "monitor" that periodically takes a snapshot of the different TaskTrackers to identify keys that have a single instance across all reduce workers. If the monitor detects a key with a single value, it concludes the key has been fully reduced, it removes it from the intermediate results and emits it to the output directory on the HDFS.

This way, keys with fewer values are both reduced early, due to the interleaved architecture of MR+, and written out to final results to avoid unnecessary I/O overhead and the correpsonding key implosion. Therefore, in MR+ only densely populated keys (keys with a higher skew) are propagated to

[5] A feature of the Linux kernel that allows it to allocate more memory than is actually available.

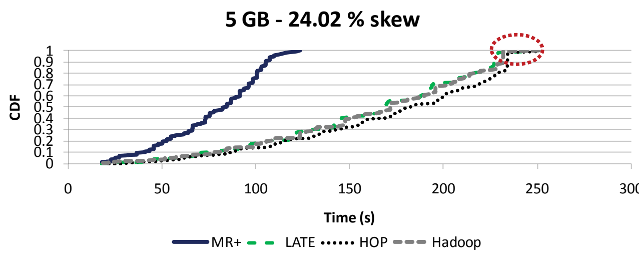


Figure 6. M = 160 and R = 20 in case of Hadoop, HOP, and LATE while R = 43 in case of MR+.

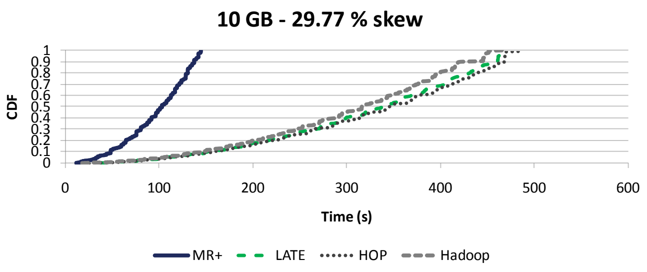


Figure 7. M = 320 and R = 20 in case of Hadoop, HOP, and LATE while R = 85 in case of MR+.

deeper levels of reduce workers; sparse keys are naturally computed (and output) early.

Note that even though the monitor performs a global optimization – the monitor needs to collect results from all the TaskTrackers – the runtime overhead is typically low since at any one snapshot only a handful of keys are returned by any one TaskTracker. Our implementation of Monitor in MR+ uses Bloom Filters [Broder 2002] for efficient comparison of keys reported by different TaskTrackers; the Monitor implementation is optimized for the common case where a monitor probe results in a high-dimension but sparsely populated structure.

4.4.5 Input Prioritization

As mentioned earlier, MR+ can be configured to run a preprocessing sampling cycle on the input dataset. In this cycle, the user defined map and reduce functions are applied to a configurable *sample_percentage* amount of input. The input for this cycle is taken from *sample_percentage* random input dataset files. This sampling cycle yields a representative distribution of data in the main dataset and is used to exploit *structure* – data with semantic grouping or clusters of relevant information.

The distribution obtained at the end of the sampling cycle is used to generate a priority queue to process structures. Map tasks are added to the FIFO priority queue in descending order of relevance according to the sampled distribution. After the priority queue has been generated, a full-fledged MR+ job is run on the original dataset. Map functions read input from the HDFS by following the priority queue. Such prioritization allows us to stop a whole job in some queries, where relevant results are clustered in a few structures. It can be argued that the sampling cycle can be used as a wrapper around vanilla Hadoop as well, to prioritize input data. This is surely possible but Hadoop provides no hooks for estimating results, i.e. even if input data is prioritized, Hadoop will still only output results when all map and reduce tasks have completed.

5. Performance Evaluation

In this section, we present a detailed evaluation of MR+ for several target applications: (1) Applications with skew in intermediate data, (2) Applications with reduce-heavy structure, (3) Applications whose input dataset has structure (clusters of relevant information) and, finally (4) Applications that require estimation of early results.

5.1 MR+ Performance with Skew

In this test, we perform a wordcount on different datasets with varying skew from the Gutenberg [Gut] library. Figure 6 and 7 show the results of this test for 2 datasets of size 5 GB and 10 GB with 24.02% and 29.77% skew, respectively. Due to Zipfian distribution of both datasets, Hadoop, HOP and LATE experience performance degradation, i.e. reduce tasks that end up with words on the higher end of the distribution take longer to compute their data than those with lighter keys. As the skew in the datasets increases, some reduce tasks take even longer to complete. MR+ on the other hand, is able to ensure proper load balancing across reduce workers by assigning comparable sized keys on the basis of locality. As shown in Figure 6, MR+ achieves a performance improvement of upto 4 times for a moderate size dataset of 10 GB with a 30% skew. Figure 6 also highlights the time spent by “extreme stragglers” that end up with an unfair distribution of work due to skew in intermediate results.

5.2 Performance of Reduce-heavy Applications

MR+ is designed specifically for reduce-heavy applications. Typically reduce takes less time as compared to maps because of the diminishing number of inputs. However, computation in the reduce phase could still be intensive, taking a large amount of time proportional to the size of the key value pairs. During experiments we have come across cases where the sizes of some keys are so large that they exceed total memory (RAM + swap space). In such cases reduce nodes often spill to disk [Gates 2009]. Such cases are quite common in the image processing community where computations like convolution of images take a relatively large portion of time.

To evaluate the performance of MR+ for these applications, we wrote an application that extracts “interesting” images from pictures taken by a large set of surveillance cameras. In our evaluation, we use a set of 70 surveillance footages, each having exactly 1200 frames with a dataset

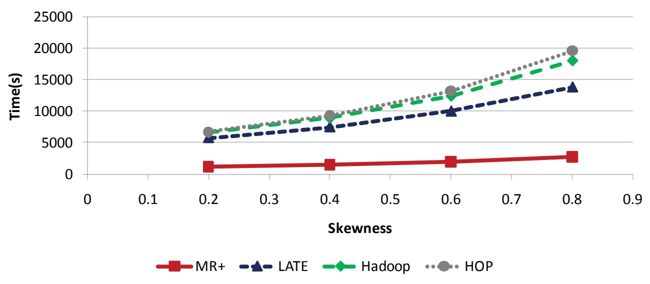


Figure 8. Image correlation application. MR+ outperforms Hadoop by almost 6.6 times for 80% skew as it can evenly divide the load between different nodes.

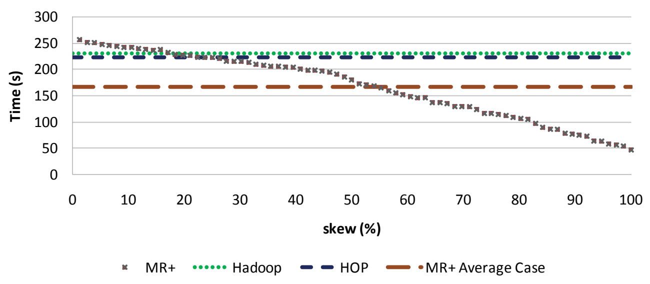


Figure 9. MR+ Gutenberg threshold tests with increasing skew against Hadoop and HOP. MR+ results in a performance improvement of a factor of 2 on average.

size of 37.5 GB. Due to the nature of surveillance, each footage has frames with varying luminance.

To compute using map-reduce, we use the map function as a filter to emit images that cross a certain threshold of brightness. This is done by checking the Brightness Value (BV) from EXIF meta-tags stored as part of the image. For simplicity, each map function is fed frames from a different camera. The images emitted by the map workers are then analyzed by reduce workers, which consolidate the results by only keeping those images that record changes across frames.

To implement this, the reduce worker computes similarity between images and only keeps one image from a set of images that cross a certain threshold of similarity. The similarity is calculated by computing a standard normalized cross-correlation between images [Lewis 1995]. Figure 5 gives the pseudo-code of the Map and Reduce functions of this application.

To evaluate the performance gain due to the architecture of MR+, we compare the performance of MR+ to Hadoop and as well as with LATE [Zaharia 2008] and Hadoop Online Prototype (HOP) [Condie 2010]. Hadoop provides a baseline comparison of our multi-stages architecture with the vanilla MapReduce architecture with a "brick-wall". We include comparison with LATE to account for the performance impact of heterogeneity in the processing capability of the nodes in our testbed. We compare with HOP to evaluate the impact of simple pipelining with the fine optimizations of MR+.

Figure 8 shows the performance comparison of Hadoop, HOP[6], LATE and MR+ for our surveillance application. For all four architectures, 1200 map tasks were spawned. Hadoop, HOP and LATE had 20 reduce tasks while MR+ had 315[7] reduce tasks – reaching a maximum final level of 6[8]. We increase the skew in the data by changing the brightness value of images from different cameras.

[6] With map pipeline enabled.

[7] Given by a *map_to_reduce_ratio* of 4 and a *reduce_input_ratio* of 20.

[8] Following equation (2).

In MapReduce, more skew in the intermediate results causes an increase in the unevenness of workload across reduce workers, concentrating the expensive cross-correlation computations at a few nodes that becomes the bottleneck. On the other hand, MR+ can evenly divide the load between different nodes, resulting in much better performance.

As mentioned in [Condie 2010] pipelining in HOP actually degrades performance in reduce intensive jobs by putting additional pressure on reduce tasks. Pipelining passes the merging work at the end of the *shuffle* phase from the map tasks to the reduce tasks. This accounts for the slow performance of HOP. The performance of LATE is in accordance with the analysis presented in [Lin 2009]. LATE effectively handles hardware heterogeneity and achieves better running time than Hadoop. But speculative execution is powerless against uneven distribution in the input or intermediate data. For all runs of the application with varying skew, the number of reduce tasks that were speculatively executed switched between 2 and 3. MR+ performs almost 6.6, 7.1 and 5 times better than Hadoop, HOP and LATE respectively with a skewness of 0.8.

5.2.1 Applications with Clustered Input

Given the multi-staged architecture of MR+, it can be used to run a sampling cycle on the input data to detect any clusters of information in the input data. In our next example, we demonstrate the use of MR+ to efficiently process data by prioritizing its input set.

For our test case, we have used the online Gutenberg library [Gut], which provides a large amount of literature categorized in 73 genres. Our application is interested in simple "exploratory" questions to find patterns in the Gutenberg database. For instance, our application is interested in finding whether any genre has more than 10,000 occurrences of words "Sherlock Holmes". Since all the books of this popular series belong to the genre of "Detective Fiction", it would help to first estimate where these books are clustered. MR+, as a first phase, computes just 5% of randomly chosen input data. Results from this phase are used to prioritize the processing of input data. Given this prioritization, the map function emits occurrences of "Sherlock Holmes" along with the name of genre as the key. Subsequently, reduce workers count the occurrences of the string "Sherlock Holmes" for each key, proceeding iteratively to balance load across reduce workers. Given the multi-level reduce of MR+, the computation is halted as soon as a reduce worker encounters a genre with more than 10,000 occurrences of "Sherlock Holmes".

Figure 9 plots results of the test where the time taken to reach the right genre is plotted against the skew in the data. The larger the skew (low entropy of string "Sherlock Holmes"), the more likely it is that the initial 5% sampling cycle would yield a more beneficial prioritization of the input data. Accuracy of results in the initial cycle eventually helps MR+ compute lesser genres to reach the threshold. The graph shows that the prioritization of data, with a 5% random sample of input files, results in better performance with increase in the skew of the data. In most cases, the application query is satisfied by just computing 3 or less genres, whereas Hadoop and HOP compute the entire 73 genres to compute an answer. As a result, in our example, MR+ results in a performance improvement of a factor of 2 on average. Between 0 and 20% skew, Hadoop and HOP perform better than MR+ due to the overhead of the MR+ sampling cycle. The performance of all 3 architectures intersects at 20% skew (though this varies on the time spent by MR+ on the sampling cycle).

5.2.2 Early Estimation of Results

To evaluate the performance of MR+ for applications that benefit from early estimation of results, we use an application that computes the average value of a volatile stock symbol (that varies between \$5 - \$91 over a year). Our application takes as input a log file that contains several hundred thousand values of the stock across a 12 months period and computes an average value.

The input file is partitioned between map workers. The final result generated at the last reduce is the mean μ of all the stock values. Figure 10 shows the output of all reduce tasks in ascending reduce level order. As we go down the reduce levels, the confidence interval keeps shrinking (or the confidence level keeps increasing). If the user is satisfied with a result within an error margin of 8% (annotated on the graph), the job can be finished at reduce level 8 – saving up to 30% of the computation time. The absolute savings in time could be much more for sophisticated applications such as backpropagation in neural networks [Chu 2006], linear regression [Ranger 2007], co-clustering [Papadimitriou 2008], tree learning [Panda 2009] and computation of node diameter and radii in Terabyte-scale graphs [Tsourakakis 2010].

6. Related Work

We present a detailed comparison of MR+ with Hadoop, HOP, SkewReduce, Mantri and LATE in Section 1.2 and Sections 4.4 and 5. We also compare MR+ to PIG, Dryad and DryadLINQ in Section 4.4 and explain how MR+ is fundamentally different from such systems since it maintains

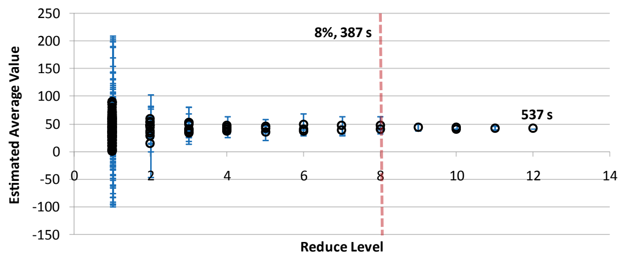


Figure 10. Estimation of the overall average value of a volatile stock. The confidence interval keeps shrinking as we go down the reduce tree.

the clean programming model of MapReduce. We are currently exploring a monitoring mechanism similar to ParaTimer [Morton 2010] that affords a progress indicator for Pig queries and estimates the progress of Pig queries in the presence of skew.

Another system relevant to MR+ is CGL MapReduce [Ekanayake 2008] which implements an inter-job iterative MapReduce architecture similar to HaLoop. CGL MapReduce also proposes a combine phase to consolidate the results of the reduce phase into a single value. HaLoop [Bu 2010] is a modification of the MapReduce framework for data analysis techniques that require iterative computations. The modified task scheduler in HaLoop terminates a job when a user defined threshold is achieved. Spark [Zaharia 2010b] also targets iterative operations by introducing *resilient distributed datasets (RDDs)*. RDDs are collections of objects in a read-only state that are replicated across partitions and can be reused iteratively in MapReduce/Dryad jobs. All of these frameworks target applications which require iteration across MapReduce jobs whereas MR+ interleaves tasks within a job to negate the effect of skew and balance workload.

7. Conclusion and Future Work

In this paper, we presented the architecture of MR+ that maintains the clean, convenient abstraction of MapReduce, but transforms its architecture from a two-staged process, with a brick-wall between the map and reduce phase, to a multi-stage architecture. In MR+, map and reduce stages are interleaved and run iteratively over the input data, either as cycles of map-reduce to prioritize the processing of input data according to the application semantics or as iterations of reduce stages to handle skew in intermediate results. The multi-stages architecture of MR+ also naturally avoids the I/O bottleneck faced by MapReduce during copying of large swathes of data from map to reduce nodes at the end of the map stage.

We have also presented preliminary evaluation of MR+ for applications in the domain of image processing, as well as examples from applications mining for patterns in large unstructured data. We have shown that MR+ outperforms

Hadoop, Hadoop Online Prototype and LATE by a factor of 1.5-7 for different applications and data sets.

The architecture of MR+ is still evolving as we continue to use it for a broader range of applications and datasets. Having removed the brick-wall between the map and reduce stages, we are currently exploring an implementation without a central master coordinating the work between two stages; since MR+ resembles a dataflow model, there is no real need for a central master. Our experiments have repeatedly highlighted that the master node is often a bottleneck and a single node of failure, especially in MR+ where it has to make many more scheduling decisions compared to MapReduce.

References

[Had] Hadoop MapReduce. `http://hadoop.apache.org/mapreduce/`.

[Gut] Project Gutenberg. `http://www.gutenberg.org/`.

[Web] The size of the World Wide Web. `http://www.worldwidewebsize.com/`.

[Adamic 2002] L. A. Adamic et al. Zipf's law and the internet. *Glottometrics*, 3, 2002.

[Ananthanarayanan 2010] G. Ananthanarayanan et al. Reining in the Outliers in Map-Reduce Clusters using Mantri. Technical Report MSR-TR-2010-69, Microsoft Research, May 2010.

[Broder 2002] Andrei Broder et al. Network applications of bloom filters: A survey. In *Internet Mathematics*, volume 1, 2002.

[Bu 2010] Y. Bu et al. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB'10*, 2010.

[Burrows 2006] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06*, 2006.

[Chen 2009] Y. Chen et al. Understanding tcp incast throughput collapse in datacenter networks. In *WREN '09*, 2009.

[Cheung 2002] D. W. Cheung et al. Effect of data skewness and workload balance in parallel data mining. *IEEE Trans. on Knowl. and Data Eng.*, 14(3), 2002.

[Chu 2006] C. T. Chu et al. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[Chuang 2006] K. Chuang et al. On Exploring the Power-Law Relationship in the Itemset Support Distribution. In *EDBT*, 2006.

[Condie 2010] T. Condie et al. MapReduce Online. In *NSDI '10*, 2010.

[Dean 2004] J. Dean et al. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*, 2004.

[Demers 2006] A. Demers et al. Towards Expressive Publish/Subscribe Systems. In *Proc. of EDBT*, 2006.

[DeWitt 1992] D. DeWitt et al. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6), 1992.

[Dikaiakos 2009] M. D. Dikaiakos et al. Cloud Computing: Distributed Internet Computing for IT and Scientific Research. *Internet Computing, IEEE*, 13(5), September 2009.

[Ekanayake 2008] J. Ekanayake et al. MapReduce for Data Intensive Scientific Analyses. In *ESCIENCE '08*, 2008.

[Elsayed 2008] T. Elsayed et al. Pairwise document similarity in large collections with mapreduce. In *HLT '08*, 2008.

[Fortunato 2008] S. Fortunato et al. Approximating PageRank from In-Degree. In *Algorithms and Models for the Web-Graph*, volume 4936 of *LNCS*. 2008.

[Gates 2009] A. F. Gates et al. Building a High-level Dataflow System on top of Map-Reduce: the Pig Experience. *Proc. of the VLDB Endowment*, 2(2), 2009.

[Gini 1971] C. W. Gini. Variability and Mutability, Contribution to the Study of Statistical Distributions and Relations, Studi Economico-Giuridici della R. Universita de Cagliari (1912). Reviewed in: Light, RJ, Margolin, BH: An Analysis of Variance for Categorical Data. *J. Amer. Stat. Assoc*, 66, 1971.

[Herlihy 1993] M. Herlihy et al. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2), 1993.

[Huang 2008] S. Huang et al. An Investigation of Zipf's Law for Fraud Detection. *Decis. Support Syst.*, 46(1), 2008.

[Isard 2007] M. Isard et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys '07*, 2007.

[Isard 2009] M. Isard et al. Quincy: Fair scheduling for distributed computing clusters. In *SOSP '09*, 2009.

[Kavulya 2010] S. Kavulya et al. An Analysis of Traces from a Production MapReduce Cluster. In *CCGrid '10*, 2010.

[Kerdprasop 2005] K. Kerdprasop et al. Weighted k-means for density-biased clustering. In *DaWaK, LNCS*, 2005.

[Kwon 2010] Y. Kwon et al. Skew-resistant Parallel Processing of Feature-extracting Scientific User-defined Functions. In *SoCC '10*, 2010.

[Lewis 1995] J. P. Lewis. Fast normalized cross-correlation. In *Vision Interface*, pages 120–123. Canadian Image Processing and Pattern Recognition Society, 1995.

[Lin 2009] J. Lin. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *LSDS-IR workshop*, 2009.

[Malewicz 2010] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD '10*, 2010.

[Morton 2010] K. Morton et al. ParaTimer: a Progress Indicator for MapReduce DAGs. In *SIGMOD '10*, 2010.

[Olston 2008] C. Olston et al. Pig Latin: a not-so-foreign Language for Data Processing. In *SIGMOD '08*, 2008.

[Panda 2009] B. Panda et al. Planet: Massively parallel learning of tree ensembles with mapreduce. *Proc. of VLDB Endow.*, 2(2), 2009.

[Papadimitriou 2008] S. Papadimitriou et al. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. In *ICDM '08*, Dec. 2008.

[Pike 2005] R. Pike et al. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4), 2005.

[Qiu 2009] X. Qiu et al. Cloud Technologies for Bioinformatics Applications. In *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, 2009.

[Ranger 2007] C. Ranger et al. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07*, 2007.

[Thusoo 2010] A. Thusoo et al. Hive - A Petabyte Scale Data Warehousing Using Hadoop. In *ICDE '10*, 2010.

[Tsourakakis 2010] C. E. Tsourakakis. Data Mining with MapReduce: Graph and Tensor Algorithms with Applications. Master's thesis, Carnegie Mellon University, March 2010.

[Volkovich 2007] Y. Volkovich et al. Determining factors behind the pagerank log-log plot. In *WAW'07*, 2007.

[Xu 2008] Y. Xu et al. Handling Data Skew in Parallel Joins in Shared-nothing Systems. In *SIGMOD '08*, 2008.

[Yu 2008] Y. Yu et al. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *OSDI '08*, 2008.

[Zaharia 2008] M. Zaharia et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI '08*, 2008.

[Zaharia 2010a] M. Zaharia et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*, 2010.

[Zaharia 2010b] M. Zaharia et al. Spark: Cluster Computing with Working Sets. In *Proc. of HotCloud*, 2010.