Minimizing Remote Accesses in MapReduce Clusters

Prateek Tandon prateekt@umich.edu
Michael J. Cafarella michjc@umich.edu
Thomas F. Wenisch twenisch@umich.edu

Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA

Abstract*—MapReduce, in particular Hadoop, is a popular framework for the distributed processing of large datasets on clusters of relatively inexpensive servers. Although Hadoop clusters are highly scalable and ensure data availability in the face of server failures, their efficiency is poor. We study data placement as a potential source of inefficiency. Despite networking improvements that have narrowed the performance gap between map tasks that access local or remote data, we find that nodes servicing remote HDFS requests see significant slowdowns of collocated map tasks due to interference effects, whereas nodes making these requests do not experience proportionate slowdowns. To reduce remote accesses, and thus avoid their destructive performance interference, we investigate an intelligent data placement policy we call 'partitioned data placement'. We find that, in an unconstrained cluster where a job's map tasks may be scheduled dynamically on any node over time, Hadoop's default random data placement is effective in avoiding remote accesses. However, when task placement is restricted by long-running jobs or other reservations, partitioned data placement substantially reduces remote access rates (e.g., by as much as 86% over random placement for a job allocated only one-third of a cluster).

I. Introduction

MapReduce [8] is a popular framework for the distributed processing of large datasets. One of the most popular implementations of the MapReduce programming model is Hadoop [1], an open-source Java implementation. Hadoop's design centers on affording scalability and availability of data. Hadoop provides scalability by making data management transparent to cluster administrators; this transparent data management allows the framework to support thousands of machines and petabytes of data. Hadoop ensures data availability (and scalability) by distributing three replicas of all data blocks that constitute a file randomly among distinct nodes. Whenever possible, Hadoop moves computation to data, as opposed to the more expensive option of moving data to computation. However, Hadoop's storage layer, the *Hadoop Distributed File System* or *HDFS* [7], facilitates remote data accesses when moving computation is not possible.

Until recently, network bandwidth has been a relatively scarce resource, and hence, conventional wisdom has held that remote data accesses should be minimized [8]. However, network performance improvements continue to outpace disk, which has led some researchers to argue that disk locality will soon be irrelevant in datacenter computing [6]. Indeed, we corroborate that this hypothesis holds even today when communicating over an unsaturated 1 Gb network—the performance gap between CPU-bound map tasks that access local and remote data (served from an idle node) is as little as 1.6%. Interestingly, however, we find that *servicing* remote HDFS requests disproportionally slows map tasks located on the same node, particularly under Linux's default "deadline" I/O scheduler (which biases scheduling to improve I/O performance; the fair I/O scheduler shrinks performance disparities at the cost of worse overall performance). For CPU-intensive map tasks, we find that *Reader* nodes, which access data from a remote node but serve no remote requests themselves, suffer only a 2% slowdown relative to local accesses. However, a *Server* node, which services remote requests while also executing map tasks, suffers a 13% slowdown. Slowdowns are much larger for I/O-bound map tasks. Hence, we conclude that, unless MapReduce clusters use dedicated storage nodes, *remote accesses must still be minimized.*

Based on these observations, we investigate intelligent data placement as a potential avenue to reduce remote accesses. We focus our investigation on the "map" phase of MapReduce jobs as initial data placement is immaterial thereafter. Hadoop's scheduler is designed to assign map tasks to nodes such that they access data locally whenever possible. When a computation resource is assigned to a job, the scheduler scans the list of incomplete map tasks for that job to find any tasks that can access locally available data. Only if no such tasks are available will it schedule a task that must perform remote accesses. Hence, jobs with dedicated access to the entire cluster rarely incur remote accesses (remote accesses only arise at the end of the map phase, when few map tasks remain, or under substantial load imbalance, for example, due to server heterogeneity [4]). However, restrictions on task assignment, because of long-running tasks, prioritization among competing jobs, dedicated allocations, or other factors, can rapidly increase the number of remote accesses.

We contrast Hadoop's default random data placement policy against an extreme alternative, *partitioned data placement*, wherein a cluster is divided into partitions, each of which contains one replica of each data block. (Note that, since the number of replicas is unchanged and placement remains random within each partition, availability is, to first-order, unchanged). By segregating replicas, due simply to combinatorial effects, we increase the probability that a large fraction of distinct data blocks is available even within relatively small, randomly selected allocations of the cluster. We further consider the utility of adding additional replicas for frequently accessed blocks, to increase the probability that these blocks will be available locally in a busy cluster.

Our evaluation, through a combination of simulation of the Hadoop scheduling algorithm and validation on a small-scale test cluster, leads to mixed conclusions:

- When scheduling is unconstrained and task lengths are well-chosen to balance load and avoid long-running tasks, Hadoop's scheduler is highly effective in avoiding remote accesses regardless of data placement, as the job can migrate across nodes over time to process data blocks locally. Under an "Unconstrained" allocation scenario, Hadoop can achieve 98% local accesses.
- However, when task allocation is constrained to a subset of the cluster (e.g., because of long-running tasks, reserved nodes, restrictions arising from job priorities, power management [16], or other node allocation constraints), partitioned data placement substantially reduces remote data accesses. For example, under a "Restricted" allocation scenario where a job may execute on only one-third of nodes (selected at random), partitioned data placement reduces remote accesses by 86% over random data placement.
- We demonstrate that selective replication of frequently accessed blocks can further reduce remote accesses in *restricted allocation* scenarios.

This paper is organized as follows: Section II provides relevant background. Section III delves into why reducing remote accesses is important even under unsaturated networks. Section IV explores the data placement policies considered in this research. Section V provides experimental results for the performance of the data placement policies under different job scheduling scenarios, and Section VI concludes.

II. Related Work

Data replication is widely used in distributed systems to improve performance when a system needs to scale in numbers and/or geographical area [23]. Replication can increase data availability, and helps achieve load balancing in the presence of scaling. For geographically dispersed systems, replication can reduce communication latencies. Hadoop leverages replication to provide both availability and scalability. Further, Hadoop places two replicas of a data block on the same rack to save inter-rack bandwidth.

Caching is a special form of replication where a copy of the data under consideration is placed close to the client that is accessing the data. Caching has been used effectively in distributed file systems such as the *Andrew File System (AFS)* and *Coda* to minimize network traffic [12], [21]. Gwertzman and Seltzer have proposed a technique of server-initiated caching called *push caching* [11]. Under this technique, a server places temporary replicas of data closer to geographical regions from which large fractions of requests are arriving. Since replication and caching imply multiple copies of a data resource, modification of one copy creates consistency issues. Much research in the distributed systems field has been devoted to efficient consistency maintenance [19], [23]. However, since Hadoop follows a write-once, read-many model for data (i.e., data files are immutable), maintaining consistency is not a concern.

In systems with distributed data replicas, achieving locality while maintaining fairness is a challenge. Isard and co-authors propose *Quincy*, a framework for scheduling concurrent distributed jobs with fine-grain resource sharing [13]. Quincy defines fairness in terms of disk-locality and can evict tasks to ensure fair distribution of disk-locality across jobs. Overall, the system improves both fairness and locality, achieving a 3.9x reduction in the amount of data transferred and a throughput increase of up to 40%.

Zaharia et al. create a fair-scheduler that maintains task locality and achieves almost 99% local accesses via *delay scheduling* [25]. Under delay scheduling, when a job that should be scheduled next under fair-scheduling cannot launch a data-local task, it stalls a small amount of time while allowing tasks from other jobs to be scheduled. However, delay scheduling performs poorly in the presence of *long tasks* (nodes do not free up frequently enough for jobs to achieve locality) and *hotspots* (certain nodes are of interest to many jobs; for example, such nodes might contain a data block that many jobs require). The authors suggest long-task-balancing and hotspot replication as potential solutions, but do not implement either. In contrast to the authors' approach, we focus on how intelligent data placement can be used to maximize MapReduce efficiency in scenarios where node allocations are restricted.

Eltabakh and co-authors present *CoHadoop* [9], a lightweight extension of Hadoop that allows applications to control where data are stored. Applications give hints to CoHadoop that certain files are related and may be processed jointly; CoHadoop then tries to co-locate these files for improved efficiency. Ferguson and Fonseca [10] highlight the non-uniformity in data placement within Hadoop clusters, which can lead to performance degradation. They propose placing data on nodes in a round-robin fashion instead of Hadoop's default data placement, and demonstrate an 11.5% speedup for the sort benchmark.

Ahmad et al. [4] observe that MapReduce's built-in load balancing results in excessive and bursty network traffic, and that heterogeneity amplifies load imbalances. In response, the authors develop *Tarazu*, a set of optimizations to improve MapReduce performance on heterogeneous clusters. Xie et al. [24] study the effect of data placement in clusters of heterogeneous machines, and suggest placing more data on faster nodes to improve the percentage of local accesses. Zaharia et al. [26] also investigate MapReduce performance in heterogenous environments. The authors design a scheduling algorithm called *Longest Approximate Time to End (LATE)*, that is robust to heterogeneity and can improve Hadoop response times by a factor of two.

Ananthanarayanan et al. [5] observe that MapReduce frameworks use filesystems that replicate data uniformly to improve data availability and resilience. However, job logs from large production clusters show a wide disparity in data popularity. The authors observe that machines and racks storing popular content become bottlenecks, thereby increasing the completion times of jobs accessing these data even when there are machines with spare cycles in the cluster. To address this problem, the authors propose a system called *Scarlett*. Scarlett accurately predicts file popularity using learned trends, and then selectively replicates blocks based on their popularity. In trace driven simulations and experiments on Hadoop and Dryad clusters, Scarlett alleviates hotspots and speeds up jobs

by up to 20.2%. We explore the utility of selective replication in combination with partitioned data placement in subsequent sections.

Prior work has also shown that well-designed data placement might allow MapReduce clusters to be dynamically resized in response to load, in an effort to increase energy efficiency without compromising data availability [15], [16].

Finally, recent research demonstrates that application demands in production datacenters can generally be met by a network that is slightly oversubscribed [14]. However, as we show subsequently, even under unsaturated network conditions, remote accesses impose significant performance penalties on nodes that service these remote requests.

III. The Cost of Remote Accesses

It is clear that remote accesses add to network traffic. When the network in a cluster is near saturation, each extra remote access contributes to longer latencies and even higher network traffic. Until recently, network bandwidth has been small compared to the combined disk bandwidth in a cluster; hence, relatively few simultaneous remote accesses can potentially constrain a network. It is therefore evident that remote accesses are best minimized under busy networks.

Perhaps more surprising, however, is our finding that remote accesses can cause performance penalties even in a low-latency network that is far from saturation (a scenario likely to become more prevalent as data center network topologies improve). As we will show, these performance penalties do not arise due to higher latency from retrieving data over the network. Indeed, to the contrary, we find that a map task accessing data locally or remotely experiences little difference in performance. Instead, we find the interference effect of *servicing* remote HDFS requests leads to a significant degradation of collocated map tasks. Hence, if all I/O can be performed locally, the peak throughput of a MapReduce cluster improves.

To study remote access overheads under unsaturated network conditions, we set up a small Hadoop cluster. We use Hadoop v0.21 on low-end servers representative of the low-cost systems often used for throughput computing clusters. Each server has eight 1.86 GHz Intel Xeon cores, and 16 GB of RAM running stock Ubuntu 10.10. This Linux release enables the "deadline" I/O scheduler (described later) by default. The inexpensive hard disks in these systems provide 50 MB/s sustained read bandwidth over HDFS. The servers are connected via a dedicated gigabit Ethernet switch.

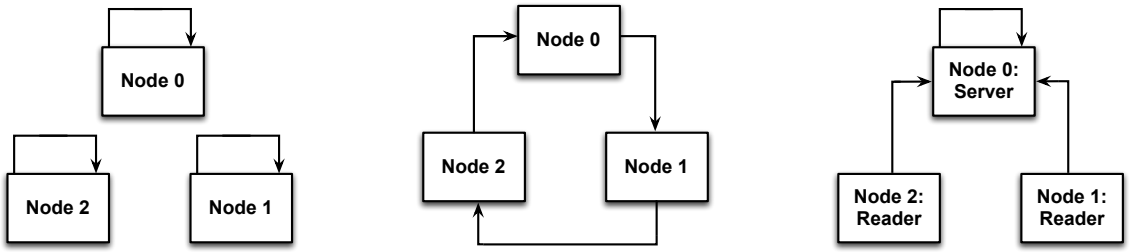
We demonstrate the cost of remote accesses using a minimal Hadoop cluster of only three datanodes and a fourth dedicated namenode. While this configuration is not representative of typical MapReduce clusters, it allows us to isolate and easily measure I/O performance effects. We use two microbenchmarks, CPU-intensive and I/O-intensive, respectively. The CPU-intensive microbenchmark runs the map task of the WordCount benchmark included with the Hadoop distribution. The I/O-intensive microbenchmark runs an empty map task. Note that since initial data placement does not matter beyond the map phase, we limit this experiment to only the map phase. The input file for both workloads is a 9.5 GB text file with 64 MB data blocks. Each datanode contains a complete copy of the file with the block replication factor set to one. We repeat experiments ten times and report averages; all reported results have 95% confidence intervals of 0.5% or better. We verify that network bandwidth never approaches saturation in any experiment. Further, to isolate the costs associated with remote accesses, we disable 7 of the 8 cores on the servers.

First, we perform a simple test contrasting the perfomance of map tasks that access local data against map tasks that access remote data served from an otherwise-idle node. This simple test isolates the impact of the network on map task performance. As predicted in recent literature [6], because the available network bandwidth exceeds the bandwidth of the disk and the data blocks transferred for each map task are large (64 MB), there is a negligble performance difference between local and remote accesses: the remote accesses incur only a 1.6% slowdown. Hence, one might conclude that remote accesses (and hence data placement) have no impact on performance.

However, this simple test neglects the interference effects of serving HDFS requests while concurrently running map tasks. We study these interference effects through three data access scenarios that are illustrated in Figure 1. In the first scenario (*Local*), each datanode runs only map tasks that access local data; we normalize runtimes to this baseline. In the second scenario (*Remote*), each datanode runs only map tasks that access remote data from the node with the next higher ID (modulo the number of nodes); hence, all nodes act concurrently as both readers and servers. In the final scenario (*Asymmetric*), two datanodes (the *Readers*) access data located on the third (the *Server*), which additionally runs its own map tasks that access data locally. In all scenarios, each map task accesses distinct files, and file system caches are initially empty.

Table I shows the normalized runtimes for each combination of I/O scenario and microbenchmark. Runtimes are normalized to the *Local* case for each microbenchmark. For the CPU-intensive microbenchmark, we see in the *Remote* scenario that remote access results in a 10% performance penalty, considerably larger than the 1.6% penalty of traversing the network to access an idle HDFS server. In the *Asymmetric* scenario, we see a further interesting effect: the slowdown on the *Reader* nodes (which serve no remote HDFS requests) shrinks to only the network-traversal penalty, while the server node sees a disproportionate slowdown of 13%. For the I/O-intensive microbenchmark, the penalties are magnified. In the *Remote* scenario, the slowdown grows to 30%. The *Asymmetric* scenario sees even larger slowdowns. Overall, we observe that map tasks running on the *Server* node see a disproportionate slowdown (i.e., they are not receiving a fair share of disk bandwidth).

To explain the behavior observed above, we configure four nodes in the *Asymmetric* mode, and vary the number of nodes reading from the *Server* node from one to four (one of these readers is always present on the *Server*). Figure 2 and Figure 3 show the breakdown of runtime spent in various CPU states on the *Server* and a *Reader* for the CPU-intensive and I/O-intensive microbenchmarks respectively. As the number of

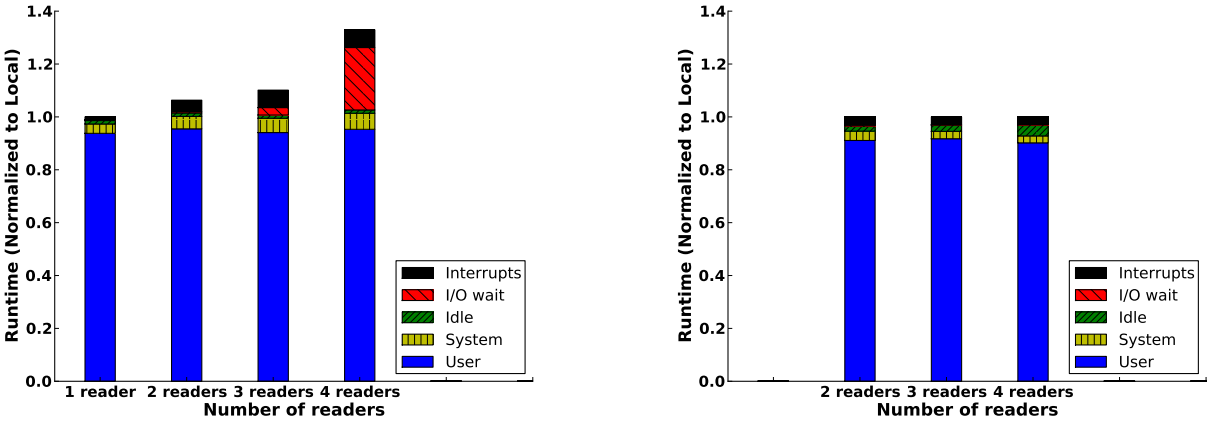


(a) Local (b) Remote (c) Asymmetric

Fig. 1: Experimental configurations for characterizing the costs associated with remote accesses: Arrows indicate read requests.

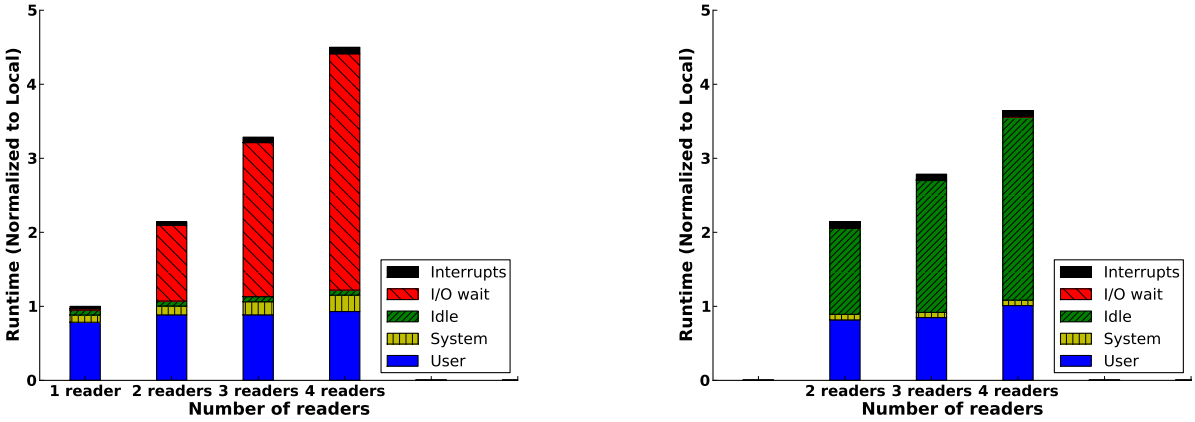
Microbenchmark	Local	Remote	Asymmetric	
			Reader	*Server*
CPU-intensive	1.0	1.1	1.02	1.13
I/O-intensive	1.0	1.3	2.65	3.14

TABLE I: Runtimes for Various I/O Configurations
(Normalized to Local for each microbenchmark)



(a) Server (b) Reader

Fig. 2: Runtime breakdown on the Server and representative Reader for the CPU-intensive microbenchmark: (a) As the number of readers increases, the CPU on the Server spends more time in I/O wait. (b) I/O wait associated with the read request is effectively masked on the Reader.



(a) Server (b) Reader

Fig. 3: Runtime breakdown on the Server and representative Reader for the I/O-intensive microbenchmark: (a) As the number of readers increases, the Server spends more time in I/O wait. (b) I/O wait associated with the read request is not effectively masked on the Reader, and is seen as increased idle time.

readers accessing the *Server's* disk goes up, the *Server* spends increasing amounts of time in the CPU I/O-wait state. In other words, the CPU is stalled on I/O and starved for data to process. (To verify that the *Server's* CPU is stalled on I/O and does not have any other tasks to process, we run a separate CPU-intensive task on the *Server* and observe that the I/O wait gets transformed into user time.)

The increased runtime and I/O wait on the *Server* stems from inherent inefficiencies in HDFS. The HDFS client implementation is highly serialized for data reads [22]. In the absence of pipelining to overlap computation with I/O, the application waits for data transfer to complete before processing commences. Additionally, each datanode (the *Server* in this case) spawns one thread per client to manage disk access and network communication, with all threads accessing the disk concurrently; these threads consume valuable resources. Shafer et al. [22] observe that as the number of concurrent readers in HDFS increases from one to four, the aggregate read bandwidth reduces by 21% with *UFS2*, and by 42% with *ext4*. Further, the average run-length-before-seeking drops from over 4MB to well under 200kB. Since most I/O schedulers are designed for general-purpose workloads and attempt to provide fairness at a fine-grained granularity of a few hundred kilobytes, the disk is forced to switch between distinct data streams in the presence of multiple concurrent readers, thereby lowering aggregate bandwidth.

Our experimental observations show that for the CPU-intensive microbenchmark, the *Server* node experiences almost an 8x increase in disk-queue size when going from 2 readers to 4 readers—from 0.42 to 3.12. Further, the period between request issue and data reception on the *Server* quadruples—from 2.2ms to 8.9ms, thereby accounting for the I/O wait observed. Disk utilization is also observed to increase from 31% to 94%.

For the I/O-intensive microbenchmark, the *Server* experiences over a 2x increase in disk-queue size when going from 2 readers to 4 readers—from 2.2 to 4.8. As with the CPU-intensive microbenchmark, the period between request issue and data reception quadruples—from 2.2ms to 8.9ms. Disk utilization is seen to increase from about 92% to about 98%.

We note from Figure 2 and Figure 3 that the I/O wait seen on *Server* does not translate into an equivalent amount of idle time on the *Readers*. For the CPU-intensive microbenchmark, a majority of the delay associated with I/O wait on the *Server* node is masked by computation on the *Readers*. However, this masking is not observed on the *Readers* for the I/O-intensive microbenchmark, and is translated into CPU-idle time. As a consequence, for the I/O-intensive microbenchmark, the *Readers* see large increases in runtime. Overall, we note that the slowdown in the *Remote* scenario can primarily be attributed to I/O stalls accounting for a larger fraction of the runtime. And the even larger slowdowns in the *Asymmetric* case can further be attributed to the saturation of the available bandwidth on the (single) disk serving all three nodes.

The central conclusion from these results is that *serving remote HDFS requests disproportionately delays locally-running map tasks*. The eventual source of this performance phenomenon can be traced to the interaction of the threading

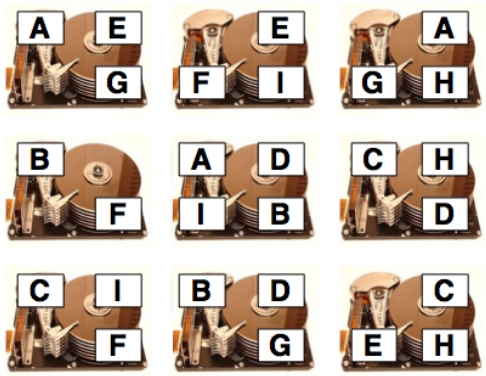


Fig. 4: Random data placement in a Hadoop cluster: Each block is replicated three times across the cluster. Each disk represents a node in the cluster.

model of HDFS and the default I/O scheduler in recent versions of Linux, the "deadline" scheduler [3]. The deadline scheduler imposes a deadline on all I/O operations to ensure that no request gets starved, and aggressively reorders requests to ensure improvement in I/O performance. Because of these deadlines, a sleeping HDFS thread that receives an I/O completion (either a new request arriving over the network or data being returned from the disk) will preempt a map task nearly immediately to finish the I/O. In contrast, map tasks that issue an I/O request, block on the I/O, thereby freeing a core to allow HDFS to execute without disturbing other map tasks.

We can eliminate the unfairness caused by the deadline scheduler by instead switching Linux to use a *completely fair scheduler* [2]. However, we find that, when doing so, overall performance suffers: in the 4-reader case, while running the CPU-intensive microbenchmark, the *Server* node slows down by an additional 6%, while the *Reader* nodes slow down by 16%, to only about 6% faster than the *Server*. The difference in runtimes between the *Server* and the *Readers* with the completely fair scheduler is consistent with the extra resources the *Server* has to sacrifice in order to service the remote requests. We also note that with the completely fair scheduler, each *Reader* sees an increase in idle time that is consistent with the I/O wait time seen on the *Server*.

The aforementioned experiments demonstrate that nodes servicing remote read requests are slowed down more significantly than nodes making these requests. Overall, our observations indicate that reducing remote accesses can improve performance even in scenarios where the network is far from saturated, and not just in highly loaded clusters with multi-job loads, busy networks, and file fragmentation associated with multiple simultaneous writers. In the next sections, we explore intelligent data placement as one avenue to reduce remote accesses.

IV. Data Placement

In this section, we propose *partitioned data placement* as an approach to reduce remote accesses. We first describe Hadoop's default *random data placement* as a baseline for comparison.

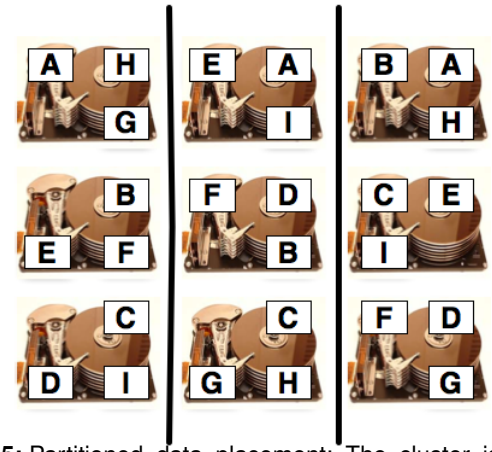


Fig. 5: Partitioned data placement: The cluster is partitioned into three sub-clusters (the vertical lines demarcate the partitions). Each partition contains one replica of each data block. Overall, the cluster still contains three replicas of each block.

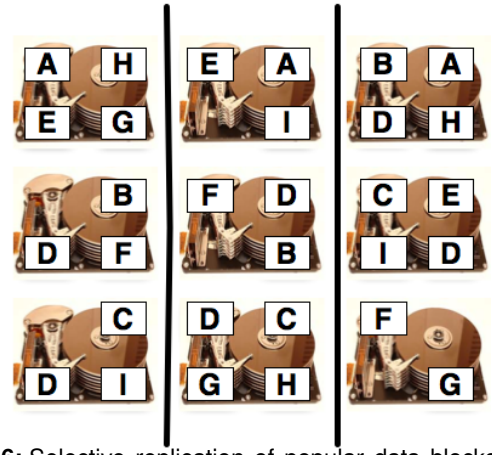


Fig. 6: Selective replication of popular data blocks within partitions: Block D is a popular block and is replicated twice within each partition. Non-popular blocks are replicated once per partition.

A. Random Data Placement

Under random data placement, blocks are randomly distributed across nodes. Figure 4 illustrates this data placement policy. Random data placement is a simplication of Hadoop's default data placement scheme (Hadoop's default locality optimization seeks to collocate two of three data replicas within a single rack; rack locality is irrelevant when network distance has no performance impact, as in the scenarios we study). Although random data placement maintains data availability in the presence of disk failures, it can be inefficient from a performance perspective, especially when a job is restricted from executing on some machines within a cluster.

The drawbacks of random data placement under restricted task scheduling are illustrated using Figure 4. Consider a cluster wherein only the three leftmost nodes may service a job (e.g., because other nodes are reserved). We define the *allocated cluster fraction* as the fraction of the cluster available to a job. In the example in the figure, the allocated cluster fraction is 3/9 or 33%. Under this scenario, it is clear that blocks *D* and *H* are not locally accessible within the available nodes, and hence must incur remote accesses.

B. Partitioned Data Placement

To reduce remote accesses when a job is restricted to a subset of the cluster, we propose *partitioned data placement*. In partitioned data placement, a cluster is divided into *N* partitions, with *N* being equal to the replication factor. Each partition contains exactly one replica of every data block; overall, the entire cluster contains *N* replicas of each block. Blocks are randomly assigned to nodes within a particular partition. Figure 5 shows an example of this data placement policy for a cluster with three partitions.

When a job can be assigned an entire partition (or more), all data can be accessed locally. Hence, if the number of active jobs is less than the number of partitions, remote accesses will be rare (arising only due to load imbalance at the tail of the job). However, as subsequently demonstrated, partitioned data placement reduces remote accesses even for jobs that execute on a smaller number of nodes. The data placement restrictions implied by partitioning reduce the probability of duplicate data blocks in a randomly selected subset of nodes, thus increasing the diversity of blocks available locally. To first order, partitioning does not sacrifice data availability since the overall number of replicas remains unchanged.

C. Replication

In addition to data placement, creating more replicas of a block can improve the probability the block will be available locally within a random subset of the cluster. But, adding extra replicas comes at a high storage cost (e.g., one extra replica per block implies a 33% storage increase). With knowledge of block access patterns, only the most popular (frequently accessed) blocks can be replicated, reducing replication costs while maintaining much of the benefit [5]. We explore the impact of selective replication similar to that proposed by Ananthanarayanan and co-authors in combination with partitioned placement. Figure 6 illustrates the concept of *selective replication* in partitioned clusters; a popular block (in the example, block *D*), is replicated at a higher replication factor within each partition.

V. Results

We contrast random and partitioned data placement through a combination of simulation of the Hadoop scheduling algorithm and validation experiments on a small scale cluster. We use simulation to allow rapid exploration of the impacts of data placement policies on clusters much larger than the real clusters to which we have access.

We contrast random and partitioned data placement policies under two scenarios: *unconstrained* and *restricted* allocation. Under unconstrained allocation, the tasks that constitute a job may be scheduled on any node in the cluster. A job is granted an allocation that limits the maximum number of simultaneously executing tasks; however, there is no restriction on the nodes that run these tasks. When multiple jobs execute

concurrently, with a suitable scheduling discipline (e.g., round robin), over time, a job's tasks will visit a time-varying subset of nodes. Because the job migrates across the cluster over time, a large fraction of data blocks can be accessed locally at some point during the job's execution, even when the job is granted a small (simultaneous) allocation.

Under restricted allocation, we assign a job to execute within only a fixed (but randomly selected) subset of nodes. The restricted allocation scenario is representative of a variety of reasons that Hadoop jobs might be precluded from executing on some nodes. The simplest example is when nodes are explicitly reserved for certain jobs or users; however, restricted allocation might also arise because nodes are rendered unavailable due to long-running tasks, job priorities, power managment, or the job scheduling discipline.

A. Simulation Methodology

We model large-scale Hadoop clusters by extending the *BigHouse* data center simulation framework [17], [18]. BigHouse is a parallel, stochastic discrete time simulation framework that represents datacenters via generalized queuing models driven by empirically-observed arrival and service distributions. Our simulation assigns MapReduce tasks lengths drawn from a service time distribution and assigns tasks to nodes in a manner similar to Hadoop's scheduler. When a task slot becomes available on a node, the scheduler checks to see if a local block required by the job is available on that node. If so, the newly created task is assigned this local block. If there are no local blocks that are awaiting processing, the scheduler picks a pending block from the closest remote node and assigns it to the new task.

We simulate a 60 node cluster that stores 10 files with up to 1200 blocks each. Block popularity is drawn from a Zipf distribution. Task execution times are drawn either from an exponential distribution with rate parameter λ=1, or a gamma distribution with shape parameter k=2 and scale parameter θ=3. The baseline replication factor for both random and partitioned data placement is set to Hadoop's default of three. For partitioned data placement, we assume three partitions, i.e., one replica per partition.

1) Unconstrained Allocation: We first consider unconstrained allocation, wherein a task may be assigned to any node. Under this scenario, provided job lengths are reasonably balanced, the tasks constituting a job will be able to migrate across the cluster over time to visit each data block such that they can access it locally. We assume that jobs are sliced into tasks at the granularity of disk blocks; finer granularity can result in higher overhead from task startup and shutdown [22], while coarser granularity may restrict task scheduling flexibility. Recent research has suggested that relatively large data block sizes improve performance [20].

Under unconstrained allocation, both partitioned and random data placement perform similarly: approximately 98% of data blocks can be accessed locally (98.2% local accesses for partitioned, and 97.9% for random). On average, a job incurs its first remote access only after over 85% of tasks have been processed, i.e., towards the tail end of the job.

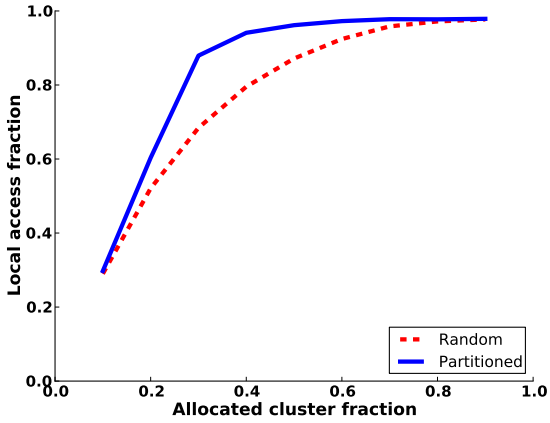


Fig. 7: Local access fraction as a function of cluster availability: Partitioned data placement dominates random data placement, especially for low cluster allocations.

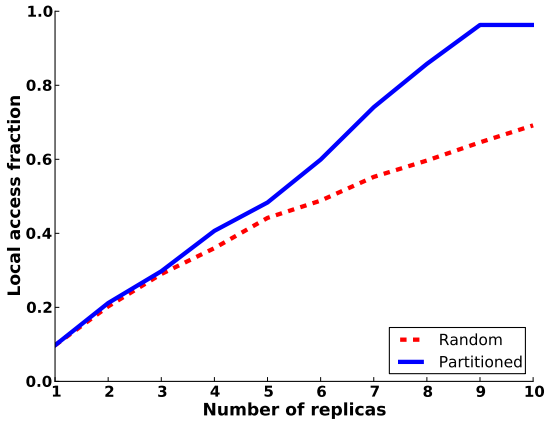


Fig. 8: Local access fraction versus number of replicas for 10% cluster availability: Adding more replicas increases the number of local accesses.

Since tasks are scheduled on a per block basis, a job gets the opportunity to visit multiple nodes through the cluster, thereby increasing the likelihood that tasks will be scheduled on nodes that contain locally accessible blocks. The key take-away is that data placement has little impact when it is possible for a job to traverse the cluster and visit nearly all nodes over time. Even under a naive round-robin scheduler across competing jobs, almost 98% of accesses can be completed locally. Hence, under these circumstances, neither partitioned placement, nor other techniques (e.g., delay scheduling) are necessary.

2) Restricted Allocation: Under restricted allocation scenarios, partitioned data placement can be effective in reducing remote accesses. Figure 7 shows the fraction of blocks accessed locally for both random and partitioned data placement as a function of the fraction of the cluster allocated to a job (i.e., the fraction of nodes the job may visit), which we vary from 10% to 90%. Under random data placement, a job must be allocated nearly 80% of the cluster to avoid remote accesses. Stated another way, if more than 20% of a cluster is reserved and may not be used by a job, then the job will suffer an increased rate of remote data accesses. Hence, even relatively small allocations of much less than half the

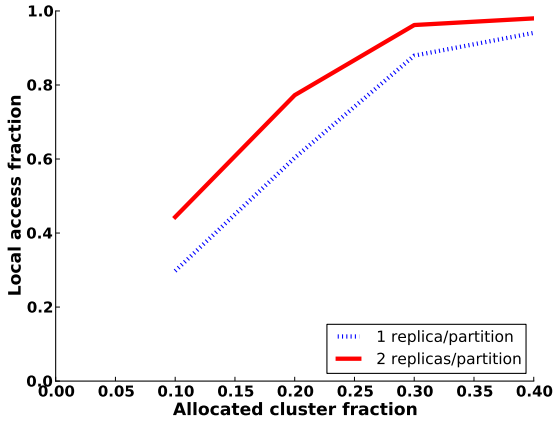


Fig. 9: Effect of selective replication of popular blocks on local access fraction: Selectively replicating popular blocks increases the number of local accesses with low storage overhead.

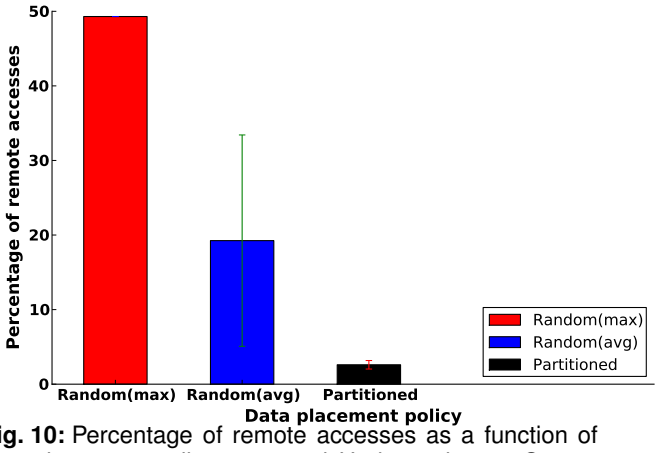


Fig. 10: Percentage of remote accesses as a function of data placement policy on a real Hadoop cluster: On average, partitioned data placement reduces the number of remote accesses by over 86% for a 33% cluster allocation.

cluster can substantially impact local versus remote access rates under Hadoop's default placement scheme. Partitioned data placement substantially improves the fraction of local accesses for cluster allocations from 10% to 80%. Once the allocation fraction is larger than a partition (33% for the 3-partition cluster in this experiment), the local access fraction rapidly approaches 100%.

3) Replication: Next we study the effect of adding more replicas on local access rates under restricted allocations. We first consider the simple case where all blocks are replicated, ignoring popularity. Figure 8 shows a graph of the local access fraction as a function of the replication factor for 10% cluster allocation, sweeping the number of replicas from one (no replication) to ten. For the partitioned scheme, the number of replicas also corresponds to the number of partitions. As the number of replicas increases, the advantage of partitioned data placement over random data placement grows.

We next consider only selective replication of blocks that exceed a popularity threshold. To model this scenario, we increase the popularity of two files (corresponding to 22% of all blocks) by a factor drawn from a Zipf distribution relative to the remaining blocks. We provision twice as many replicas of blocks in the popular files. These additional replicas are again distributed across the partitions, such that there are now two replicas of the popular blocks per partition. Figure 9 shows the effect of such selective replication, relative to a baseline of only a single replica per partition for all blocks. Selective replication increases the rate of local accesses by 10-20% over the relevant range of cluster allocations (above 40% allocation, nearly all accesses are local without additional replicas). Of course, selective replication results in a far lower storage overhead than naive replication of all blocks.

B. Validation on a Real Hadoop Cluster

We validate our simulation results via a small-scale test on a 10-node Hadoop cluster (nine datanodes and one dedicated namenode). We contrast Hadoop's default random data placement against partitioned data placement. In both cases, we distribute a 9.5 GB file across nodes in 64 MB blocks with a replication factor of three. We emulate restricted allocation by marking six randomly selected nodes as unavailable to execute tasks, corresponding to a 33% cluster allocation. We report the fraction of map tasks that access remote data.

Figure 10 shows the percentage of remote accesses for each data placement policy. Over a series of ten trials, random data placement requires a maximum of 48.7% of map tasks to access remote data. On average, random data placement results in 19.2% remote accesses. Partitioned data placement reduces the average number of remote accesses to 2.6%, an 86% reduction compared to random data placement. In summary, our results show that the partitioned data placement policy reduces remote accessess relative to random data placement under restricted allocation scenarios. Additionally, increasing replication factors can further reduce remote accesses, especially for small allocations.

VI. Conclusion

MapReduce, in particular Hadoop, is a popular framework for the distributed processing of large datasets on clusters of networked and relatively inexpensive servers. Whereas Hadoop clusters are highly scalable and ensure data availability in the face of server failures, their efficiency is poor. We demonstrate that remote accesses can cause significant performance degradation, even under unsaturated network conditions, due to the disproportionate interference effects on nodes servicing remote HDFS requests. We study an intelligent data placement policy we call *partitioned data placement* as an avenue to reduce the number of remote data accesses, and the associated performance degradation, when task placement is restricted due to reasons such as long-running jobs or other reservations. During the course of our investigation, we find that partitioned data placement can reduce the number of remote data accesses by as much as 86% when a job is restricted to execute on only one-third of the nodes in a cluster.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of National Science Foundation grants IIS-1054913, IIS-1064606, OCI-1047871, and CSR-0834403, and US Department of Energy Award DE-SC0005026 (please note disclaimer at http://www.hpl.hp.com/DoE-Disclaimer.html).

REFERENCES

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

[2] IBM: Inside the Linux 2.6 Completely Fair Scheduler. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler.

[3] Red Hat Corporation: Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel. http://www.redhat.com/magazine/008jun05/features/schedulers.

[4] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[5] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proceedings of the European Conference on Computer Systems*, 2011.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-Locality in Datacenter Computing Considered Irrelevant. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2011.

[7] D. Borthakur. *The Hadoop Distributed File System*. Apache Software Foundation, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.

[9] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: Flexible Data Placement and its Exploitation in Hadoop. In *Proceedings of the VLDB Endowment*, 2011.

[10] A. D. Ferguson and R. Fonseca. Understanding Filesystem Imbalance in Hadoop. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[11] J. S. Gwertzman and M. Seltzer. The Case for Geographical Push-Caching. In *Workshop on Hot Topics in Operating Systems*, 1995.

[12] J. H. Howard. An Overview of the Andrew File System. In *USENIX Winter Technical Conference*, 1988.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 2009.

[14] S. Kandula, J. Padhye, and P. Bahl. Flyways To De-Congest Data Center Networks. In *Proceedings of the Workshop on Hot Topics in Networks*, 2009.

[15] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. In *Proceedings of the VLDB Endowment*, 2010.

[16] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. *SIGOPS Operating Systems Review*, 44(1), 2010.

[17] D. Meisner and T. F. Wenisch. Stochastic Queuing Simulation for Data Center Workloads. In *Proceedings of the Workshop on Exascale Evaluation and Research Techniques*, 2010.

[18] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, 2012.

[19] S. Mullender(ed.). *Distributed Systems*. ACM Press, 1989.

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[21] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput.*, 39(4), Apr. 1990.

[22] J. Shafer, S. Rixner, and A. Cox. The Hadoop Distributed Filesystem: Balancing Portability and Performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*, 2010.

[23] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2002.

[24] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving MapReduce Performance Through Data Placement in Heterogeneous Hadoop Clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010.

[25] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the European Conference on Computer Systems*, 2010.

[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2008.