Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters

Ganesh Ananthanarayanan[∓], Sameer Agarwal[∓], Srikanth Kandula[◇],
Albert Greenberg[◇], Ion Stoica[∓], Duke Harlan[†], Ed Harris[†]

[∓] UC Berkeley [◇] Microsoft Research [†] Microsoft Bing

{ganesha, sameerag, istoica}@cs.berkeley.edu, {srikanth, albert, dukehar, edharris}@microsoft.com

Abstract

To improve data availability and resilience MapReduce frameworks use file systems that replicate data *uniformly*. However, analysis of job logs from a large production cluster shows wide disparity in data popularity. Machines and racks storing popular content become bottlenecks; thereby increasing the completion times of jobs accessing this data even when there are machines with spare cycles in the cluster. To address this problem, we present Scarlett, a system that replicates blocks based on their popularity. By accurately predicting file popularity and working within hard bounds on additional storage, Scarlett causes minimal interference to running jobs. Trace driven simulations and experiments in two popular MapReduce frameworks (Hadoop and Dryad) show that Scarlett effectively alleviates hotspots and can speed up jobs by 20.2%.

Categories and Subject Descriptors D.4.3 [*Operating Systems*]: File Systems Management–Distributed file systems

General Terms Algorithms, Measurement, Performance

Keywords Datacenter Storage, Locality, Fairness, Replication

1. Introduction

The MapReduce framework has become the de facto standard for large scale data-intensive applications. MapReduce based systems, such as Hadoop [Hadoop], Google's MapReduce [Dean 2004], and Dryad [Isard 2007] have been deployed on very large clusters consisting of up to tens of thousands of machines. These systems are used to process large datasets (e.g., to build search indices or refine ad placement) and also in contexts that need quick turn-around times (e.g., to render map tiles [Dean 2009]). These deployments represent a major investment. Improving the performance of MapReduce improves cluster efficiency and provides a competitive advantage to organizations by allowing them to optimize and develop their products faster.

MapReduce jobs consist of a sequence of dependent *phases*, where each phase is composed of multiple *tasks* that run in parallel. Common phases include map, reduce and join. Map tasks read data from the disk and send it to subsequent phases. Since the data is distributed across machines and the network capacity is limited, it is desirable to *co-locate* computation with data. In particular, co-locating map tasks with their input data is critical for job performance since map tasks read the largest volume of data. All MapReduce based systems go to great lengths to achieve data locality [Dean 2004, Isard 2009, Zaharia 2010].

Unfortunately, it is not always possible to co-locate a task with its input data. The uniform data replication employed by MapReduce file systems (e.g., Google File System [Ghemawat 2003], Hadoop Distributed File System (HDFS) [HDFS]) is often sub-optimal. Replicating each block on three different machines is not enough to avoid *contention* for slots on machines storing the more popular blocks. Simply increasing the replication factor of the file system is not a good solution, as data access patterns vary widely in terms of the total number of accesses, the number of concurrent accesses, and the access rate over time. Our analysis of logs from a large production Dryad cluster supporting Microsoft's Bing, shows that the top 12% of the most popular data is accessed over ten times more than the bottom third of the data. Some data exhibits high access concurrency, with 18% of the data being accessed by at least three unique jobs at a time.

Contention for slots on machines storing popular data may hurt job performance. If the number of jobs concurrently accessing a popular file exceeds the number of replicas, some of these jobs may have to access data remotely and/or compete for the same replica. Using production traces, we estimate that as a direct consequence of contentions to popular files, the median duration of Dryad jobs increases by 16%.

To avoid contentions and improve data locality, we design a system, Scarlett, that *replicates files based on their access patterns and spreads them out to avoid hotspots, while minimally interfering with running jobs*. To implement this approach it is critical to accurately predict data popularity. If we don't,

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

Dates	Phases ($x10^3$)	Jobs ($x10^3$)	Data (PB)	Network (PB)
May 25,29	44.3	23.4	35.5	1.55
Aug 20,24	77.1	48.8	47.7	2.10
Sep 15,19	74.4	40.1	54.0	1.82
Oct 15,19	49.0	33.0	69.5	2.17
Nov 16,20	96.4	45.3	54.4	1.70
Dec 10,14	46.4	42.4	51.9	1.40

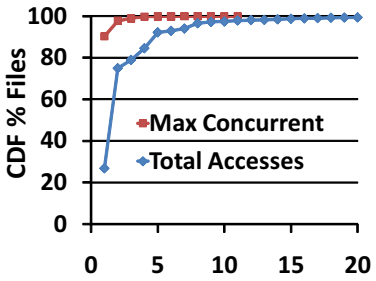
Table 1: **Details of Dryad job logs collected from Microsoft Bing's cluster.**

we may either create too few replicas thus failing to alleviate contention, or create too many replicas thus wasting both storage and network bandwidth. To guide replication, Scarlett uses a combination of historical usage statistics, online predictors based on recent past, and information about the jobs that have been submitted for execution. To minimize interference with jobs running in the cluster, Scarlett operates within a storage budget, replicates data lazily, and uses compression to trade processing time for network bandwidth. Finally, Scarlett benefits from spreading out the extra replicas, and hence cluster load, on machines and racks that are lightly loaded.

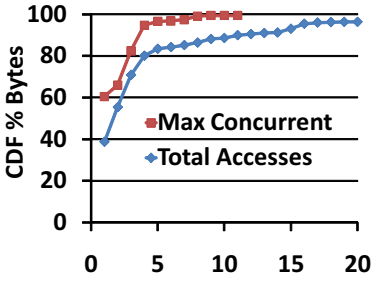
We evaluate the benefits of Scarlett's replication scheme in the context of two popular MapReduce frameworks, Hadoop [Hadoop] and Dryad [Isard 2007]. These frameworks use different approaches to deal with data contention. While Hadoop may run a late-arriving task remotely, Dryad aims to enforce locality by evicting the low priority tasks upon the arrival of higher priority tasks [Isard 2009]. We note that these approaches, as well as new proposals to improve locality in the presence of contention by delaying the tasks [Zaharia 2010], are orthogonal and complementary to Scarlett. Indeed, while these schemes aim to minimize the effect of contentions, Scarlett seeks to avoid contentions altogether. By providing more replicas, Scarlett makes it easier for these schemes to co-locate data with computation and to alleviate contention to popular data.

We have deployed Scarlett on a 100-node Hadoop cluster, and have replayed the workload traces from Microsoft Bing's datacenter. Scarlett improves data locality by 45%, which results in a 20.2% reduction of the job completion times of Hadoop jobs. In addition, by using extensive simulations, we show that Scarlett reduces the number of evictions in the Dryad cluster by 83% and speeds up the jobs by 12.8%. This represents 84% of the ideal speedup assuming no contention. Finally, we show that Scarlett incurs low overhead, as it is able to achieve near-ideal performance by altering replication factors only once in 12 hours, using less than 10% extra storage space, and generating only 0.9% additional network traffic.

The rest of the paper is outlined as follows. In §2, we quantify the skew in popularity and its impact. §3 presents a solution to cope with the popularity skew using adaptive and efficient replication of content. We look at how Dryad and Hadoop are affected by popularity skew in §4. §5 evaluates the performance benefits of Scarlett. We discuss related work in §6 and conclude in §7.



(a) File Accesses



(b) Byte Accesses

Figure 1: **File and Byte Popularity: CDFs of the total numbers of jobs that access each file (or byte) as well as the number of concurrent accesses. The x-axis has been truncated to 20 for clarity, but goes up to 70.**

2. Patterns of Content Access in Production

We analyzed logs from a large production cluster that supports Microsoft Bing over a six month period in 2009 (see Table 1). The cluster software, known as Cosmos, is a modified form of Dryad [Isard 2007] that runs jobs written in Scope [Chaiken 2008]. The dataset contains more than 200K jobs that processed over 300 petabytes of data of which over 10PB were moved on the network across racks. The logs contain details of individual tasks as well as dependencies across phases (map, reduce, etc.). For each task, we record its start and end times, the amount of data it reads and writes and the location in the cluster where the task ran and where its inputs were drawn from. We also obtain similar data at the granularity of phases and jobs. The cluster primarily ran production jobs. Some of these are mining scripts that repeatedly ran as new data arrives.

We examine the variation in popularity across files and how popularity changes over time. We also quantify the effects of popularity skew – hotspots in the cluster.

2.1 Variation in Popularity

Since accesses to content are made by jobs, we examine popularity at the smallest granularity of content that can be addressed by them. We colloquially refer to this unit as a *file*. In practice, this smallest unit is a collection of many blocks and

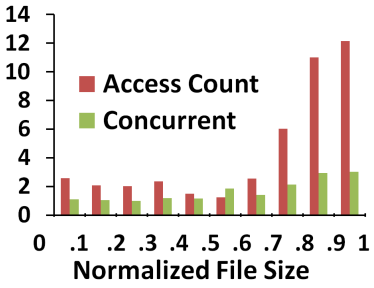


Figure 2: **Popularity of files as a function of their sizes, normalized to the largest file; the largest file has size 1. The columns denote the average value (# accesses, concurrence) of files in each of the ten bins.**

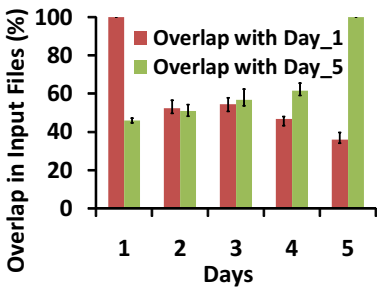


Figure 3: **Overlap in files accessed across five days, for each month listed in Table 1. With the first and fifth day as references, we plot the fraction of bytes accessed on those days that were also accessed in the subsequent and preceding days, respectively.**

often has semantic meaning associated with it such as records within a certain time range from a data stream.

There is a large variation among files in their number of accesses as well as in their number of concurrent accesses. Figure 1a plots CDFs over files of the total number of tasks that access each file and the maximum number of tasks that concurrently access each file. The figure shows that 2.5% of the files are accessed more than 10 times and 1.5% of the files are accessed more than three times concurrently. On the other hand, a substantial fraction of the files are accessed by no more than one task at a time (90%) and no more than once over the entire duration of the dataset (26%).

Files vary in size, so to examine the byte popularity, Figure 1b weights each file by its size. Compared to Figure 1a, we see that both CDFs move to the right, indicating that more fraction of the bytes are in files that have more accesses. We see that 38% of all data is accessed just once in the five-day interval. On the other hand, 12% of the data is accessed more than 10 times, i.e., 12% of the data is 10x more popular than roughly a third of the data. Recall that each block in the file system is replicated three times. The figure shows that 18% of the data have at least three concurrent accesses, i.e., are operating *at brim*, while 6% of them have more concurrent accesses than the number of replicas.

We believe that popularity skew in MapReduce clusters arises due to a few reasons. Due to abundantly available storage, a lot of data is logged for potential future analysis but only a small fraction is ever used. Some other datasets, however, correspond to production pipelines (e.g., process newly crawled web content) and are always used. Their popularity spikes when the data is most fresh and decays with time. The sophistication of analysis and the number of distinct jobs varies across these production datasets. In contrast to the popularity skew observed in other contexts (e.g., of web and peer-to-peer content), we see that the hottest content is not as hot and that there is more moderately hot content. Likely this is because the content consumer in this context (business groups for research and production purposes) have wider attention spans, are more predictable, and less peaky as compared to consumers of web and p2p content.

When multiple tasks contend for a few replicas, the machines hosting the replicas become hotspots. Even if these tasks ran elsewhere in the cluster, they compete for disk bandwidth at the machines hosting the replicas. When the cluster is highly utilized, a machine can have more than one popular block. Due to collisions between tasks reading different popular blocks, the effective number of replicas per block can be fewer as some of the machines hosting its replicas are busy serving other blocks.

We find high correlation between the total number of accesses and number of concurrent accesses with a Pearson's correlation factor of 0.78, implying that either of these metrics is sufficient to capture file popularity.

We also find that large files are accessed more often. Figure 2 bins files by their size, with the largest file having a normalized size of 1, and plots the average number of accesses (total and concurrent) to files in each bin. Owing to their disproportionately high access counts, focusing on just the larger files is likely to yield most of the benefits.

2.2 Change in Popularity

Files change in popularity over time. Figure 3 plots the overlap in the set of files accessed across five consecutive days for each month listed in Table 1, with *day-1* and *day-5* as references. We observe a strong day effect – only 50% of the files accessed on any given day are accessed in the next or the previous days. Beyond this initial drop, files exhibit a gradual ascent and decline in popularity. Roughly 40% of the files accessed on a given day are also accessed four days before or after. The relatively stable popularity across days indicates potential for prediction techniques that learn access patterns over time to be effective.

On an hourly basis, however, access patterns exhibit not only the gradual ascent and decline in popularity that we see over days but also periodic bursts in popularity. Figure 4 plots hourly overlap in the set of files accessed, with two

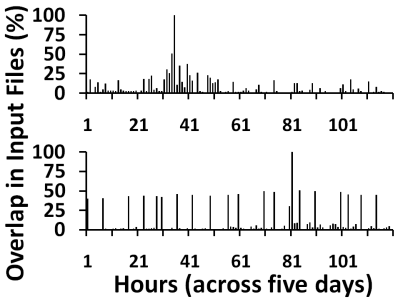


Figure 4: **Hourly overlap in the set of files accessed with two sample reference hours (*hour-35* and *hour-82*). The graph on the top shows a gradual change while the bottom graph shows periodically accessed files.**

illustrative reference hours. The figure on top shows gradual variation while the bottom figure shows that some sets of files are accessed in bursts. We conjecture that the difference is due to the types of files involved – the hour on the top likely consists of a time-sensitive set of files used by many different users or groups, so their popularity decays faster and more smoothly, while the bottom hour likely consists of a set of files used by fewer but more frequent users explaining the periodic bursts.

2.3 Effect of Popularity Skew: Hotspots

When more tasks want to run simultaneously on a machine than that machine can support, we will say a *contention event* has happened. MapReduce frameworks vary in how they deal with contention events. Some queue up tasks, others give up on locality and execute tasks elsewhere in the cluster and some others evict the less preferred tasks. Others adopt a combination approach – make tasks wait a bit before falling back and running them elsewhere. Regardless of the coping mechanism, contention events slow down the job and waste cluster resources.

Figure 5 plots a CDF of how contentions are distributed across the machines in the cluster. The figure shows that 50% of contentions are concentrated on a small fraction of machines (less than $(\frac{1}{6})^{th}$) in the cluster. Across periods of low and high cluster utilization (5AM and 12PM respectively), the pattern of hotspots is similar.

We attribute these hotspots to skew in popularity of files. Hotspots occur on machines containing replicas of files that have many concurrent accesses. Further, current placement schemes are agnostic to correlations in popularity, they do not avoid co-locating popular files, and hence increase the chance of contentions.

2.4 Summary

From analysis of production logs, we take away these lessons for the design of Scarlett:

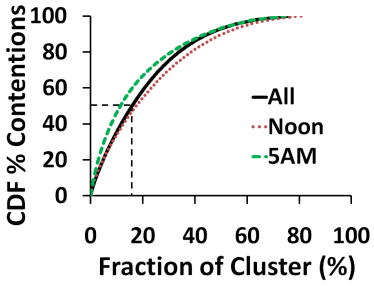


Figure 5: **Hotspots: One-sixth of the machines account for half the contentions in the cluster.**

1. The number of concurrent accesses is a sufficient metric to capture popularity of files.
2. Large files contribute to most accesses in the cluster, so reducing contention for such files improves overall performance.
3. Recent logs are a good indicator of future access patterns.
4. Hotspots in the cluster can be smoothened via appropriate placement of files.

3. Scarlett: System Design

We make the following two design choices. First, Scarlett considers replicating content at the smallest granularity at which jobs can address content. Recall that we call this a *file*. Scarlett does so because a job will access all blocks or none in a file. Even if some blocks in a file have more replicas, the block(s) with the fewest replicas become the bottleneck, i.e., tasks in the job that access these hot blocks will straggle [Ananthanarayanan 2010] and hold back the job. Second, Scarlett adopts a *proactive* replication scheme, i.e., replicates files based on predicted popularity. While we considered the reactive alternative of simply caching data when tasks executed non-locally (thereby increasing the replication factor of hot data), we discarded it because it does not work for frameworks that cope with contention by other means (e.g., by queuing or evicting low priority tasks). In addition, proactively replicating at the granularity of files is simpler to implement; it keeps improvements to the storage layer independent of the task execution logic.

Scarlett captures the popularity of files and uses that to increase the replication factor of oft-accessed files, while avoiding hotspots in the cluster and causing minimal interference to the cross-rack network traffic of jobs. To do so, Scarlett computes a replication factor r_f for each file that is proportional to its popularity (§3.1) while remaining within a budget on extra storage due to additional replicas. Scarlett smooths out placement of replicas across machines in the cluster so that the expected load on each machine (and rack) is uniform (§3.2). Finally, Scarlett uses compression and memoization to reduce the cost of creating replicas (§3.3).

```
Used Budget, B_used ← 0
F ← Set of files sorted in descending order of size
Set r_f ← 3    ∀f ∈ F                          ▷ Base Replication
for file f ∈ F do
    r_f ← max(c_f + δ, 3)                     ▷ Increase r_f to c_f + δ
    B_used ← B_used + f_size · (r_f − 3)
    break if B_used ≥ B
end for
```

Pseudocode 1: Scarlett **computes the file replication factor r_f based on their popularity and budget B. c_f is the observed number of concurrent accesses. Here files with larger size have a strictly higher priority of getting their desired number of replicas.**

```
Used Budget, B_used ← 0
F ← Set of files sorted in descending order of size
Set r_f ← 3    ∀f ∈ F                          ▷ Base Replication
while B_used < B do
    for file f ∈ F do
        if r_f < c_f + δ then
            r_f ← r_f + 1                     ▷ Increase r_f by 1
            B_used ← B_used + f_size
            break if B_used ≥ B
        end if
    end for
end while
```

Pseudocode 2: **Round-robin distribution of the replication budget B among the set of files F.**

Recall that current file systems [Ghemawat 2003, HDFS] divide files into blocks and uniformly replicate each block three times for reliability. Two replicas are placed on machines connected to the same rack switch, and the third is on a different rack. Placing more replicas within a rack allows tasks to stay within their desired rack. Datacenter topologies are such that there is more bandwidth within a rack than across racks [Kandula 2009]. The third replica ensures data availability despite rack-wide failures. Our analysis in §2 shows sizable room for improvement over this policy of uniform replication.

3.1 Computing File Replication Factor

Scarlett replicates data at the granularity of files. For every file, Scarlett maintains a count of the maximum number of concurrent accesses (c_f) in a *learning window* of length T_L. Once every *rearrangement period*, T_R, Scarlett computes appropriate replication factors for all the files. By default, $T_L = 24$ hours and $T_R = 12$ hours. The choice of these values is guided by observations in the production logs that show relative stability in popularity during a day. It also indicates Scarlett's preference to conservatively replicate files that have a consistent skew in popularity over long periods.

Scarlett chooses to replicate files proportional to their expected usage c_f. The intuition here is that the expected load at a machine due to each replica that it stores be constant – the load for content that is more popular is distributed across a proportional number of replicas. To provide a cushion against under-estimates, Scarlett creates δ more replicas. By default $\delta = 1$. Scarlett lower bounds the replication by three, so that data locality is at least as good as with the current file systems. Hence the desired replication factor is $\max(c_f + \delta, 3)$.

Scarlett operates within a fixed budget B on the storage used by extra replicas. We note that storage while available is not a free resource, production clusters routinely compress data before storing to lower their usage. How should this budget be apportioned among the various files?

Scarlett employs two approaches. In the *priority* approach, Scarlett traverses the files in descending order of their size and increases each file's replication factor up to the desired value of $c_f + \delta$ until it runs out of budget. The intuition here is that since files with larger size are accessed more often (see Figure 2) and also have more tasks working on them, it is better to spend the limited budget for replication on those files. Pseudocode 1 summarizes this approach. We would like to emphasize that while looking at files sorted by descending order of size is suited for our environment, the design of Scarlett allows any ordering to be plugged in.

The second *round-robin* approach alleviates the concern that most of the budget can be spent on just a few files. Hence, in this approach, Scarlett increases the replication factor of each file by at most 1 in each iteration and iterates over the files until it runs out of budget. Pseudocode 2 depicts this approach. The round-robin approach provides improvements to many more files while the priority approach focuses on just a few files but can improve their accesses by a larger amount. We evaluate both distribution approaches for different values of the budget in §5.

The following desirable properties follow from Scarlett's strategy to choose different replication factors for files:

- Files that are accessed more frequently have more replicas to smooth their load over.
- Together, δ, T_R and T_L track changes in file popularity while being robust to short-lived effects.
- Choosing appropriate values for the budget on extra storage B and the period at which replication factors change T_R can limit the impact of Scarlett on the cluster.

3.2 Smooth Placement of Replicas

We just saw which files are worthwhile to replicate but where to place these replicas? A machine that contains blocks from many popular files will become a hotspot, even though as shown above, there may be enough replicas for each block such that the per-block load is roughly uniform. Here, we show how Scarlett smooths the load across machines.

In current and future hardware SKUs, reading from the local disk is comparable to reading within the rack,

```
for file f in F do
    if r_f > r_f^desired then
        Delete Replicas                          ▷ De-replicate
        Update l_m accordingly
    end if
end for
for file f in F do
    while r_f < r_f^desired do
        for blocks b ∈ f do
            m* ← arg min(l_m) ∀ machines m
            Replicate(b) at m*
            l_m* ← l_m* + c_f / r_f              ▷ Update load
        end for
        r_f ← r_f + 1
    end while
end for
```

Pseudocode 3: **Replicating the set of files F with current replication factors r_f to the desired replication factors $r_f^{desired}$. l_m is the current expected load at each machine due to the replicas it stores.**

since top-of-rack switches have enough backplane bandwidth to support all intra-rack transfers. Reading across racks however continues to remain costly due to network oversubscription [Kandula 2009]. Hence, Scarlett spreads replicas of a block over as many racks as possible to provide many reasonable locations for placing the task.

Scarlett's placement of replicas rests on this principle: place the desired number of replicas of a block on as many distinct machines and racks as possible while ensuring that the expected load is uniform across all machines and racks.

A strawman approach to achieve these goals would begin with random circular permutations of racks and machines within each rack. It would place the first replica at the first machine on the first rack. Advancing the rack permutation would ensure that the next replica is placed on a different rack. Advancing to the next machine in this rack ensures that when this rack next gets a replica, i.e., after all racks have taken a turn, that replica will be placed on a different machine in the rack. It is easy to see that this approach smooths out the replicas across machines and racks. The trouble with this approach, however, is that even one change in the replication factor changes the entire placement leading to needless shuffling of replicas across machines. Such shuffling wastes time and cross-rack network bandwidth.

Scarlett minimizes the number of replicas shuffled when replication factors change while satisfying the objective of smooth placement in the following manner. It maintains a *load* factor for each machine, l_m. The load factor for each rack, l_r, is the sum of load factors of machines in the rack. Each replica is placed on the the rack with the least load and the machine with the least load in that rack. Placing a replica increases both these factors by the expected load due to that replica ($=\frac{c_f}{r_f}$). The intuition is to keep track of the current load via the load factor, and make the desired changes in replicas (increase or decrease) such that the lightly loaded machines and racks shoulder more load. In practice, Scarlett uses a slight fuzz factor to ensure that replicas are spread over many distinct racks and machines. This approach is motivated by the Deficit Round Robin scheme [Shreedhar 1996] that employs a similar technique to spread load across multiple queues in arbitrary proportions.

Pseudocode 3 shows how, once every T_R, after obtaining a new set of replication factors $r_f^{desired}$, Scarlett places those replicas. Files whose replication factors have to be reduced are processed first to get an updated view of the load factors of racks and machines. We defer how replicas are actually created and deleted to the next subsection. Traversing the list of files and its blocks, Scarlett places each replica on the next lightly loaded machine and rack. Replicas of the same block are spread over as many machines and racks as possible.

3.3 Creating Replicas Efficiently

Replication of files causes data movement over already oversubscribed cross-rack links [Kandula 2009]. This interferes with the performance of tasks, especially those of network-intensive phases like reduce and join. A skew in the bandwidth utilization of racks leads to tasks that read data over them lagging behind the other tasks in their phase, eventually inflating job completion times [Ananthanarayanan 2010]. While our policy of placing one replica per rack makes cross-rack data movement inevitable during replication, we aim to minimize it. The approximation algorithm in Pseudocode 3 takes a first stab by retaining the location of existing blocks. As a next step, we now reduce the interference caused due to replication traffic. In addition to replication traffic running at lower priority compared to network flows of tasks, we employ two techniques that complement each other – (a) equally spread replication traffic across all uplinks of racks, and (b) reduce the volume of replication traffic by trading network usage for computation using compression of data.

3.3.1 While Replicating, Read From Many Sources

We adopt the following simple approach to spread replication traffic equally across all the racks. Suppose the number of replicas increases from r^{old} to r^{new}. The old replicas equally distribute the load of creating new replicas among themselves. Each old replica is a source for $\lceil \frac{r_{new}-r_{old}}{r_{old}} \rceil$ new replicas. In the case of $\frac{r_{new}}{r_{old}} \leq 2$, each rack with old replicas will have only one copy of the block flowing over their uplinks at a time.

When the increase in number of replicas is greater than 2, Scarlett starts from r_{old} and increases the replication factor in steps of two, thereby doubling the number of sources in every step. This strategy ensures that no more than a logarithmic number of steps are required to complete the replication while also keeping the per-link cost for each block being replicated a constant independent of the number of new replicas being created.

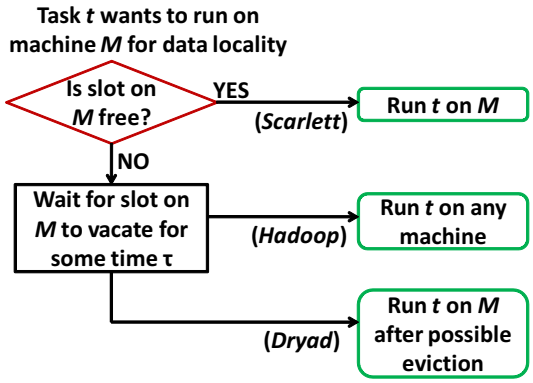


Figure 6: **Scheduling of tasks, and the different approaches to deal with conflict for slots due to data locality.** Scarlett **tries to shift the focus to the "YES" part of the decision process by preferentially replicating popular content.**

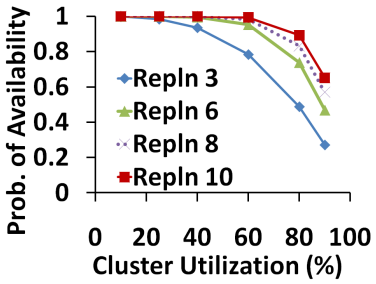


Figure 7: **The probability of finding a replica on a free machine for different values of file replication factor and cluster utilization.**

3.3.2 Compress Data Before Replicating

Recent trends in datacenter hardware and data patterns point to favorable conditions for data compression techniques. These techniques tradeoff computational overhead for network bandwidth [Skibinski 2007]. However, the trend of multiple cores on servers presents cores that can be devoted for compression/decompression purposes. Also, the primary driver of MapReduce jobs are large text data blobs (e.g., web crawls) [Hadoop Apps] which can be highly compressed.

Libraries for data compression are already used by MapReduce frameworks. Hadoop, the open-source version of MapReduce, includes two compression options [Chen 2010]. The *gzip* codec implements the DEFLATE algorithm, a combination of Huffman encoding and Lempel-Ziv 1977 (LZ77). Other popular variants of Lempel-Ziv include LZMA and LZO. Dryad supports similar schemes. Since replication of files is not in the critical path of job executions, our latency constraints are not rigid. With the goal of minimizing network traffic, we employ compression schemes with highest reduction factors albeit at the expense of computational overhead for compression and decompression. We present benchmarks of a few compression schemes (e.g., the PPMVC compression scheme [Skibinski 2004]) as well as the advantages of compressing replication data in §5.

3.3.3 Lazy Deletion

Scarlett reclaims space from deleted replicas lazily, i.e., by overwriting it when another block or replica needs to be written to disk. By doing so, the cost to delete is negligible. Deleted replicas of a block are removed from the list of available replicas.

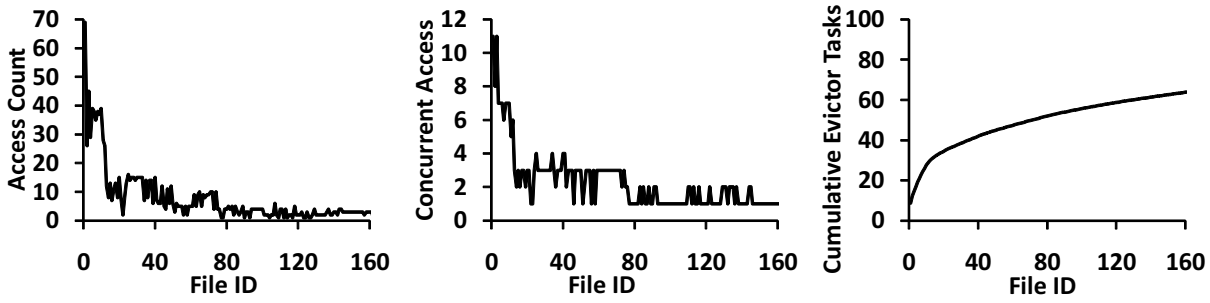
4. Case Studies of Frameworks

In this section, we describe the problems caused due to contention events in two popular MapReduce frameworks, Dryad [Isard 2007] and Hadoop [Hadoop]. The impact of hotspots on a framework depends on how it copes with contention. As a prelude to evaluating Scarlett, we analyze how hotspots impact these frameworks, quantify the magnitude of problems and the potential gains from avoiding hotspots.

There is a growing trend towards sharing of clusters for economic benefits (e.g., as mentioned in [Cloud Benefits]). This raises questions of resource allocation and MapReduce job managers support weighted distribution of resources between the different jobs [Isard 2009], reflecting the relative importance of jobs. Each job is entitled to use a *legitimate* quota of slots. However, all frameworks allow jobs to use more than their legitimate share subject to availability, called *bonus* slots. This reduces idling of resources and improves the overall throughput of the cluster.

A natural question that arises is, *how to deal with a task that cannot run at the machine(s) that it prefers to run at?* Job managers confront this question more frequently for tasks with data locality constraints. Despite the presence of free slots, locality constraints lead to higher contention for certain machines (§2.3). Many solutions to deal with contention are in use. First, less preferred tasks (e.g., those running in bonus slots) can be evicted to make way [Isard 2007; 2009]. Second, the newly arriving task can be forced to run at a suboptimal location in the cluster [Hadoop, Isard 2007, Zaharia 2010]. Third, one of the contending tasks can be paused until contention passes (e.g., wait in a queue) [Isard 2007]. In practice, frameworks use a combination of these individual approaches, such as waiting for a bounded amount of time before evicting or running elsewhere in the cluster. See Figure 6 for a summary. Each of these solutions are suited to specific environments depending on the service-level agreement constraints, duration of tasks, congestion of network links and popularity skew of input data (described in detail in §6).

Note that Scarlett provides orthogonal benefits. Scarlett minimizes the occurrence of such contentions in the first place by replicating the popular files thereby ensuring that enough machines with replicas are available.



(a) Access Counts (b) Concurrent Accesses (c) CDF of evictor tasks

Figure 8: **Correlation between file characteristics and eviction of tasks. We plot only the top 1% of the eviction-causing files for clarity. Popular files directly correlate with more evictions ((a) and (b)). The cumulative number of evictor tasks is plotted in (c). Large files also correlate with evictions – the 1% of the files in this figure account for 35% of the overall storage, and 65% of overall evictions.**

4.1 Benefits from selective replication

We present a simple analysis that demonstrates the intuition behind how increased replication reduces contention. With m machines in the cluster, k of which are available for a task to run, the probability of finding one of r replicas of a file on the available machines is $1 - (1 - \frac{k}{m})^r$. This probability increases with the replication factor r, and decreases with cluster utilization $(1 - \frac{k}{m})$.

Figure 7 plots the results of a numerical analysis to understand how this probability changes with replication factors and cluster utilizations. At a cluster utilization of 80%, with the current replication factor (r=3), we see that the probability of finding a replica among the available machines is less than half. Doubling the replication factor raises the probability to over 75%. Even at higher utilizations of 90%, a file with 10 replicas has a 60% chance of finding a replica on a free machine. By replicating files proportionally to their number of concurrent accesses, Scarlett improves the chances of finding a replica on a free machine.

As described earlier, Scarlett's replication reduces contention and provides more opportunities for map tasks to attain machine or rack locality. Storing more replicas of popular files provides more machine-local slots (§3.1) while spreading out replicas across racks and preventing concentration of popularity (§3.2) facilitates rack locality when machine locality is not achievable.

4.2 Evictions in Dryad

Dryad's [Isard 2007] scheduler pre-emptively evicts a bonus task when a legitimate task makes a request for its slot. We examine an early version of the Cosmos cluster that does so. Here we quantify the inefficiencies due to such evictions. In this version, bonus tasks were given a 30s notice period before being evicted. We refer to the legitimate and bonus tasks as *evictor* and *evicted* tasks respectively.

Likelihood of Evictions: Of all tasks that began running on the cluster, 21.1% of them end up being evicted. An overwhelming majority of the evicted tasks (98.2%) and the evictor tasks (93%) are from map phases. These tasks could have executed elsewhere were more replicas available. Reclaiming resources used by killed tasks will reduce the load on the cluster. As a second-order effect, we see from Figure 7, that the probability of finding a replica on the available machines improves with lower utilization. In addition, the spare resources can be used for other performance enhancers, such as speculative executions to combat outliers.

Correlation of evictor tasks and file popularity: Figure 8 explores the correlation between evictor tasks and the characteristics of the input files they work on – access count, concurrency and size. For clarity, we plot only the top 160 files contributing to evictions, out of a total of 16000 files (or $\sim$1%) – these account for 65% of the evictor tasks. As marked in the figure, we see that the files that contribute the most to evictor tasks are directly correlated with popularity – they have high total and concurrent numbers of accesses (Figures 8a and 8b). High concurrency directly leads to contention and eviction. A larger number of accesses implies that more tasks run overall on the machines containing these files and hence there is a greater probability of evictions. In addition, the worst sources of evictor tasks also are the bigger files. The 1% of the files plotted in Figure 8 contribute to a disproportionate 35% of the overall storage size and account for 65% of all evictions (Figure 8c). This validates our design choice in §3.1 where we order the files in descending order of size before distributing the replication budget.

Inflation of Job Durations: Figure 9 plots the improvement in completion times for the jobs in the cluster in an ideal case wherein all the evicted tasks execute to completion (i.e., do not have to be re-executed) and the evictor tasks achieve locality. The potential median and third-quartile improvements are 16.7% and 34.1% respectively. In large production clus-

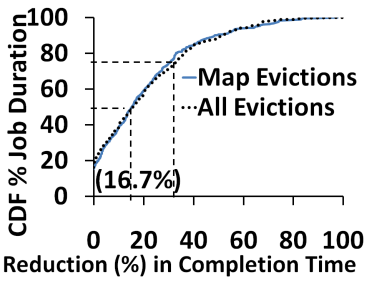


Figure 9: **Ideal improvement in job completion times if eviction of tasks did not happen.**

ters, this translates to millions of dollars in savings. Since map tasks dominate among the evictees, we see that providing this hypothetical improvement (locality without evictions) to just the map tasks is nearly the same (median of 15.2%) as when all evictions are avoided.

That evictions happen even in the presence of idle computational resources points to evictions being primarily due to contention for popular data. Our results in §5 show that Scarlett manages to reduce evictions by 83% in Dryad jobs.

4.3 Loss of Locality in Hadoop

Hadoop's policy of dealing with contention for slots is to force the new task to forfeit locality. Delay Scheduling [Zaharia 2010] improves on this default policy by making tasks wait briefly before deciding to cede locality. The data from Facebook's Hadoop logs in [Zaharia 2010] shows that small jobs (which constitute 58% of all Hadoop jobs) achieve only 5% node locality and 59% rack locality.

Hadoop does not use evictions despite a few reasons in their favor. First, eviction can be more efficient than running the new task elsewhere, for example, if the ongoing task has just started and some other machines on the cluster that the task can run at are available. Second, Quincy [Isard 2009] shows that if bonus tasks were not pre-empted, the cluster's resource allocation can be significantly far away from the desired value causing jobs to be starved and experience unpredictable lags. Our evaluation in §5 shows a 45% increase in locality for map tasks in Hadoop jobs, resulting in three quarters of the jobs speeding up by more than 44%. Half of the jobs improve by at least 20%. Note that these observed gains due to Scarlett are larger for Hadoop than for Dryad.

5. Evaluation

We first present the evaluation set-up and then proceed to the performance benefits due to Scarlett.

5.1 Methodology

We evaluate the benefits of Scarlett using an implementation and deployment of Hadoop jobs (§5.2) as well as extensive simulation of Dryad jobs (§5.3) described in Table 1. In addition, we also check the sensitivity of Scarlett's performance to the various algorithmic parameters (§5.4), budget size and distribution (§5.5), and compression techniques (§5.6).

Implementation: We implement Scarlett by extending the Hadoop Distributed File System (HDFS) [HDFS]. Our modules in HDFS log the access counters for each file and appropriately modify the replication factors of files using the adaptive learning algorithm described in §3.1. Note that this change is transparent to the job scheduler.

Hadoop Workload: Our workload of Hadoop jobs is constructed out of the traces mentioned in Table 1. We use the same inter-arrival times and input files for jobs, thereby preserving the load experienced by the cluster as well as the access patterns of files. However the file sizes are appropriately scaled down to reflect the reduced cluster size. We replace the Dryad job scripts by Hadoop programs, randomly chosen between wordcount, group by, sort and grep. We believe this approximation is reasonable as the thrust of our work is largely on the advantage of reading data locally as opposed to the specific computation. We replay a 10 hour trace from Table 1. We test our implementation on a 100-node cluster in the DETER testbed [Benzel 2006] each with 4GB memory and 2.1GHz Xeon processors.

Trace-driven Dryad Simulator: We replay the production Dryad traces shown in Table 1 with detailed simulators that mimic job operation. The simulator is extensive in that it mimics various aspects of tasks including distribution of duration and amount of input data read/written, locality of input data based on the placement of replicas, probability of failure, stragglers and recomputations [Ananthanarayanan 2010], and cluster characteristics of when computation slots open up. It also takes evictions into account by verifying if a replica can be found in the unutilized machines.

Metrics: Our primary figure of merit is the reduction in completion time of jobs where,

$$Reduction = \frac{Current - Modified}{Current}$$

We weight the jobs by their duration and use CDFs to present our metric. Weighting jobs by their duration helps differentiate the impact of Scarlett on larger jobs versus smaller jobs. Larger jobs contain more tasks and utilize more cluster cycles, therefore an improvement in a larger job would be as a result of more tasks benefiting. We also consider improvements in locality, defined as the number of tasks that are able to run on the same machines that have their input data.

We first present a summary of our results:

- Scarlett's replication speeds up median Hadoop jobs by 20.2% in our cluster, and median Dryad jobs by 12.8% (84% of ideal) in our trace-driven simulations.
- Revisiting replication factors and placement of files once in 12 hours is sufficient, thereby limiting replication overhead.

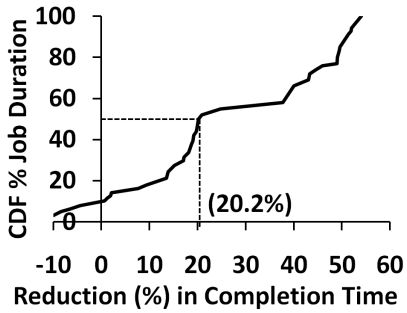


Figure 10: **Improvement in data locality for tasks leads to median and third-quartile improvements of 20.2% and 44.6% in Hadoop job completion times.**

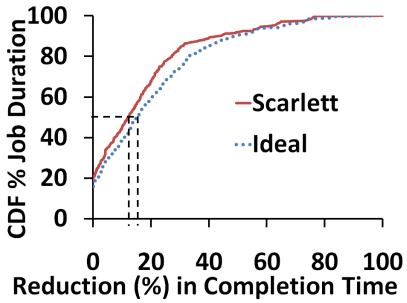


Figure 11: **Increased replication reduces eviction of tasks and achieves a median improvement of 12.8% in job completion times, or 84% of ideal.**

- Performance under a storage budget of 10% is comparable to an unconstrained replication.
- Replication increases network overhead by only 1% due to effective data compression.

5.2 Does data locality improve in Hadoop?

Hadoop's reaction to a request for a slot that is currently in use is to forfeit locality after a brief wait (See §4.3). We measure the improvement in completion times due to higher data locality for Hadoop jobs in our cluster using Scarlett over the baseline of HDFS that replicates each file thrice. We set $\delta = 1$, let T_L range from 6 to 24 hours, set storage budget $B = 10\%$ and rearrange once at the beginning of the ten hour run ($T_R \geq 10$ hours).

Figure 10 marks the reduction in completion times of 500 jobs. We see that completion times improve by 20.2% and 44.6% at the median and 75^{th} percentile respectively. This is explained by the increase in fraction of map tasks that achieve locality. The fraction of map tasks that achieve locality improves from 57% with vanilla HDFS to 83% with Scarlett, in other words a 45% improvement.

5.3 Is eviction of tasks prevented in Dryad?

As described in §4.2, replicating popular files reduces the necessity for eviction and wastage of work, in turn leading to jobs completing faster. The ideal improvement when all evictions by map tasks are avoided is 15.2% at median, and Scarlett produces a 12.8% median improvement. Here, we set $\delta = 1$, $T_R = 12$ hours, let T_L range from 6 to 24 hours, and set storage budget $B = 10\%$. Figure 11 compares the ideal case with our replication scheme where we obtain 84% of ideal performance at median. The ideal case contains no evictions and at the same time assumes that all evictor tasks achieve locality.

A closer look reveals that by replicating popular content, Scarlett avoids 83% of all evictions, i.e., the evictor tasks could be run on another machine containing a replica of their input. Note that this number goes up to 93%, when we consider evictions by tasks operating on the top hundred popular files, confirming the design choice in Scarlett to focus on the more popular files. Increasing the storage budget provides marginal (but smaller improvements) – with an increased storage budget of 20% Scarlett prevents 96% of all evictions.

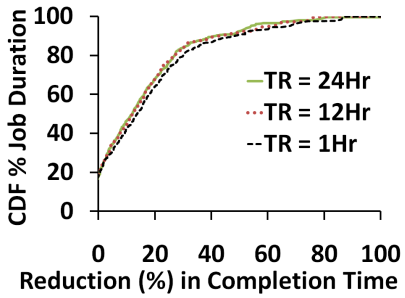
5.4 Sensitivity Analysis

We now analyze the sensitivity of Scarlett to the parameters of our learning algorithm– rearrangement window, T_R, and the cushion for replication, δ. T_R decides how often data is moved around, potentially impacting network performance of currently running jobs. δ results in greater storage occupancy and more replication traffic.

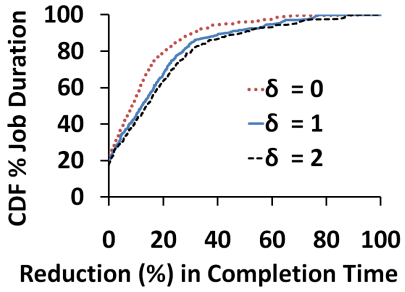
Figure 12a compares the improvement in Dryad job completion times for different rearrangement windows, i.e., $T_R = \{1, 12, 24\}$ hours. Interestingly, we see that T_R has little effect on the performance of jobs. Re-evaluating replication decisions once a day is only marginally worse than doing it once every hour. This points to Scarlett's minimal interference on running jobs. It also points to the fact that most of the gains in the observed Dryad workload accrue from replicating files that are consistently popular over long periods. Results from our Hadoop deployment are similar. For T_R values of 1, 5 and 10 hours, the median improvements are 21.1%, 20.4% and 20.2% respectively. By default, we set T_R to 12 hours, or rearrange files twice a day.

The replication allowance δ impacts performance. Changing δ from 0 to 1 improves performance substantially, but larger values of δ have lower marginal increases. Figure 12b shows that for δ values of 0, 1 and 2, the median reductions in Dryad job durations are 8.5%, 12.8% and 13.8%. Note the improvement of 42% as δ changes from 0 to 1. Likewise, our Hadoop jobs see a 56% increase from 12.9% to 20.2% in median improvement in completion time as we shift δ from 0 to 1. We believe this is because operating at the brim with a replication factor equal to the observed number of concurrent accesses is inferior to having a cushion, even if that were only one extra replica.

Figure 13 shows the cost of increasing δ. Values of storage overhead change upon replication, i.e., once every T_R=12



(a) Rearrangement Window (T_R)



(b) Replication Allowance (δ)

Figure 12: **Sensitivity Analysis of T_R and δ. Rearranging files once or twice a day is only marginally worse than doing it at the end of every hour. We set T_R as 12 hours in our system. On the other hand, δ plays a vital role in the effectiveness of** Scarlett**'s replication scheme.**

hours. $\delta = 2$ results in a 24% increase in storage, almost double of the overhead for $\delta = 1$. Combined with the fact that we see the most improvement when moving from δ from 0 to 1, we fix δ as 1.

5.5 Storage Budget for Replication

Figure 14a plots reduction in Dryad job completion times for various budget values, measured with respect to the baseline storage of three replicas per file. Here, we use the priority distribution, i.e., larger files are preferentially replicated over smaller files within the budget. A budget of 10% improves performance substantially (by 88%) over a 5% limit. As expected, the lower budget reduces storage footprint at the expense of fewer files being replicated or a smaller replication factor for some files. The marginal improvement is smaller as the budget increases to 15%. This indicates that most of the value from replication accrues quickly, i.e., at small replication factors for files. Conversations with datacenter operators confirm that 10% is a reasonable increase in storage use.

Note however that the improvement going from a budget of 2% to a budget of 5% is smaller than when going from 5% to 10%. This is likely because the distribution policy used by Scarlett is simple and greedy but not optimal. Likely, there are some files, replicating which yields significantly more benefit per unit extra storage, that Scarlett fails to replicate when

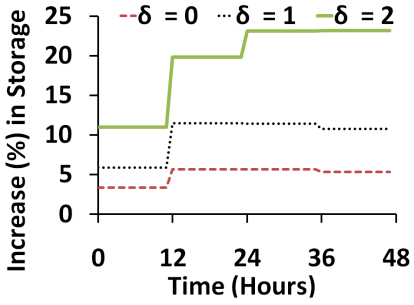


Figure 13: **Increasing the value of the replication allowance (δ) leads to** Scarlett **using more storage space. We fix δ as 1.**

budgets are small. However, these inefficiences go away with a slightly larger budget value of 10%, and we choose to persist with the simpler algorithm.

Hadoop jobs, from Figure 14b, exhibit a similar trend. The increase in median completion time when moving from a budget of 5% to 10% is much higher (120%). This indicates that how Hadoop deals with contentions (by moving tasks elsewhere) is likely more sensitive to the loss of locality when popular files are not replicated.

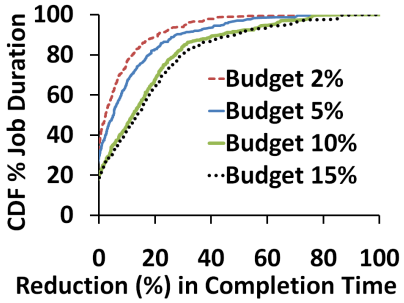
Priority vs. Round-robin Distribution: Recall from §3 that the replication budget can be spread among the files either in a priority fashion – iterate through the files in decreasing order of size, or distributed iteratively in a round-robin manner. Figure 15a plots the performance of Dryad jobs with respect to both these allocations. For a replication budget of 10%, we observe that the priority allocation gives a median improvement of 12.8% as opposed to 8.4% with round-robin allocation, or a 52% difference. This is explained by our causal analysis in Figure 2 and Figure 8 that shows that large files account for a disproportionate fraction of the evicting tasks while also experiencing high levels of concurrent accesses. Hence, giving them a greater share of the replication budget helps avoid more evictions. Hadoop jobs exhibit a greater difference of 63% between the two distributions showing greater sensitivity to loss of locality (Figure 15b).

However, the difference in advantage between the two distributions are negligible at small replication budgets. As we see in Figure 15a, the limited opportunity to replicate results in there being very little to choose between the two distribution strategies.

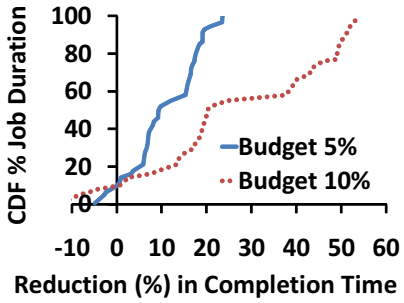
5.6 Increase in Network Traffic

For $\delta = 1$, the maximum increase in uncompressed network traffic during rearrangement of replicas is 24%. Using the PPMVC compression scheme [Skibinski 2004], this reduces to an acceptable overhead of 0.9%.

We also present micro-benchmarks of various compression techniques. Table 2 lists the compression and de-compression speeds as well as the compression ratios achieved by a few compression algorithms [Skibinski 2004;

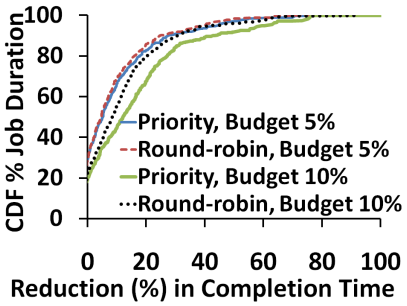


(a) Dryad

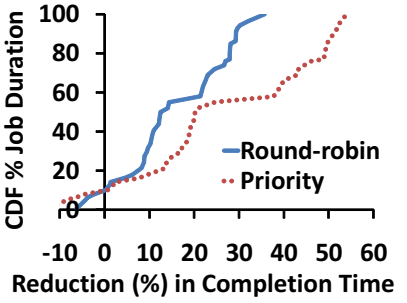


(b) Hadoop

Figure 14: **Low budgets lead to little fruitful replication. On the other hand, as the graph below shows, budgets cease to matter beyond a limit.**



(a) Dryad



(b) Hadoop

Figure 15: **Priority distribution of the replication budget among the files improves the median completion time more than round-robin distribution.**

Scheme	Throughput (Mbps)		Compression Factor
	Compress	De-compress	
gzip	144	413	12-13X
bzip2	9.7	88.2	19-20X
LZMA	3.6	375	22-23X
PPMVC	30.2	31.4	26-27X

Table 2: **Comparison of the computational overhead and compression factors of compression schemes.**

2007]. There is a clear trend of more computational overhead providing heavier compression. Given the flexible latency constraints for replication and low bandwidth across racks, Scarlett leans toward the choice that results in the least load on the network.

6. Related Work

The principle of "replication and placement" of popular data has been employed in different contexts in prior work. Our contributions are to (i) identify (and quantify) the content popularity skew in the MapReduce scenario using production traces, (ii) show how the skew causes contention in two kinds of MapReduce systems (ceding locality for Hadoop vs. eviction for Dryad), and (iii) design solutions that operate under a storage budget for large data volumes common in MapReduce systems. While we have evaluated Scarlett primarily for map tasks, we believe the principle of proactively replicating content based on its expected concurrent access can be extended to intermediate data too (e.g., as in Nectar [Gunda 2010] that stores intermediate data across jobs).

Much recent work focuses on the tussle between data locality and fairness in MapReduce frameworks. Complementary to Scarlett, Quincy [Isard 2009] arbitrates between multiple jobs. Delay Scheduling [Zaharia 2010] on the other hand supports temporary relaxation of fairness while tasks wait to attain locality. This can alleviate contention by steering tasks away from a hotspot. It however makes some assumptions that do not hold universally: (a) task durations are short and bimodal, and (b) one task queue per cluster (as in Hadoop). In the Cosmos clusters at Microsoft, tasks are longer (median of 145s as opposed to the 19s in [Zaharia 2010]) to amortize overheads in maintaining task-level state at the job manager, copying task binaries etc. Task lengths are also more variable in Dryad owing to diversity in types of phases. Finally, Dryad uses one task-queue per machine, further reducing the load at the job scheduler to improve scalability. Scarlett does not rely on these assumptions and addresses the root cause of contention by identifying and replicating popular content. Furthermore, Scarlett can be beneficially combined with both Delay Scheduling and Quincy.

The idea of replicating content in accordance to popularity for alleviating hotspots has been used in the past. Caching

popular data and placing it closer to the application is used in various content distribution networks (CDNs) [Akamai, Coral CDN] in the Internet. Beehive [Ramasubramanian 2004] proactively replicates popular data in a DHT to provide constant time look-ups in peer-to-peer overlays. Finally, dynamic placement of popular data has also been recently explored in the context of energy efficiency [Verma 2010]. To the best of our knowledge, ours is the first work to understand popularity skew and explore the benefits of dynamic data replication in MapReduce clusters. The context of our work is different as file access patterns and sizes in MapReduce significantly differ from web access patterns. It differs from Beehive due to the different application semantics. While Beehive is optimized for lookups, Scarlett aims at parallel computation frameworks like MapReduce. Further, our main goal is to increase performance rather than be energy efficient, so we aim for spreading data across nodes as opposed to compaction.

Bursts in data center workloads often result in peak I/O request rates that are over an order of magnitude higher than average load [Narayanan 2008]. A common approach to deal with such bursts is to identify overloaded nodes and offload some of their work to less utilized nodes [Appavoo 2010, Narayanan 2008]. In contrast, our approach is geared towards a read-heavy workload (unlike [Narayanan 2008]), common to MapReduce clusters. While Dynamo [Appavoo 2010] reactively migrates (not replicate) popular data, we replicate and do so proactively, techniques more suited to our setting. Recent work [Belaramani 2009, Stribling 2009] on providing semantic context to the file system can be leveraged to implement our replication policies.

A wide variety of work has also been done in the area of predictive pre-fetching of popular files based on historical access patterns [Curewitz 1993] as well as elaborate program and user based file prediction models [Yeh 2002]. However, these are in the context of individual systems and deal with small amounts of data unlike our setting with petabytes of distributed storage, the replication and transfer of which require strict storage/network constraints.

Some prior work on dynamic database replication policies [Soundararajan 2006] is very similar in flavor to ours. However, these policies are reactive in reference to application latency requirements. Our work, on the other hand, focuses on designing proactive replication policies.

Finally, much recent work has gone into designs for full bisection bandwidth networks. By suitably increasing the numbers of switches and links in the core, these designs ensure that the network will not be the bottleneck for well-behaved traffic [Al-Fares 2008, Greenberg 2009]. Well-behaved refers to the hose model constraint, which requires the traffic incoming to each machine to be no larger than the capacity on its incoming network link. We note that Scarlett's benefits remain even if networks have full bisection bandwidth, since concurrent access of blocks results in a bottleneck at the source machine that stores them. By providing more replicas (as many as the predicted concurrent access), Scarlett alleviates this bottleneck.

7. Conclusion

Analyzing production logs from Microsoft Bing's datacenters revealed a skew in popularity of files, making the current policy of uniform data replication sub-optimal. Machines containing popular data became bottlenecks, hampering the efficiency of MapReduce jobs. We proposed Scarlett, a system that replicates files according to their access patterns, ageing them with time. Using both a real deployment and extensive simulations, we demonstrated that Scarlett's replication improved data locality in two popular MapReduce frameworks (Dryad and Hadoop) and sped up jobs by 20.2%. Scarlett's guided replication used limited extra storage (less than 10%) and network resources (1%).

Acknowledgments

We would like to thank the Microsoft Bing group for access to Cosmos traces and informative discussions that guided this work. Bikas Saha and Ramesh Shankar contributed domain knowledge and critiques that improved this work. We are also grateful to our shepherd, Robbert van Renesse and the anonymous reviewers whose input greatly improved the paper. For feedback on the draft, we acknowledge Ali Ghodsi, Lucian Popa, Matei Zaharia and David Zats of the RAD Lab. We are also thankful to the DETER team for supporting our experiments.

References

[Akamai] Akamai. Akamai content distribution network. http://www.akamai.com/.

[Al-Fares 2008] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM'08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008.

[Ananthanarayanan 2010] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Appavoo 2010] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg. Providing a Cloud Network Infrastructure on a Supercomputer. In *HPDC'10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[Belaramani 2009] N. M. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for Distributed Storage Systems. In *NSDI'09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[Benzel 2006] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Experience with DETER: A Testbed for Security Research. In *International Conference*

on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006.

[Chaiken 2008] Ronnie Chaiken, Bob Jenkins, Perke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB'08: Proceedings of the 34th Conference on Very Large Data Bases*, 2008.

[Chen 2010] Y. Chen, A. Ganapathi, and R. Katz. To Compress or Not To Compress - Compute vs. IO tradeoffs for MapReduce Energy Efficiency. In *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*, 2010.

[Cloud Benefits] Cloud Benefits. Cloud compute can save govt agencies 25-50% in costs, 2010. http://googlepublicpolicy.blogspot.com/2010/04/brookings-cloud-computin%g-can-save-govt.html.

[Coral CDN] Coral CDN. The coral content distribution network. http://www.coralcdn.org/.

[Curewitz 1993] K. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *SIGMOD'93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[Dean 2009] J. Dean. Designs, lessons and advice from building large distributed systems, 2009. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis20%09.pdf.

[Dean 2004] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[Ghemawat 2003] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *SOSP'09: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[Greenberg 2009] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM'09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

[Gunda 2010] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Data Centers. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

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

[Hadoop Apps] Hadoop Apps. Applications and organizations using hadoop, 2010. http://wiki.apache.org/hadoop/PoweredBy.

[HDFS] HDFS. Hadoop distributed file system. http://hadoop.apache.org/hdfs.

[Isard 2007] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys'07: Proceedings of the European Conference on Computer Systems*, 2007.

[Isard 2009] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP'09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[Kandula 2009] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. Nature of Datacenter Traffic: Measurements and Analysis. In *IMC'09: Proceedings of the Ninth Internet Measurement Conference*, 2009.

[Narayanan 2008] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[Ramasubramanian 2004] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-law Query Distributions in peer-to-peer Overlays. In *NSDI'04: Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[Shreedhar 1996] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round-Robin. In *IEEE/ACM Transactions on Networking*, 1996.

[Skibinski 2004] P. Skibinski and S. Grabowski. Variable-length contexts for PPM. In *DCC'04: Proceedings of IEEE Data Compression Conference*, 2004.

[Skibinski 2007] P. Skibinski and J. Swacha. Fast and Efficient Log File Compression. In *CEUR ADBIS'07: Advances in Databases and Information Systems*, 2007.

[Soundararajan 2006] G. Soundararajan, C. Amza, and A. Goel. Database Replication Policies for Dynamic Content Applications. In *EuroSys'06: Proceedings of the European Conference on Computer Systems*, 2006.

[Stribling 2009] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, F. Kaashoek, and R. Morris. Flexible, Wide-Area Storage for Distributed Systems with WheelFS. In *NSDI'09: 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[Verma 2010] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *FAST'10: Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.

[Yeh 2002] T. Yeh, D. E. Long, and S. A. Brandt. Increasing Predictive Accuracy by Prefetching Multiple Program and User Specific Files. *HPCS'02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002.

[Zaharia 2010] M. Zaharia, D. Borthakur, J. Sen Sharma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys'10: Proceedings of the European Conference on Computer Systems*, 2010.