Balanced and Predictable Networked Storage

Jaimie Kelley and Christopher Stewart
The Ohio State University

***Abstract*—Networking bandwidth and latency have improved in recent years, prompting a wide range of workloads to move back to key value stores, databases, and other types of networked storage. However, networked storage has a well known drawback: Outlier access times create a heavy tailed distribution. Outlier accesses can take much longer than normal access times. This paper studies the effects of outliers on data processing workloads. These workloads strive for balance, i.e., all nodes are kept busy at all times. Outlier accesses can cause bubbles in the pipeline, slowing down the whole workload. For this paper, we modeled the effect of outliers in balanced map reduce systems. We found that outliers can cause 70% slowdown. We also modeled a solution: Use 5% of system resources on replication for predictability—an old but seldom used approach to mask outliers. We found that this approach could return more than 5% in speedup.**

I. Introduction

Big data is often too complex for mere mortals. Graph processing [5], [8], [13], NLP [12], and data mining tools try to reduce big data to smaller but still useful nuggets. These workloads pull in large amounts of data, process it, and then return a smaller result. Pulling in the data is often the slowest part [15]. Loading 1GB from today's disks takes almost as long as it did 4 years ago. 10Gb Ethernet exceeds disk bandwidth by more than 10X, making it faster to access data stored in a remote node's main memory than to access it from local disk. As a result, network storage is used more often for big data workloads.

Big data workloads strive for balance, i.e., all nodes should be busy at all times. Well-balanced workloads achieve high throughput without wasting resources. For workloads that use networked storage, balance means there should always be a few backlogged accesses, but the backlog should not idle nodes in the data processing layer. One approach to achieving balance is to 1) measure typical storage access times, 2) measure the average access rate of each node that does data processing, and 3) size the data processing cluster according to the quotient of these numbers. In practice, this approach falls short, because access times in networked storage often have heavy tails. A few outlier accesses take much longer (100X) than typical accesses. These outliers cause delays in the data processing layer, delays that can not be recovered easily.

For this paper, we studied slowdown caused by slow storage accesses in balanced map reduce systems. First, we compared access times from a real key value store against exponential and Pareto Distributions. The Pareto was a better fit because of its heavy tail. Then, we modeled an access's *slack*, i.e., the smallest response time that would cause a delay in data processing. Finally, we used the Pareto to compute the expected delay caused by accesses that exceed their slack time.

Our model showed that outliers can slow down balanced map reduce by 70% when map tasks complete quickly (i.e., within 40ms). Slowdown decreases for workloads with longer map times and lighter tails. Storage capacity per map node also affects slowdown. Maps that need random access to big data spread across many nodes are vulnerable to slowdown. We concluded that these properties, short map times and random access to big data, often describe workloads that reduce big data, e.g., graph processing and stream sampling.

We extended our model to study replication for predictability, an old but seldom used approach to reduce the effects of outliers. We found that replication for predictability was most effective for short map jobs with large working sets, the conditions where outliers caused large slowdown. When maps complete quickly, replication for predictability prevented 12.5% of lost throughput while using only 5% of storage resources.

The remainder of this paper is as follows: Section II discusses the trends and motivations in data processing that underlie this work. Section III presents our problem statement. Section IV walks through our model that captures slowdown caused by outliers. Section V extends our model to consider replication for predictability and studies its cost effectiveness. Section VI discusses related work.

II. Trends

Hadoop [3], Ceil [11], and Dryad [6] share a common trait: data pipelining. These data processing platforms try to keep disks, CPUs, and network links busy at all times. For example, Hadoop works best when data from a node's local disks is pulled in asynchronously while map and reduce tasks run concurrently. However, a node's local disks no longer offer the best performance [15], [16]. Today's disks support 800Mb/s whereas today's local area networks can support 10Gb/s. Networks will become even faster in the future as 40Gb/s Ethernet and hybrid electrical/optical switches are adopted.

When raw processing speed is the metric of merit, a 1 TB disk should be replaced with 16 64GB in-memory networked stores. On 10Gb Ethernet, the latter can achieve more than 300X speedup. Even on 1Gb Ethernet, a well-managed in-memory networked store can offer 20X speedup. The downsides for in-memory approaches are cost and power usage. Both increase quickly as data sets scale. When cost is also a concern, each node should support multiple disks with data striped across them. 16 disks accessed in parallel fall just below the throughput of 10Gb Ethernet [16]. However, 16 disks may not be enough in a few years. Further, the benefit

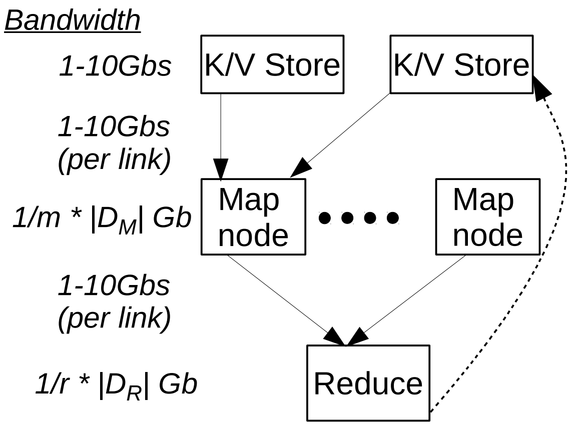

Fig. 1: Data processing backed by networked storage under the map reduce model. Processing rate (bandwidth) at each stage is shown on the left.

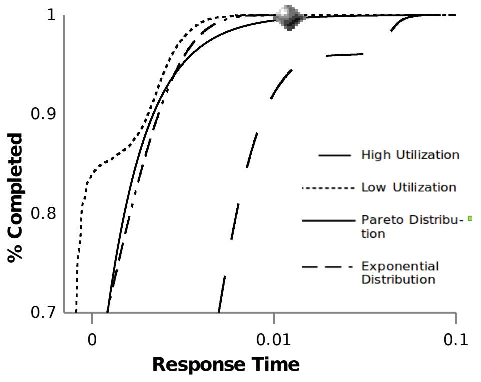

Fig. 2: A cumulative distribution function regarding the times to access a Redis store under high and low utilization, shown with a Pareto and an exponential distribution based on the low utilization numbers. The 99.99th percentile of the exponential distribution's heavy tail is marked.

of fast, random data access will make networked storage attractive.

Figure 1 depicts data flow and bandwidth when map reduce uses networked storage instead of node-local disks. In this paper, we will assume networked storage is in the form of in-memory key value stores, e.g., MemCached [1], but our ideas extend broadly to other types of stores. For a balanced system, the networked store should fully use bandwidth offered by its network card, either 1 or 10GbE. Maps may use data spread across multiple stores for three reasons. First, other unrelated jobs may lower the bandwidth available on a networked store [17]. Second, maps that access many small keys can encounter bottlenecks in TCP, operating system, and network congestion. Finally, networked stores that access disk have about 1/16th the bandwidth as 10GbE networks. Partitioning allows map jobs to regain lost bandwidth.

As shown in Figure 1, the map phase is often the slowest. Because of this, the map reduce model parallelizes this phase as much as possible. Let m be the average map time and $|D_M|$ be the average working set per map. If $|D_M|$ falls below 0.1Gb on a 1Gb/E network, then m must fall below 0.1 seconds to avoid slowing down the system. On the other hand, a map task that completes in constant time would require parallel data access as the data sizes grow. Reduce times are usually smaller than map times. They do not bound map reduce overall system times, and thus are not the focus of this paper.

A. Outliers in Networked Storage

Networked storage is a more complicated storage fabric than local disks. Networked stores may include processors, DRAM, SSDs, and rotating disks. Operating systems and middleware connect these hardware devices. A mishap by any of these components can slow down access times by a significant amount. Networked stores are known to have outlier access times that are much slower than normal access times.

The root causes of outliers vary. For a concrete example, consider write buffering in a key value store. To keep fast response times, most stores keep a relatively large in-memory write buffer. The buffer is flushed to disk periodically (every few seconds) to ensure a degree of fault tolerance to power loss. Writes that hit in the buffer can proceed at the speed of main memory, completing within a few hundred microseconds. However, writes that are stuck behind a buffer flush may be delayed by several hundred milliseconds.

Figure 2 shows the access times for a Redis [2] store deployed on a 2GHz core with 2GB main memory. The workload shown represents 100% reads, and is tested under mean CPU utilization of 65.45% (high) and utilization of 10.75%(low). Under low utilization, we observe a heavy tail beginning with the 99th percentile, but when the Redis store is heavily utilized, we observe a heavy tail earlier, beginning with the 95th percentile. Most importantly, we note the respective lengths of these heavy tails. The exponential and Pareto distributions plotted here use the low utilization access times as a basis. The low utilization, high utilization, and Pareto heavy tails are much longer than the exponential heavy tail. The results are similar in production systems. Google BigTable reports default access times where the 99.9th percentile is 31X the mean [4]. Other works have noted similar results with MemCached [7].

B. Workloads that Reduce Big Data

Workloads that reduce big data to smaller chunks can use fast networked storage well. These workloads access a lot of data per map and they complete map tasks quickly. Graph analysis and data mining are well-known examples of such data reduction–the above shows a read-only workload on Redis primarily because graph processing works mostly with read-only data [14]. Consider the problem of finding 2-hop friends in a social network. One approach pulls in data from a large subgraph of the network and then looks up all unique 2 hops within the subgraph from the origin friend. The subgraph itself can easily exceed $10MB$, yet looking up $16K$ hops during each map task can complete within milliseconds. Many data mining problems have similar properties due to statistical sampling.

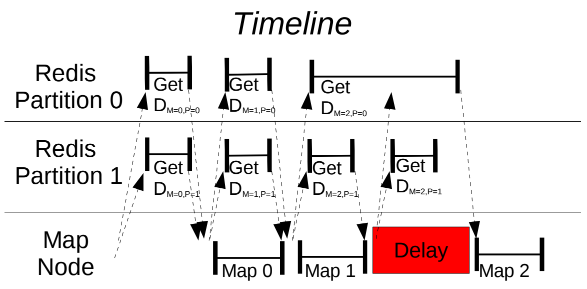

Fig. 3: Slowdown caused by an outlier access to networked storage. Dotted lines are messages over the network. Solid lines reflect processing. For simplicity, we show all accesses for a single map stemming from a single network message.

	Controlled Model Inputs
C	Storage capacity per map node
m	Average map time
α	Pareto coefficient of the networked store
f	Reserved (unused) capacity on the networked store
	Derived Model Parameters
μ	Mean service time for the networked store
$\tilde{x}$	Median service time for the networked store
a	Average accesses per map
s_n	Slack time produced by n accesses
$\phi(t)$	Probability of an access longer than t

TABLE I: Model Inputs.

Widely used tools for data processing, like Hadoop, target data transforms—not data reduction. Map tasks for transforming data take longer since every bit is touched. The Hadoop manual [3] calls for map jobs that take hours (meaning $|D_M|$ would need to exceed 36TB/s to balance network speeds). Workloads like Terasort provide such semantics. Emerging platforms for graph processing are more inline with big-data reduction [5], [8], [13].

III. Problem Statement

Figure 3 depicts a delay caused by an outlier access to networked storage in a data processing workload. Data accesses to Redis are pipelined, keeping all nodes busy in the ideal case. In the common case, the map node receives data just before it is needed. However, the last access on partition 0 is orders of magnitude slower than usual, preventing the next map from beginning. Such delays reflect lost throughput. Even if subsequent data accesses complete more quickly than usual, the pipelined nature of map tasks would not speed up.

This paper explores two questions:

1. *How much do outliers slow down data processing?*
2. *Can we effectively mitigate outliers with redundancy?*

IV. Modelling Outliers

We used Operational Laws to model resource needs for networked stores and map nodes in a balanced system. We converted resource needs into expected delay. Finally, we used stochastic analysis to capture delays caused by outliers. This section describes our efforts. Table I describes the model parameters used in this section. We set controlled parameters directly. We restricted storage capacity per map node (C) to positive integers, meaning each map node pulled data from 1 or more dedicated storage partitions. A map node would pull from more than 1 storage partition in parallel if it needed access to a large working set. We varied map times (m) from 20ms (small) to 5s (large). We set the Pareto coefficient to control the heaviness of access time tails from the networked store: lightly heavy tail (1.76), normal heavy tail (1.44), and heavy heavy tail (1.13). In a nod to real system managers, we allow for some reserved, unused capacity, f, which represents both time that could have been used to access data from networked storage but is not, and also a percentage of the networked storage data nodes that is unused. We set this parameter to 5% universally for networked stores.

We also made the following assumptions about our target systems:

- The networked store supports gets and puts on keys and values.
- The size of keys and values are fixed. In our tests, we use 1KB blocks.
- Maps know which data to request in advance before they execute.

We believe these assumptions can be relaxed in future work without changing our conclusions.

We used the control parameters and our assumptions to derive other parameters. First, we computed the average and median access times in a Pareto distribution given the Pareto coefficient (α).

$$\tilde{x} = X_{min} * \alpha^{0.5}$$

$$\mu = \frac{\alpha * X_{min}}{\alpha - 1}$$

Here, X_{min} is the smallest observed access time. We set this to 600µs based on data from Figure 2.

We used the Utilization Law to get the average accesses to storage per map. In a balanced system, the quotient of map time divided by average access time is multiplied by the capacity that is actually used; some storage capacity is kept in reserve. Accesses per storage partition per map simply divides this number by the storage capacity.

$$a = \frac{m}{\mu} * (1 - f)$$

$$a_i = \frac{a}{C}$$

Next, we computed *slack time*, the minimum delay for 1 outlier that could delay a map task. Slack time depends on the number of storage accesses that follow an outlier. An outlier followed by many accesses can be masked if subsequent accesses complete quickly. An outlier followed by only a few accesses is more likely to cause a delay. In our approach, slack time is comprised of two components. First, we turned the unused, reserved capacity (f) into idle time by multiplying this by the average map time. Then, we added the difference of the mean and the median, multiplied by n. This means that an outlier that occurs when there are n outstanding accesses to the networked store can be masked if the remaining accesses complete according the median. For simplicity, our model makes the quantity of final storage accesses proportional to the over-provisioning range.

$$n = a * f$$

$$s_n = m * f + \frac{n}{C} * \mu - \frac{n}{C} * \tilde{x}$$

Given n, we can compute the probability and expected delay of an outlier that exceeds slack time. If networked stores had exponentially distributed access times, the expected delay would be fixed. However, heavy tail Pareto distributions are more complex. The first equation below computes the cumulative distribution function given α, X_{min}, and s_n. The equation after that computes the probability that 1 of n accesses is greater than s_n, i.e., the probability of a delayed map.

$$\phi(s_n) = 1 - \frac{X_{min}}{s_n}^{\alpha}$$

$$Pr(x > s_n) = 1 - \phi(s_n)^n$$

$$E(x|x > s_n) = \frac{2^{\frac{1}{\alpha}} X_{min}}{[1 - \phi(s_n)]^{\frac{1}{\alpha}}}$$

The final equation shows the typical (median) access time for such an outlier. The median delay of an outlier is the middle percentile starting from $\phi(s_n)$. A quick check reveals that when $\phi(s_n) = 0$, the result is the equation for the global median in a Pareto distribution.

Model Results: For Figure 4, we fixed storage capacity per node ($C = 4$), the Pareto coefficient, and unused capacity ($f = 5\%$). We controlled average map time and studied its effect on the slowdown caused by outliers. We show the equation for slowdown below:

$$slowdown = \frac{m + Pr(x > s_n)E(x|x > s_n)}{m}$$

We found that large map times (>5s) have first-order effects on slowdown. Large map times hide outliers in two ways. First, m is the only parameter in the denominator in our slowdown formula above. An outlier that causes the same absolute delay leads to less slowdown under large map times. Also, large map times can afford more slack time. Hadoop workloads often have large map times. In fact, the Hadoop manual calls for workloads with large (many minutes) map times [3]. Such workloads might disregard the impact of outliers when they move to networked storage.

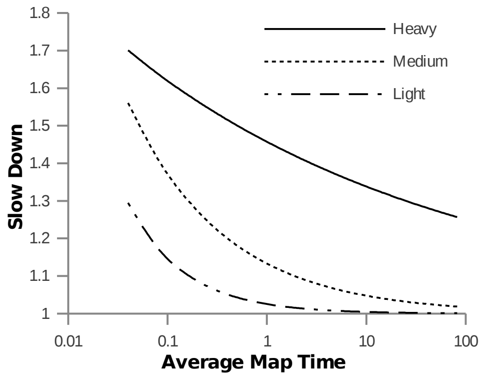

Fig. 4: Slowdown caused by outliers as average map time varies.

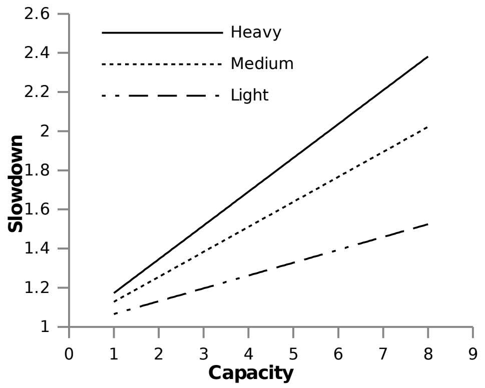

Fig. 5: Slowdown caused by outliers as storage capacity per map node varies.

On the other hand, data reduction workloads, e.g., graph processing, often have small map times. For example, a map may de-reference a few links in a large graph. Workloads with small map times suffer under heavy tail outliers. Our model expects that maps that take less than 100ms will be delayed (on average) by 5-15%. The heaviness of the tail also matters. Our heavy heavy tail setting caused up to 12X and 30X more slowdown than the normal heavy tail and light heavy tail.

Our model showed that slowdown is proportional to storage capacity per map node. Heavy tails cause outliers at a higher rate, but the effect remains linear. Even though the effects are only linear, reasonable ranges for I/O capacity lead to the largest slowdown. When each map node must contact 8 partitions in parallel, our model expects minimum slowdown around 40%. Even with just 4 nodes per partition, the worst case slowdown can exceed 63%.

V. Replication for Predictability

The model used in the previous section quantified the delay caused by outliers across map times and storage capacity. Outliers cause large delays for balanced systems with fast map times. Also, outliers cause large delays when the working set for maps is large. This section studies the potential for using replication for predictability as a solution.

Replication for predictability is an old but seldom used technique to mask outliers that manifest independently. The basic idea is simple. Instead of sending storage accesses to only one node, send parallel requests to multiple nodes and use the result from the first node to respond. Intuitively, it is unlikely that all *duplicates* will return a slow result.

Replication for predictability has been rarely used in practice. Even though it reduces the effect of outliers, it does not improve throughput. Replication for predictability also does not reduce the effects of correlated outliers. For example, accesses to rarely viewed content will be slowed by cache misses in both redundant nodes. Thus, the key question for replication for predictability is, *can it be cost effective?*

To assess whether an idea is cost effective, we must model the cost and the return. We call the ratio of these terms the yield. In this paper, we study a simple way to use replication for predictability sparingly. We use idle (unused) capacity on the networked store, i.e., f in Table I. To be concrete, the cost is 5% of networked storage resources. For that investment, we hope to make the system more predictable and to recover throughput lost to outlier effects. We measured yield as the return in slowdown divided by the investment. The full equation for yield is shown below.

$$yield = \frac{slowdown_{default} - slowdown_{rp}}{f}$$

Our model of replication for predictability assumed that storage accesses would be sent to only two duplicates. In ongoing work, we have extended the model to scale [18]. To capture the effects of replication for predictability, we have changed two aspects of the model presented in Section IV. First, accesses per storage node (a) ran at full capacity. Note, operating at full capacity increases the waiting time for accesses to networked storage. For interactive services, slow response times are costly. For the high throughput data processing workloads that we target, slow response times are only costly if they lead to delayed map jobs. In other words, our concern is the effect of queuing on slack time (s_n), where full capacity removes the buffer idle time. Updated equations are shown below.

$$a = \frac{m}{\mu}$$

$$s_n = \frac{n}{C} * \mu - \frac{n}{C} * \tilde{x}$$

On the positive side, replication for predictability reduces the chance of an outlier. If we assume that outliers arise independently, then the benefit of replication for predictability is shown below.

$$Pr(x > s_n) = 1 - \phi(s_n)^{2n}$$

Model Results For Figure 6, we again computed our model with all control parameters fixed except for the average map time. This plot shows the effect of map time on yield. We observed an effect that is comparable to the slowdown curve, but less dramatic. Map times below 100ms only reach yields ranging from 0.9–1.3. This result indicates that small map time alone do not warrant replication for predictability as this paper proposes. For small map times, outliers beyond the last n may

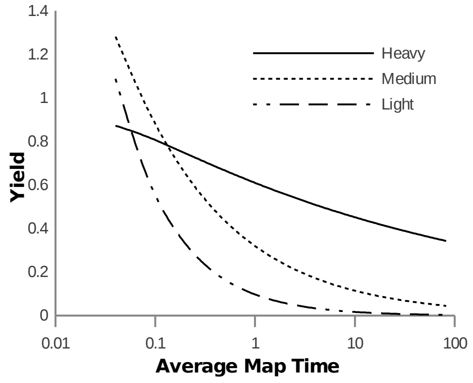

Fig. 6: Yield caused by replication for predictability increases as average map time decreases.

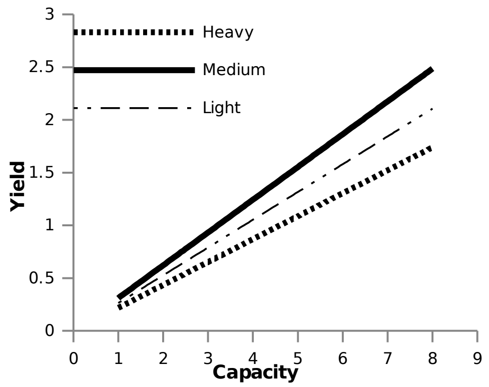

Fig. 7: Yield caused by replication for predictability increases as storage capacity per map node increases.

cause delays. Our sparing use of replication for predictability does not mask such outliers.

Looking deeper into Figure 6, the effect of outliers outside of the last n are most evident in the heavy heavy tail setting. At first, we expected this setting to provide the highest yield. However, looking further into the results showed that most of the delay under a very heavy tailed access time distribution was caused by accesses outside of the last n.

Figure 7 shows that high storage capacity per map node is sufficient reason to warrant replication for predictability. After capacity per node exceeds 5, we observed only high yields (>1).

Discussion: Our results show that the workloads that can best take advantage of replication for predictability are those with short map times and a large working set of data which is distributed in networked storage over multiple nodes. Graph processing and stream sampling workloads, which have these properties, would be ideal settings for replication for predictability to make a real difference.

Our simulations here provide a proof of construct, but in future work we intend to analyze the results of replication for predictability used on real data reduction systems.

We have focused on the cost of replication for predictability in terms of storage access rate. The approach assumes that all possible network bandwidth is used within the constraints of the data that is needed by map jobs. But since our model studied a limited use of replication for predictability, we did not consider the cost of this network bandwidth, or whether spare capacity would need to be available. We have also not looked into variances in hardware configurations here. If replication for predictability is expanded to use more resources, a topology aware approach may be needed [9].

Our model predicts yield but does not judge its value. Our intuition suggests that yield above 1 is a good investment, but ultimately, any novel scale out technique must be compared to other alternatives. If 5% spare capacity can be used in another way that provides higher yield, then replication for predictability should not be used.

Finally, we assume that the data processing platform comprises mostly reads. If writes were more frequent, we would need to consider consistency challenges posed by replication for predictability.

VI. Related Work

Networked storage is a (re)emerging trend in high-throughput systems. However, networked storage is inherently more complicated than other storage mediums, e.g., disk. This paper studies one product of such complexity: Heavy tailed access times. We make the case for a research agenda that studies this phenomena. Prior research has 1) sped up networked stores or 2) improved overall throughput for data processing.

Speeding Up Networked Stores: MemCached and Redis are widely used open source networked stores [1], [2]. They both achieve high throughput (80K–100K requests per core). Other stores proposed by researchers have achieved high throughput also [7], [10], [17]. A common approach across these stores is to avoid touching disk, keeping operations within main memory. While key value stores are most widely used, in-memory database systems have also gained traction. These databases relax their support for distributed transactions and also stay within main memory. When data sizes approach the capacity of main memory, it is better to compress data than to go to a single local disk [10]. Along with high throughput, stores can lower their latency by streamlining their execution path. [7] used soft direct memory access to remove the operating system for the data path for MemCached. These approaches make networked storage faster in the common case, however outliers (due to garbage collection, snapshots, etc.) still persist.

Balanced Data Processing: Disk is the primary component that has fallen behind. Recent work improves disk bandwidth by using multiple disks at each machine and modifying the data processing platform to access disk as little as possible [15], [16]. While these works have targeted data processing with node-local storage, they apply to networked storage with high bi-sectional bandwidth as well. Further, making the system more complex by adding multiple disks behind the networked store exacerbates outliers.

VII. Conclusion

Bandwidth and latency for datacenter networks has grown much faster than for disks. Emerging 40Gb/E and hybrid electrical and optical switches suggest that this trend will continue. Because it is and for the foreseeable future will be a faster solution than local disk, networked in-memory storage is likely to underlie many data processing platforms in the future. However, networked storage suffers from the well-known, widespread problem of access times with heavy tail distributions. This paper quantifies the effect of outliers on processes that rely on the map reduce model. These heavy tail outliers slow down the data processing pipeline at the mapping stage, and when they happen close enough to the beginning of a map, they cannot be easily masked. We saw that workloads with short map times and large data sets were most affected, with delays up to 70%. We created a model predicting the affect of these outliers in order to assess one possible solution to heavy tails: replication for predictability. This approach masks outliers by redundantly sending accesses to multiple nodes containing the same data and taking the response from the first. Our model used only 5% of storage capacity for replication for predictability, yet we often reduced slowdown by more than 7% using this approach.

References

[1] Memcached: A distributed memory object caching system. www.memcached.org.

[2] Redis.

[3] Welcome to apache hadoop. hadoop.apache.org.

[4] J. Dean. Achieving rapid response times in large online services, 2012.

[5] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symp. on Operating Systems Design and Implementation*, 2012.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems*, 2007.

[7] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. 2012.

[8] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *USENIX Symp. on Operating Systems Design and Implementation*, 2012.

[9] G. Lee, N. Tolia, P. Ranganathan, and R. Katz. Topology-aware resource allocation for data-intensive workloads. In *APSys*, 2010.

[10] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *USENIX File and Storage Technologies*, Cascais, Portugal, Oct. 2011.

[11] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *USENIX Symp. on Networked Systems Design and Implementation*, 2011.

[12] Openephyra: Question answering system. https://mu.lti.cs.cmu.edu/trac/Ephyra/wiki/OpenEphyra.

[13] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX Symp. on Operating Systems Design and Implementation*, 2010.

[14] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. 2012.

[15] A. Rasmussen, M. Conley, R. Kapoor, V. Lam, G. Porter, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *ACM Symp. on Cloud Computing*, 2012.

[16] A. Rasmussen, G. Porter, M. Conley, G. Madhyastha, R. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *USENIX Symp. on Networked Systems Design and Implementation*, 2012.

[17] D. Shue, M. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *USENIX Symp. on Operating Systems Design and Implementation*, 2012.

[18] D. Yang and C. Stewart. Zoolander: Modelling and managing replication for predictability. 2011.