Cloud analytics: Do we *really* need to reinvent the storage stack?

Rajagopal Ananthanarayanan, Karan Gupta, Prashant Pandey, Himabindu Pucha
Prasenjit Sarkar, Mansi Shah, Renu Tewari
IBM Research

Abstract

Cloud computing offers a powerful abstraction that provides a scalable, virtualized infrastructure as a service where the complexity of fine-grained resource management is hidden from the end-user. Running data analytics applications in the cloud on extremely large data sets is gaining traction as the underlying infrastructure can meet the extreme demands of scalability. Typically, these applications (e.g., business intelligence, surveillance video searches) leverage the MapReduce framework that can decompose a large computation into a set of smaller parallelizable computations. More often than not the underlying storage architecture for running a MapReduce application is based on an Internet-scale filesystem, such as GFS, which does not provide a standard (POSIX) interface.

In this paper we revisit the debate on the need of a new non-POSIX storage stack for cloud analytics and argue, based on an initial evaluation, that it can be built on traditional POSIX-based cluster filesystems. In the course of the evaluation, we compare the performance of a traditional cluster file system and a specialized Internet file system for a variety of workloads for both traditional and MapReduce-based applications. We present modifications to the cluster filesystem's allocation and layout information to better support the requirements of data locality for analytics applications. We introduce the concept of a metablock that can enable the choice of a larger block granularity for MapReduce applications to coexist with a smaller block granularity required for traditional applications. We show that a cluster file system enhanced with metablocks can not only match the performance of specialized Internet file systems for MapReduce applications but also outperform them for traditional applications.

1 Introduction

Cloud computing is a compelling new paradigm that provides a scalable, virtualized infrastructure as a service, thereby, enabling the end-user to exploit supercomputing power on-demand without investing in huge infrastructure and management costs. This potential for unlimited scaling has made possible a plethora of cloud-based data analytics applications that can process extremely large sets of data. These include newer applications for business intelligence, semantic web searches, video surveillance search, medical image analysis along with traditional data-intensive scientific applications such as satellite image pattern matching. A common feature in all these applications is that they are extremely parallel and their data access bandwidth requirements dominates other resource requirements.

Such data-intensive applications where the computation can be easily decomposed into smaller parallel computations over a partitioned data set are a perfect match for Google's MapReduce framework [5] that provides a simple programming model using map and reduce functions over key/value pairs that can be parallelized and executed on a large cluster of machines. More recently, an open source version of MapReduce developed under the Apache Hadoop project is becoming a popular platform for building cloud data analytics applications.

The underlying architecture for cloud computing typically comprises of large distributed clusters of low-cost servers in concert with a server virtualization layer and parallel programming libraries. One of the key infrastructure elements of the cloud stack, for data analytics applications, is a storage layer designed to support the following features: (1) scalable – to store petabytes of data, (2) highly reliable – to handle frequently-occurring failures in large systems, (3) low-cost – to maintain the economics of cloud computing, and (4) efficient – to best utilize the compute, network and disk resources. The prevailing trend is to build the storage layer using an Internet scale filesystem such as Google's GFS [6] and its numerous clones including HDFS [1] and Kosmix's KFS [2]. The essential aspect of these filesystems is that they provide extreme scalability with reliability by striping and replicating the data in large chunks across the locally attached storage of the cluster servers, but simplify design and implementation by not providing a POSIX interface or consistency semantics. Thus, they work well for MapReduce applications but cannot support traditional applications. We refer to such MapReduce focused file systems as *specialized* in the rest of the paper.

In this paper, we revisit the debate on the need of a new non-POSIX storage stack for cloud analytics and argue, based on an initial evaluation, that it can be built on traditional POSIX-based cluster filesystems. Existing deployments of cluster file systems such as Lustre [7], PVFS [3], and GPFS [8] show us that they can be extremely scalable without being extremely expensive. Commercial cluster file systems can scale to thousands of nodes while supporting 100 GBps sequential throughput. Furthermore, these file systems can be configured using commodity parts for lower costs without the need for specialized SANs or enterprise-class storage. More importantly, these file systems can support traditional applications that rely on POSIX file API's and provide a rich set of management tools. Since the cloud storage stack may be shared

across different classes of applications it is prudent to rely on standard file interfaces and semantics that can also easily support MapReduce style applications instead of being locked in with a particular non-standard interface.

To this end, we address the challenges posed by the access characteristics of cloud analytics applications to traditional cluster file systems. First, we observe that MapReduce-based applications can co-locate computation with data, thus reducing network usage. We present modifications to the cluster filesystem's data allocation and data layout information to better support the requirements of data locality for analytics applications. Next, we observe that using large stripe unit sizes (or chunks) benefits MapReduce applications at the cost of other traditional workloads. To address that, we introduce a novel concept called *metablock* that can enable the choice of a larger block granularity for MapReduce applications to *coexist* with a smaller block granularity required for pre-fetching and disk accesses for traditional applications. While most analytics applications are read-intensive, we also enable *write affinity* that can better the performance of storing intermediate results by writing data locally.

We compare the performance of both an Internet scale filesystem (Hadoop's HDFS) with a commercial cluster filesystem (IBM's GPFS) over a variety of workloads. We show that a suitably optimized cluster filesystem can match the performance of HDFS for a MapReduce workload (ideal data access pattern for HDFS) while outperforming it for the data access patterns of traditional applications. Concurrent to our work, researchers at CMU have undertaken an effort to provide support for Hadoop's MapReduce framework with PVFS [9]. It should be noted that we don't report HDFS performance for traditional file benchmarks since these benchmarks cannot be run on HDFS (even running with a FUSE layer only provides a subset of the POSIX interface).

2 Challenges

In this section, we evaluate the suitability of cluster file systems for cloud analytics applications. In our study, we selected for comparison the HDFS (Hadoop 18.1) filesystem which is the de-facto filesystem for Apache's Hadoop project and IBM's GPFS cluster filesystem which is widely deployed in high-performance computing sites and whose source was readily available to us for modification.

The hardware configuration we used is based on the IBM iDataPlex modular hardware architecture consisting of a single iDataPlex system with 42 nodes in two racks, where each node has 2 quad-core 2.2 GHz Intel Core2Duo CPUs, 8 GB RAM and 4 750 GB SATA drives. The nodes are connected by 2 Gigabit Ethernet switches (one per rack) with a 1 Gbps inter-switch link. The switch is Blade Network Technologies G8000 RackSwitch with 48 1 Gbps ports. The software running on each of these nodes in Linux 2.6.18 (CentOS 5.3) with two disks dedicated to the ext3 file system for storing intermediate results from computations and the remaining two disks dedicated to either GPFS or HDFS. We use 16 nodes in the experiment with 8 nodes on either rack.

File System	Completion Time (seconds)	Network MB Transferred
HDFS	10	10
GPFS	220	66980

Table 1: Initial Evaluation of GPFS and HDFS.

Function shipping. The first drawback we found of cluster file systems is that they do not support shipping computation to data, a key feature exploited by the MapReduce class of applications [5]. In addition, the default block sizes are small which leads to a high task overhead for MapReduce applications that schedule one task per data block.

To evaluate the effect of function shipping, we measured performance of a simple MapReduce grep application with GPFS and HDFS. The input to the grep application is a 16 GB text file. The Hadoop implementation did not take advantage of any block location information in GPFS and function shipping was not enabled as a result. Furthermore, we used the default block size of 64 MB in HDFS, whereas for GPFS we used a block size of 2 MB with pre-fetching turned on by default.

Table 1 shows that HDFS is 22 times faster than GPFS in executing the simple MapReduce application. The lack of co-location of computation with data, and the use of small blocks, are the main reasons for the slow-down in GPFS. The table shows that GPFS transfers several orders of magnitude more data over the network. In fact, the total amount of data transferred exceeds the input data size because of the default pre-fetching in GPFS. The filesystem sees 2 MB of data being read sequentially and pre-fetches multiple data blocks to satisfy expected reads.However, the map task for the next block may be scheduled on another node and thus most of the pre-fetched data is not used.

High Availability. Another key requirement for data intensive applications is the ability to mask the failures of commodity components. Programs should be able to recover and progress in the event of multiple node and disk failures. This requires the data to be replicated across multiple nodes such that in the event of a node or disk failure, the computation can be restarted on a different node. Specialized file systems are designed based on this philosophy, and are able to tolerate multiple failures in the infrastructure.

In comparison, cluster file systems have traditionally been designed to use underlying data protection techniques (such as RAID) in shared storage to circumvent failures. However, the clusters that run data intensive applications typically do not use shared storage due to concerns regarding cost and bandwidth limitations, and instead attach local storage to each node in the cluster. While cluster file systems will run on nodes with locally attached storage (without replication or shared storage), the file systems will suffer data loss in the event of node or disk failures.

Some cluster file systems (like GPFS) do provide data and meta-data replication as a means to survive node failures. The mechanism of replication can vary across file systems.

GPFS, for example, uses a single source replication model, with the writer forwarding copies to all replicas. Specialized file systems, in contrast, use pipelined replication due to two important considerations: first, the out-bound bandwidth at the writer is not shared across multiple streams unlike the single-source model; second, write data can be pipelined in sequence from a node to the next node in the pipeline while the data is being written in the node.

For traditional applications, cluster file systems allow the use of concurrent writers to the same file, enabling the sharing of write bandwidth across multiple nodes. MapReduce applications usually have multiple simultaneous writers (to different files), so we don't expect the benefits of single-source replication to be significant. We hypothesize that it is possible for cluster file systems to match the write performance of specialized file systems and validate that in the experimental evaluation in Sections 4 and 5. However, we are continuing to explore the use of pipelined replication in cluster file systems.

3 Metablocks

Clearly, the grep application in the previous section demonstrated that running a MapReduce based application on a specialized file system has much better performance. In this section, we first attempt to mimic the basic properties of a specialized file system in GPFS and show the limitations of this approach. Next, we introduce the concept of a metablock, highlight the challenges in implementing the concept and demonstrate that GPFS is able to match the read performance of HDFS for MapReduce applications.

Large blocks. One approach would be to mimic the properties of specialized file systems as attempted in [9]. To achieve this, we increase the block size to a large value (16 MB) so that the map task and disk seek overhead is reduced (as one map task is assigned to each data block and will fetch the entire block for processing). Furthermore, we expose GPFS's block location information to the MapReduce layer in Hadoop so that tasks could be scheduled on the node where the data resides. In addition, we align the records in the input file with block boundaries, because a lack of alignment could result in the fetch of a large data block just to read a partial record that straddles a block boundary. Finally, we turned pre-fetching off to avoid the network traffic of transporting large data blocks. This particular version of GPFS is referred to as GPFS_lb (GPFS with large blocks).

To validate whether the approach would work, we use the same experimental setup as in Section 2 but with an input size of 80 GB. Figure 1 shows the relative performance of GPFS_lb and HDFS in the experimental setup. The execution time of GPFS_lb is almost the same as that of HDFS, but the network overheads of GPFS_lb and HDFS are 2 GB and 1.4 GB of data transferred over the network during the duration of the experiment.

However, the performance parity with HDFS comes at a price. Turning off pre-fetching and making the unit of caching large in GPFS_lb is detrimental to the performance of traditional filesystem workloads. Pre-fetching has been demonstrated to be extremely beneficial for sequential workloads and small block sizes are ideal for random workloads. To verify these effects, we compared unmodified GPFS to GPFS_lb using the popular Bonnie filesystem benchmark [4]. The results of the experiment are shown in Table 2 and show a marked performance degradation for random workloads with the optimizations used in this section. There is an improvement for sequential workloads due to the large block size but the scale is not commensurate to the extent of the previously mentioned degradation. Bonnie also output other results that were consistent with the conclusions from the experiment.

File System	Normalized Random Performance	Normalized Sequential Performance
Unmodified GPFS	1	1
GPFS_lb	0.15	2.29
GPFS_mb	0.99	1.19

Table 2: Evaluation of GPFS optimizations with Bonnie.

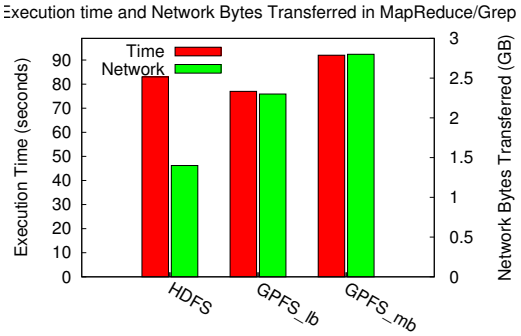

Figure 1: Execution time and network bytes transferred for MapReduce grep with HDFS, GPFS_lb and GPFS_mb (16 nodes, and 80 GBytes input data).

Metablocks. The results of the evaluation indicate an interesting tradeoff in optimizing for data intensive and traditional applications. While a large block size is needed to minimize seek overheads and create a reasonable number of tasks in MapReduce applications, a small block size is needed for effective cache management and to reduce the pre-fetch overhead particularly when application records could span multiple blocks on different disks. Ideally, we need the best of both worlds where both seeks and pre-fetching are optimized so that both MapReduce and traditional applications can be supported. If the cluster file system could expose a large node-local block size to the MapReduce application and use a smaller block size for internal book-keeping, data transfer and pre-fetching, we can achieve the tradeoff. To better understand how we can manage this, we first describe the block allocation strategy used by GPFS.

GPFS implements wide-striping across the file system where large files are divided into equal sized blocks, and consecutive blocks are placed on different disks in a round-robin fashion. An allocation map keeps track of all disk blocks in the file system. To enable parallel updates to the allocation

bit map, the map is divided into a large number of lock-able allocation regions, with at least n regions for an n node system. Each region contains the allocation status of $1/n^{th}$ of the disk blocks on every disk in the file system. This bitmap layout allows GPFS to allocate disk space properly striped across all disks by accessing only a single allocation region at a time. This approach minimizes lock conflicts because different nodes can allocate space from different regions. The allocation manager is responsible for keeping the free disk space statistics loosely- up-to-date across the cluster.

To balance the block size selection tradeoff, we define a new logical construct called a *metablock*. A metablock is basically a consecutive set of blocks of a file that are allocated on the *same* disk. For example, 64 blocks of size 1 MB could be grouped into a 64 MB metablock. The GPFS round-robin block allocation is modified to use a metablock as the allocation granularity for the striping across the disks. Consequently, the block location map returned to the MapReduce application is also at the metablock granularity with the guarantee that all blocks in the metablock are in the same disk. Internally for all other pre-fetching and accesses, GPFS uses the normal block size granularity (which is 1 MB in our example).

However there are two important challenges in implementing metablocks in GPFS – *contiguity* and *fragmentation*. First, it may not be possible to get a region with a set of blocks that is able to satisfy the contiguity requirement of a metablock. In such a situation, the node trying to allocate a metablock will need to request a region with a contiguous set of blocks that can be used to build a metablock. However, a request to the allocation manager may incur network latency and affect the performance of a MapReduce application. To remedy the situation, a node prefetches a pool of contiguous regions ahead of time and requests new regions when the cardinality of the pool drops below a threshold. This means that a node will always have a ready pool of contiguous regions and will not incur network latency in the path of an I/O request.

Second, allocating contiguous sets of blocks can lead to fragmentation in the allocation map, with skews in the free space of the regions in the allocation map depending on the nature of requests to the allocation manager. To address this issue, we relax the requirement for contiguity and only allocate sets of blocks which are contiguous only around 1 MB. This level of contiguity is reasonable for maintaining optimal sequential read and write performance, provides node-level locality, and minimizes the fragmentation problem.

We evaluate the effectiveness of the metablock allocation scheme for the MapReduce grep application. The experimental setup and the input data size (80 GB) is identical to that in the previous experiment. Here, we experiment with HDFS (as described before) and GPFS enhanced with a 16 MB metablock size and 512 KB block size, and no prefetching. The results demonstrate that the execution time of GPFS with metablocks (referred to as GPFS_mb) is within 10% of that of HDFS, while the network traffic is 2× worse than that of HDFS[1]. Further, when we enabled controlled prefetching in GPFS (by specifying prefetch percentage), we incurred additional network traffic proportional to prefetch percentage.

A possible cause for concern is that the metablock optimization, which changes GPFS's allocation scheme, could have affected the performance of traditional applications. To confirm this hypothesis, we compared unmodified GPFS to GPFS_mb. The results of the experiment are shown in Table 2 and show no marked difference between the two file systems. The other results from Bonnie were also consistent with this result. Consequently, we conclude that metablocks do not hurt the performance of GPFS for traditional applications. It is important to note that this change to the allocation policy of the cluster file system does not impact the interface to the applications, and preserves the POSIX semantics provided by the unmodified system.

4 Real-life Benchmarks

We selected three benchmarks to analyze the relative efficiencies of the specialized and cluster file systems and their effect on MapReduce applications: Hadoop grep, Teragen and Terasort applications. Teragen does a parallel generation of a large data set and is consequentially write-intensive. The grep application does a search for a regular expression on the input data set and is read-intensive and Terasort does a parallel merge-sort on the keys in the data set and does heavy reading and writing in phases.

The experimental setup was the same as before except for the changes noted below. We used a replication factor of 1 to evaluate basic data access efficiency of the two file systems and also measured the effect of using a replication factor of 2 in order to evaluate the relative performance of n-way and pipelined replication. We assumed that a replication factor of 2 is appropriate for a 16-node cluster, while the default value of 3 in GFS and HDFS are more appropriate for clusters with hundreds of nodes. A higher replication factor increases write overhead but may speed read performance by exposing more opportunities for load balancing MapReduce tasks.

We used the default block size of 64 MB for HDFS and set the metablock size for GPFS to be 64 MB as well, for a fair comparison. We found that using 1 MB as the block size of GPFS was the best compromise between the performance of traditional and MapReduce applications, and results presented here use that value.

Furthermore, we ran the benchmarks on 16 node clusters with two configurations - in the first, all nodes were in one rack, while in the second, the nodes are equally distributed across 2 racks. The 1-rack setup essentially provides 1 Gbps links between each node-pair, while the 2-rack setup has a network bottleneck in the form of a single 1 Gbps link between the two 8-node sub-clusters. In the 2-rack setup, when we enable 2-way replication, we configure the file systems to replicate

[1]We have isolated this issue to an unusual interaction between our data ingestion and GPFS allocation, and are improving the performance further.

each block so that one copy is on each rack, for better fault tolerance.

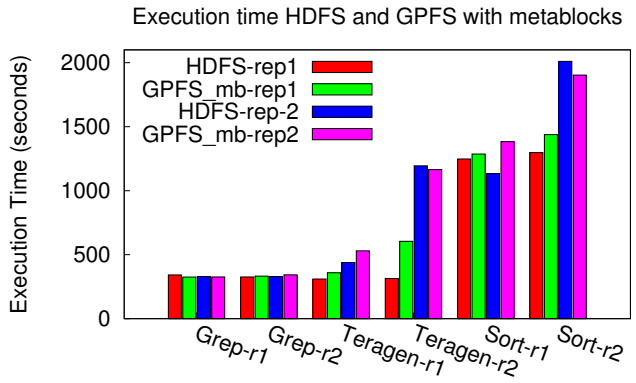

Figure 2: Benchmark evaluation of HDFS and GPFS_mb, using 160GB of input data and 16 nodes; replication factor = 1 (rep-1), 2 (rep-2). The 1-rack configuration is marked as r1, and the 2-rack configuration as r2.

Figure 2 shows the execution times of the HDFS and GPFS with metablock support (referred to as GPFS_mb) for the selected applications. The figure shows six clusters of four bars each, with two clusters for each benchmark. The grep benchmark is dominated by reads to locally attached disks in a node and is unaffected by a higher replication factor which impacts writes. Similarly, the benchmark shows no difference in execution times in the 1-rack and 2-rack cases due to the lack of network traffic.

The Teragen benchmark shows a difference in the write behavior of GPFS_mb and HDFS. When the replication factor is 2, GPFS_mb stripes all writes across the cluster while HDFS writes the first replica to the local node and the second replica to another node (on the remote rack, in the 2-rack case). Consequently in this benchmark, we see the added latency caused by GPFS_mb's use of the network in the 1-rack experiments, and the 2-rack experiments with replication factor 1. However, with a 2-rack configuration and 2-way replication, the 1 Gpbs link becomes the bottleneck for both file systems and the performance is equivalent.

Finally, the Terasort benchmark presents a mixed I/O workload to the file systems, and we see that the execution times of HDFS and GPFS_mb are roughly equal in the 1-rack experiments. The differences in the two rack experiments can be explained by the difference in network utilization for writes, just like in the Teragen experiments.

5 Future Optimizations

The results above encouraged us to look more closely at avenues for improvement of cluster file systems for MapReduce workloads. The most important was trying to make writes as network efficient in GPFS as they are in HDFS (due to the first replica being written to the local node). We designed an extension to metablocks which has allowed GPFS to potentially match the performance of HDFS for writes as well. The extension involves adding an `ioctl` call to GPFS which lets an application specify the set of hosts to be used by the metablock allocation scheme for a particular file. This allows Hadoop applications to specify that the first copy of data should reside on the local host, which is the policy used by HDFS. This technique reduces the network traffic during writes, and significantly improves write performance (up to a factor of 5).

True to our theme, we use GPFS with pre-fetching enabled to benefit traditional as well as MapReduce workloads. This, however, exposes two interesting questions we are currently exploring: (1) Can we design an adaptive prefetching scheme such that it only consumes spare network bandwidth, and does not contend with critical network traffic? (2) Can any MapReduce workloads benefit from such prefetching, thereby outperforming HDFS?

Similarly, we are also pursuing use cases of MapReduce workloads where GPFS, can in fact, outperform HDFS by leveraging features unique to a true file system such as ability to cope with client-side caching, and simultaneously support random and sequential workloads.

6 Conclusions

This paper evaluates the debate whether cluster file systems can potentially match the performance of Internet scale filesystems for cloud-based analytics applications. We examine the requirements of data intensive applications and show that cluster file systems are deficient in support for large block sizes and exposing block location information to MapReduce applications. To remedy this, we introduce the concept of metablocks that provide the illusion of large blocks for MapReduce applications, while providing the benefits of small blocks for traditional applications at the same time. We show that a cluster file system enhanced with metablocks can provide the best of both worlds performance.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback on this work; Frank Schmuck and the GPFS team for their insights on GPFS design, implementation and tuning.

References

[1] Hadoop Distributed Filesystem. `http://hadoop.apache.org`.

[2] Kosmos Filesystem. `http://kosmosfs.sourceforge.net/`.

[3] Parallel Virtual Filesystem. `http://www.pvfs.org/`.

[4] The Bonnie Filesystem Benchmark. `http://www.textuality.com/bonnie/`.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation*, pages 137–150, December 2004.

[6] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM SOSP, October 2003.*, 2003.

[7] Lustre. The lustre storage architecture. `http://www.lustre.org/`.

[8] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[9] W. Tantisiriroj, S. Patil, and G. Gibson. The crossing the chasm: Sneaking a parallel file system into hadoop. In *SC08 Petascale Data Stroage Workshop*, 2008.