MixApart: Decoupled Analytics for Shared Storage Systems

Madalin Mihailescu[⋆†], Gokul Soundararajan[†], Cristiana Amza[⋆]
University of Toronto[⋆], NetApp, Inc.[†]

Abstract

Distributed file systems built for data analytics and enterprise storage systems have very different functionality requirements. For this reason, enabling analytics on enterprise data commonly introduces a separate analytics storage *silo*. This generates additional costs, and inefficiencies in data management, e.g., whenever data needs to be archived, copied, or migrated across silos.

MixApart uses an integrated data caching and scheduling solution to allow MapReduce computations to analyze data stored on enterprise storage systems. The front-end caching layer enables the local storage performance required by data analytics. The shared storage back-end simplifies data management.

We evaluate MixApart using a 100-core Amazon EC2 cluster with micro-benchmarks and production workload traces. Our evaluation shows that MixApart provides (i) up to 28% faster performance than the traditional *ingest-then-compute* workflows used in enterprise IT analytics, and (ii) comparable performance to an ideal Hadoop setup without data ingest, at similar cluster sizes.

1 Introduction

We design a novel method for enabling distributed data analytics to use data stored on enterprise storage. Distributed data analytics frameworks, such as MapReduce [12, 19] and Dryad [22], are utilized by organizations to analyze increasing volumes of information. Activities range from analyzing e-mails, log data, or transaction records, to executing clustering algorithms for customer segmentation. Using such data flow frameworks, data partitions are loaded from an underlying commodity distributed file system, passed through a series of operators executed in parallel on the compute nodes, and the results are written back to the file system [19, 22]. The file system component of these frameworks [13, 21] is a standalone storage system; the file system stores the data on local server drives, with redundancy (typically, 3-way replication) to provide fault tolerance and data availability.

While the benefits of these frameworks are well understood, it is not clear how to leverage them for performing *analysis of enterprise data*. Enterprise data, e.g., corporate documents, user home directories, critical log data, e-mails, need to be secured from tampering, protected from failures, and archived for long-term retention; *high-value* data demands the enterprise-level data management features provided by enterprise storage.

Traditional analytics file systems [13], however, have been designed for *low-value* data, i.e., data that can be regenerated on a daily basis; low-value data do not need enterprise-level data management. Consequently, these file systems, while providing high throughput for parallel computations, lack many of the management features essential for the enterprise environment, e.g., support for standard protocols (NFS), storage efficiency mechanisms such as deduplication, strong data consistency, and data protection mechanisms for active and inactive data [26].

These disparities create an environment where two dedicated reliable storage systems (silos) are required for running analytics on enterprise data: a feature-rich *enterprise silo* that manages the total enterprise data and a high-throughput *analytics silo* storing enterprise data for analytics. This setup leads to a substantial upfront investment as well as complex workflows to manage data across different file systems. Enterprise analytics typically works as follows: (i) an application records data into the enterprise silo, (ii) an extraction workflow reads the data and ingests the data into the analytics silo, (iii) an analytics program is executed and the results of the

analytics are written into the analytics file system, (iv) the results of the analytics are loaded back into the enterprise silo. The workflow repeats as new data arrives.

To address the above limitations, we design MixApart to facilitate scalable distributed processing of datasets stored in existing enterprise storage systems, thereby displacing the dedicated analytics silo. MixApart *replaces* the distributed file system component in current analytics frameworks with an end-to-end storage solution, XDFS, that consists of:

- a stateless *analytics caching layer*, built out of local server disks, co-located with the compute nodes for data analytics *performance*,
- a *data transfer scheduler* that schedules transfers of data from consolidated enterprise storage into the disk cache, just in time, in a controlled manner, and
- the enterprise *shared storage system* for data reliability and data management.

The design space for MixApart includes two main alternatives: (i) *unified caching*, by having the XDFS cache interposed on the access path of both the analytics and the enterprise workloads, (ii) *dedicated caching*, by having the XDFS cache serve analytics workloads only. Moreover, in terms of implementation, for each design option, we can either leverage the existing HDFS [13] codebase for developing our cache, or leverage existing file caching solutions e.g., for NFS or AFS [8] file systems. With MixApart we explore the design of a dedicated cache with an implementation based on the existing Hadoop/HDFS codebase. Our main reasons are prototyping speed and fair benchmarking against Hadoop.

We deployed MixApart on a 100-core Amazon EC2 cluster, and evaluated its performance with micro-benchmarks and production workload traces from Facebook. In general, the results show that MixApart provides comparable performance to Hadoop, at similar compute scales. In addition, MixApart improves storage efficiency and simplifies data management. First, the stateless cache design removes the redundancy requirements in current analytics file systems, thus lowering storage demands. Moreover, with classic analytics deployments, ingest is usually run periodically as a way to synchronize two storage systems, independent of job demands. In contrast, our integrated solution provides a dynamic ingest of only the needed data. Second, MixApart eliminates cross-silo data workflows, by relying on enterprise storage for data management. The data in the cache is kept consistent with the associated data in shared storage, thus enabling data freshness for analytics, transparently, when the underlying enterprise data changes.

2 MixApart Use Cases

MixApart allows computations to analyze data stored on enterprise storage systems, while using a caching layer for performance. We describe scenarios where we expect this decoupled design to provide high functional value.

Analyzing data on enterprise storage: Companies can leverage their existing investments in enterprise storage and enable analytics incrementally. There exist many file-based data such as source-code repositories, e-mails, and log files; these files are generated by traditional applications but currently require a workflow to ingest the data into an analytics filesystem. MixApart allows a single storage back-end to manage and service data for both enterprise and analytics workloads. Data analytics, using the same filesystem namespace, can analyze enterprise data with no additional *ingest* workflows.

Cross-data center deployments: MixApart's decoupled design also allows for independent scaling of compute and storage layers. This gives the flexibility of placing the analytics compute tier on cloud infrastructures, such as Amazon EC2 [3], while keeping the data on-premise. In this scenario, upfront hardware purchases are replaced by the pay-as-you-go cloud model. Cross-data center deployments benefit from efforts such as AWS Direct Connect [2] that enable high-bandwidth connections between private data centers and clouds.

3 MapReduce Workload Analysis

Data analytics frameworks process data by splitting a user-submitted *job* into several *tasks* that run in parallel. In the input data processing phase, e.g., the *Map* phase, tasks read data partitions from the underlying distributed file system, compute the intermediate results by running the computation specified in the task, and shuffle the output to the next set of operators – e.g., the *Reduce* phase. The bulk of the data processed is read in this initial phase.

Recent studies [10, 16] of production workloads deployed at Facebook, Microsoft® Bing, and Yahoo! make three key observations: (i) there is high *data reuse* across jobs, (ii) the input phases are, on average, *CPU-intensive*, and (iii) the I/O demands of jobs are *predictable*. Based on the above workload characteristics, we motivate our MixApart design decisions. We then show that, with typical I/O demands and data reuse rates, MixApart can sustain large compute cluster sizes, equivalent to those supported by current deployments using dedicated storage.

3.1 High Data Reuse across Jobs

Production workloads exhibit high data reuse across jobs with only 11%, 6%, and 7% of jobs from Facebook, Bing, and Yahoo!, respectively, reading a file once [10]. Using the job traces collected, Ananthanarayanan et al. estimate that in-memory cache hit rates of 60% are possible by allocating 32GB of memory on each machine, with an optimal cache replacement policy [10].

MixApart employs a distributed cache layer built from local server drives; this disk cache is two orders of magnitude larger than an in-memory cache. For example, Amazon EC2 [3] instances typically provide about 1TB of local disk space for every 10GB of RAM. Hence, with these very large on-disk caches even a simple LRU policy suffices to achieve near-optimal hit rates, rendering cache thrashing irrelevant. Furthermore, we expect data reuse to increase as computations begin to rely on iterative processing – e.g., Mahout [14]. Interconnected jobs, such as job pipelines, will naturally exhibit data reuse in MixApart, as the current job input would be the output of the previous job. These trends and observations indicate that by caching data after the first access, MixApart can significantly reduce the number of I/Os issued to shared storage for subsequent accesses.

3.2 CPU-intensive Input Phases

In addition to high data reuse, data operations such as compression/decompression, serialization/deserialization, task setup/cleanup, and the sorting of outputs increase the average time spent on the CPU [9]. Zaharia et al. show, with 64 MB partition sizes, that the median map task duration is 19s for Facebook's workloads, and 26s for Yahoo!'s workloads [29]. Higher processing times indicate that a task's effective I/O rate is low, thereby there is ample time to move data from the shared storage to the distributed cache.

For instance, a task running for 20s to process a 64 MB input partition implies a task I/O rate of 25 Mbps. A storage server, with a 1 Gbps link, can sustain 40 such map tasks concurrently, even when all data is read from shared storage, with no loss in performance. The distributed cache layer further improves scalability. For example, with a 70% cache hit rate and a task I/O rate of 25 Mbps, more than 130 map tasks can process data in parallel from cache and shared storage.

3.3 Predictable I/O Demands

Average high data reuse and low task I/O rates confirm the feasibility of MixApart. Individual job patterns that deviate from the norm, however, could potentially impact scalability, by congesting shared storage when data reuse is low and aggregate job I/O demands are high. Hence, *coordination* is required to smooth out traffic and ensure efficient storage bandwidth utilization at all times.

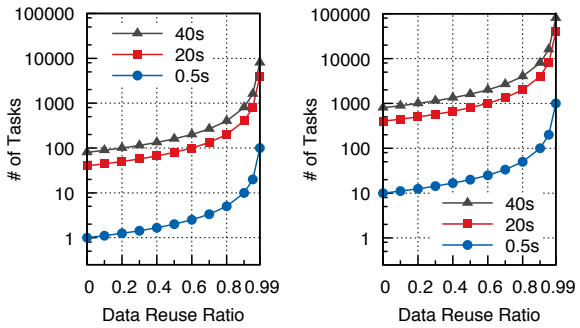


Figure 1: **Compute Cluster Size.** We show the number of parallel map tasks sustained by the cache + shared storage architecture for MapReduce analytics. Labels represent average map task durations – e.g., 0.5s for I/O intensive tasks. We plot values for cache-storage bandwidths of (a) 1Gbps, and (b) 10Gbps.

Previous studies observe that production MapReduce workloads have very predictable task times [10, 16]. In fact, the analysis of production traces for various Hadoop clusters [16] shows that processing jobs can be classified into less than 10 bins. In addition, tasks of a job have similar durations [10]. These two observations, combined with the fact that a task will read all data in its input partition, allow MixApart to coordinate shared storage traffic for individual jobs, and across multiple jobs.

3.4 Estimating Cluster Sizes Supported

We expand our analysis to estimate the average compute cluster sizes that can be supported by MixApart based on the workload characteristics introduced above, i.e., the typical data reuse across jobs, the computation to I/O ratio in the workload, and the storage bandwidth utilization. We use a back-of-the-envelope calculation similar to the one in Section 3.2 to derive the number of parallel map tasks under different parameters. Figure 1 plots the estimated cluster sizes (in number of map tasks). We vary the data reuse ratios from 0 to 0.99, task durations from 0.5s to 40s, and vary the storage bandwidth between 1 Gbps and 10 Gbps. The analysis shows that large analytics clusters can be sustained; for example, as shown in the Figure, with a data reuse ratio of 0.8 and average map task duration of 20s, MixApart can support 2000 parallel tasks.

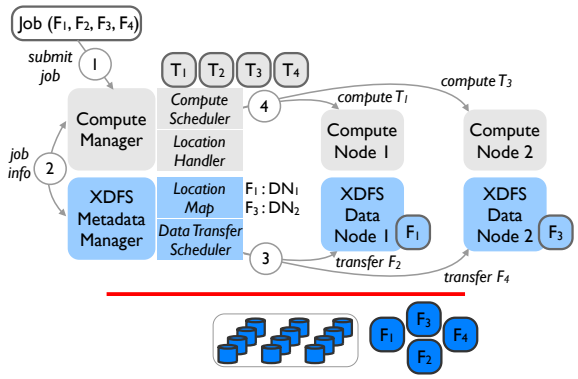


Figure 2: **MixApart Architecture.** We show the components of the compute and storage layers. We illustrate the execution flow of a job with four compute tasks at job submission (1), analyzing four files. The compute and XDFS managers exchange job-level information, e.g., I/O rates and data locations (2). Data from files F_1 and F_3 for tasks T_1 and T_3 is cached on DN_1 and DN_2, respectively, in the beginning. Tasks T_1 and T_3 are scheduled accordingly (4), and proceed in parallel with data transfers F_2 and F_4 (3). T_2 and T_4 will be scheduled next.

4 MixApart Architecture

MixApart enables scalable and efficient analytics for data stored on shared storage systems. Towards this, it uses the disk space available on each compute node in the analytics cluster as an on-disk caching tier for data on shared storage. Figure 2 shows all the components of the MixApart architecture. The architecture consists of two layers: a distributed compute layer and a distributed data storage layer (XDFS), along with the schedulers and metadata managers associated with the two layers.

The operational goals of MixApart are twofold: (i) preserve the scalability and performance benefit of compute-data co-location, and (ii) ensure efficient utilization of the compute nodes, cache and storage bandwidth. Towards this, MixApart uses per-job task I/O rates and the job scheduling policy to efficiently overlap computation with data fetches, whenever possible, by utilizing two correlated components:

- *Compute Scheduler*: a module that schedules tasks according to the XDFS cache contents and a job scheduling policy e.g., *FIFO*.
- *Data Transfer Scheduler*: a module that transfers data from shared storage to caches, as needed by compute tasks, based on job I/O rate predictions.

Once a job is submitted to the compute layer, this layer pre-processes the job to derive the job tasks, their data requests, and data request I/O rates. This job information is passed to the *data transfer scheduler*. Guided by the I/O rate prediction, the *data transfer scheduler* schedules data fetches from shared storage into one or more XDFS cache nodes, just in time, on behalf of job tasks.

While following a given job scheduling policy, e.g., FIFO, the *compute scheduler* schedules tasks out of order from its task queue, according to data availability in the cache nodes. Specifically, within each job, the *compute scheduler* prioritizes tasks with a higher fraction of data already in the cache, or on their way to the cache.

The logic of the two schedulers is correlated; we employ two scheduler components in order to have separation of concerns, and the flexibility to plug in any policy, independently, at the compute and I/O layers, respectively. For example, the compute scheduler can use FIFO or Fair job scheduling, or real-time job deadlines. On its end, the data transfer scheduler can work in conjunction with shared storage policies for I/O scheduling to achieve better utilization of the shared storage bandwidth. Moreover, for cross-data center deployments of MixApart, i.e., compute on-cloud, data on-premise, the data transfer scheduler can provide end-to-end WAN and shared storage bandwidth management.

Figure 2 shows an example of the integrated operation of the two schedulers. We see that the compute scheduler assigns tasks T_1 and T_3 to nodes DN_1 and DN_2, respectively, where the data needed by each of the two tasks is already cached. In parallel with these computations, transfers occur for data expected to be needed next by tasks T_2 and T_4; these tasks will be scheduled when their data is in the cache.

5 MixApart Compute Layer

The compute layer is composed of a set of compute nodes and our compute scheduler. Also a part of this layer, a *compute manager* (see Figure 2) accepts jobs into the system, and maintains job-level information, such as task I/O rates, that our schedulers need. Furthermore, a *location handler* associated with the compute scheduler keeps track of locations for data blocks currently in the cache and estimates of future data transfers.

Compute Manager: The MixApart *compute manager* maintains job-level information, such as job submission time, job priority, job data blocks stored in the cache, and the job-specific average map task I/O rate. When an I/O rate estimate is not available on job submission, the job-level I/O rate can be estimated very fast based on the monitored I/O rate of the first running job map task. Given that tasks within a job are highly homogeneous [10], this is expected to be a good predictor of I/O rates of all job tasks.

Compute Scheduler: Algorithm 1 illustrates the compute scheduler logic. The compute nodes advertise their resource availability by sending periodic *heartbeats*

Algorithm 1 **MixApart Data-Aware Compute Scheduler**

```
 1: if a heartbeat is received from node N then
 2:     if N has a free slot then
 3:         Sort JobQueue based on policy (FIFO, Fair)
 4:         for Job J in JobQueue do
 5:             if J has DataLocal(N) task T then
 6:                 Launch T on node N
 7:             else if J has RackLocal(N) task T then
 8:                 Launch T on node N
 9:             else if J has DataLocalInProgress(N) task T then
10:                 Launch T on node N
11:             else if J has RackLocalInProgress(N) task T then
12:                 Launch T on node N
13:             else if J has CacheLocal task T then
14:                 Launch T on node N
15:             else if J has CacheLocalInProgress task T then
16:                 Launch T on node N
17:             end if
18:         end for
19:     end if
20: end if
```

to the compute manager (step 1). Upon receiving a resource availability event, the compute scheduler sorts the job scheduling queue based on the policy (3). For example, for a *FIFO* policy, jobs are sorted based on arrival time and a user-defined job priority; for a *Fair* policy, jobs are sorted in increasing order of number of running tasks [29]. The compute scheduler assigns a task from the job at the head of the scheduling queue (4), in decreasing order of data partition locality (5-17), i.e., *data-local* for tasks with input data cached on the respective node, *rack-local* for tasks with input data cached on a different node in the same rack as the current node, and *cache-local* for tasks with input data anywhere in the XDFS cache. If the data accessed by a task is *in the process* of data transfer, the compute scheduler considers the *future* locality benefit and schedules accordingly (9-12, 15-16).

To saturate the compute tier, the task scheduler crosses job boundaries when assigning tasks, i.e., the task scheduler is work conserving. Specifically, if the tasks on behalf of jobs currently expected to run do not saturate the compute tier capacity due to lack of data in the cache, the scheduler selects tasks from lower priority jobs for which data is already in the cache.

6 XDFS Distributed Storage Layer

The XDFS on-disk cache layer contains a metadata manager, a set of cache nodes, and a data transfer scheduler.

XDFS Metadata Manager: The XDFS metadata manager implements both the distributed filesystem functionality as well as MixApart functionality. Namely, the manager implements group membership, failure detection of cache nodes, and interfaces to query the filesystem metadata. It also provides interfaces to query and manage the cache contents.

The metadata manager stores the data location information to be used by the compute tier for scheduling map tasks. It interfaces with the compute scheduler to assign tasks to nodes where a large fraction of the task's associated *input data partition* is cached. For every MapReduce job submitted to the system, the compute scheduler queries the metadata node for input data partition locations. For input partitions currently in the cache, the XDFS metadata manager replies with associated cache node locations. For input partitions not present in any cache node, the data transfer scheduler chooses cache nodes to transfer the data to, based on cache node utilization, and coordinates the data transfers from shared storage. The metadata manager notifies the compute scheduler of cache locations as soon as a transfer is initiated; the data locations are used by the compute scheduler to maximize compute-data locality.

XDFS Data Nodes: The XDFS data nodes store the data needed by MapReduce tasks on local storage. The data nodes copy data blocks into the cache from shared storage, as instructed by the data transfer scheduler. We use a default block granularity of 64 MB for large files; data blocks are stored on the local filesystem, e.g., ext4.

XDFS Data Consistency: Data consistency in the XDFS cache is maintained through notifications from the shared storage server upon changes to enterprise data. With the NFSv4 protocol, for example, this is achieved through client delegations and server callbacks [7]. Similarly, the CIFS protocol enables cache consistency through opportunistic locks [4]. Metadata consistency is maintained through directory delegations and/or periodic checking of directory attributes.

The XDFS manager holds read delegations for all files in the XDFS cache. The XDFS manager also maintains a list of files that represent inputs of active jobs. For files that are not part of the input of active jobs, the XDFS manager invalidates the corresponding file blocks in the XDFS cache as file update notifications are received from the enterprise storage. However, for files that represent inputs of active jobs, file blocks are marked as invalid after job completion. Invalid file blocks are refetched using the usual data transfer path, upon newer jobs analyzing the respective files.

Data Transfer Scheduler: The data transfer scheduler prefetches data across multiple jobs to utilize the shared storage bandwidth effectively. It allows administrators to specify bounds on bandwidth consumed by

Algorithm 2 XDFS Compute-Aware Data Transfer Scheduler

```
1: while AvailStorageBandwidth > 0 do
2:     Sort JobQueue based on policy (FIFO, Fair)
3:     J = head of JobQueue
4:     Sort J's BlockQueue based on block demand
5:     B = head of BlockQueue
6:     D = LocateAvailableDataNode()
7:     Transfer block B at IORate(J)
8:     AvailStorageBandwidth -= IORate(J)
9: end while
```

analytics workloads such that enterprise applications are not impacted. I/O scheduling provides performance isolation between data analytics and enterprise workloads at storage; we use quanta-based scheduling as it minimizes interference effects e.g., due to disk seeks, when different workloads share the same storage subsystem [28].

As shown in Algorithm 2, the data transfer scheduler mimics the job scheduling logic to ensure that the next expected job will have its input data available in the cache (steps 2-3). For example, when the *FIFO* policy is used at the compute tier, *job arrival times* and *priorities* determine the data transfer order. Alternatively, the Fair compute scheduler prioritizes jobs in increasing order of number of running tasks. In this case, the data transfer scheduler selects the next transfer from the job with the lowest number of *not-yet-analyzed cached data blocks* (including blocks under analysis). Transfer requests are further sorted per-job, by global block demand, i.e., the number of job tasks interested in a data block, across all jobs (steps 4-5). Destination cache nodes are selected based on available node capacity (6).

7 Prototype

We implemented the MixApart prototype within the Hadoop 0.20.203.0 version. The data-aware compute scheduler is a variant of the default task scheduler in Hadoop; the compute scheduler uses the FIFO job scheduling policy. We reused much of the HDFS codebase and retrofitted it for our purposes. Specifically, the NameNode, acting as XDFS metadata manager, loads shared storage metadata, i.e., NFS metadata, at startup time; data on enterprise storage is made visible through a local mount point on the NameNode. Furthermore, we modified HDFS to support cache blocks in addition to the reliable, persistent blocks. The compute-aware data transfer scheduler is implemented as a separate module at the NameNode (metadata manager) level.

XDFS can be viewed as HDFS with a redundancy of 1, and with a smart layer that ingests data from NFS into XDFS transparently and on-demand. HDFS, acting as reliable storage, has additional code to handle data loss; XDFS is stateless where a node failure simply results in additional fetches from the shared storage. Being a cache, we have modified XDFS to be aware of data transfers and take advantage of compute times for prefetching.

8 Evaluation

We evaluate the performance of MixApart in comparison with a dedicated Hadoop setting. We also study the impact MixApart has on enterprise workloads when running concurrently on shared storage.

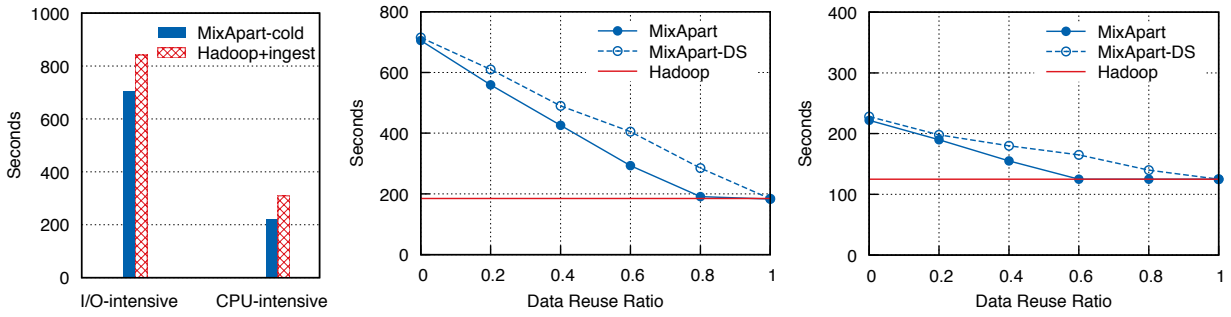
8.1 MixApart vs. Hadoop

We evaluate MixApart on a 100-core cluster connected to a NFS server, running on Amazon's Elastic Compute Cloud [3]; the XDFS caches access the shared storage using local mount points. We run micro-benchmarks, as well as production traces from a 3000-machine Hadoop deployment at Facebook. We begin by comparing the performance of MixApart to that of Hadoop with data ingest into the HDFS cluster to show the benefits of overlapping computation with I/O prefetches in MixApart. In subsequent experiments, we consider an ideal version of Hadoop with no data ingest, i.e., all data already placed in HDFS, for comparison against MixApart. Favoring Hadoop in these experiments allows to investigate the upper limit for MixApart's scheduling for concurrency.

8.1.1 Testbed

We evaluate MixApart on Amazon EC2 using three types of EC2 instances. We use 50 "standard-large" instances to deploy the compute nodes and the XDFS data nodes; each instance hosts a pair of compute and data nodes, and is configured with 2 virtual cores, 7.5 GB RAM, 1 Gbps network, and 850 GB of local *ephemeral* storage. Each compute node provides 2 map slots and 1 reduce slot. The compute manager and the XDFS metadata manager use one "standard-extra-large" instance configured with 4 virtual cores, 15 GB of RAM, 1 Gbps network, and 1690 GB of local storage. The shared storage is provided by a "cluster-compute-extra-large" instance configured with 23 GB RAM, 2 Intel®Xeon®X5570 quad-core CPUs, 10 Gbps network, and 1690 GB of local storage. We configure the shared storage server to store data on 4 EBS (*elastic block storage*) volumes assembled in a RAID-0 setting; the storage is exposed using NFS.

We use local instance storage for the XDFS cache; each instance has 850 GB of local storage for a total of



(a) Ingest Time (b) Pageview (I/O-intensive) (c) Pageview (CPU-intensive)

Figure 3: **Individual Jobs.** We show *job durations* for a MapReduce job running on a Wikipedia dataset. The *CPU-intensive* experiment processes the *compressed* data; the *I/O-intensive* processes the *uncompressed* data. In (a) we run MixApart with a cold cache (MixApart-cold). In (b), (c) we run MixApart with a warm cache. MixApart-DS has the data transfer scheduler disabled, i.e., parallel prefetching based on task I/O rates is not performed.

42 TB in the disk cache. The same disks are used for HDFS; HDFS is configured with 3-way replication. We limit the NFS server bandwidth to 1 Gbps in our experiments. This setting also mimics a production environment where enterprise workloads would still be able to use the remaining 9 Gbps.

8.1.2 Estimated Capacity

Figure 4 presents the estimated storage capacity needed for different Hadoop and MixApart configurations[1]. We compare standard HDFS (3 copies) with configurations that include enterprise storage either using double parity RAID-6 (NAS+HDFS), or in a disaster-recovery setup (DR-NAS+HDFS). The disaster-recovery setup mirrors the entire storage system to an offsite copy. We estimate the RAID overhead using the 12+2 (17% overhead) configuration. We assume all data is copied from the enterprise storage system into HDFS in the NAS+HDFS and DR-NAS+HDFS schemes; HDFS uses three copies even when a copy is kept on enterprise storage. This results in NAS+HDFS using 4.17 units of space for every unit of data and DR-NAS+HDFS consumes 5.35 units. In contrast, the capacity needed by MixApart varies with the fraction of active (cached) data. The MixApart configurations (NAS+MIX and DR-NAS+MIX) also maintain the entire dataset in enterprise storage and the active data is assumed to have three copies in the caching layer. The analysis shows that MixApart can be more efficient than HDFS; using the NAS+MIX scheme, half of the dataset can be analyzed while using less storage than HDFS.

[1] An analysis of the reliability and availability of the different configurations is beyond the scope of this paper.

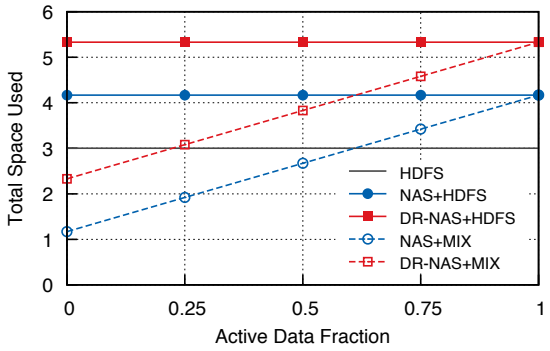


Figure 4: **Estimated Storage Capacity.**

8.1.3 Running Individual Jobs

We use a simple *pageview count* microbenchmark (derived from the standard *grep*) that aggregates page views for a given regular expression, from a Wikipedia dataset [1]. We use 12 days of Wikipedia statistics as the dataset; the data is 82 GB uncompressed (avg. file size is 300 MB) and 23 GB compressed (avg. file size is 85 MB). We run the job on both uncompressed and compressed input. Tasks processing uncompressed data are more I/O intensive than when input is compressed. Namely, for *uncompressed*, the effective map task I/O rates are roughly 50 Mbps. For *compressed*, I/O rates are 20 Mbps (due to higher CPU use). We denote runs on uncompressed data as *I/O-intensive*, and runs on compressed data as *CPU-intensive*.

We run each job in isolation, with various ratios of data reuse. The reuse ratio is the fraction of job input data in the XDFS cache at job submission time. HDFS uses 246 GB of storage capacity for uncompressed data, and 69 GB for compressed. MixApart uses 82 GB, and 23 GB for the cache, respectively, when the entire dataset

Mix	Jobs	Test
1	$A_1^\star$ B_1	data transfers for B_1 while A_1 running
2	A_2 B_2	just in time data transfers for B_2
3	A_3 B_3	just in time data transfers for both jobs at different I/O rates
4	$A_4^\star$ B_4	work conserving compute scheduling

Table 1: **Job Mixes.** We show a summary of the job mixes used to evaluate MixApart. * marks a high priority job.

is analyzed. As expected, the cache storage needs for MixApart are one third of HDFS capacity needed.

Figure 3 shows that: (i) overlapping the computation with data ingest improves performance, (ii) data reuse allows MixApart to match the performance of Hadoop+HDFS, and (iii) scheduling data transfers using compute-awareness enables efficient bandwidth use.

Data Ingest: We compare the performance of MixApart with no data in the cache (denoted MixApart-cold) to Hadoop (denoted Hadoop+ingest) – Figure 3(a). With Hadoop+ingest, data is ingested into HDFS before a job starts. By overlapping the computation and data transfers, MixApart reduces durations by approximately 16% for the *I/O-intensive* job and by 28% for the *CPU-intensive* job. Having shown that MixApart enables faster computations by overlapping compute and data ingest, in the following experiments, we compare MixApart with an ideal version of Hadoop that has all its input data in HDFS at job submission time.

Caching and Scheduling: Figures 3(b) and 3(c) show job durations for various data reuse ratios. Durations decrease with higher reuse as a bulk of the data is fetched from local disks, thereby avoiding the NFS bottleneck. Specifically, MixApart with 0.8 reuse matches the performance of Hadoop with all data in HDFS. The compute-aware data transfer scheduler improves performance by scheduling data transfers just-in-time.

Feasibility Analysis: Furthermore, the results are consistent with the analysis presented in Section 3. For *I/O-intensive*, the NFS server with 1 Gbps of bandwidth can sustain 20 parallel tasks (map task I/O rates are 50 Mbps). With 0.8 data reuse ratio, 80 parallel tasks can use local disks and 20 parallel tasks can use the NFS server to achieve the same performance as Hadoop with all data in HDFS. For *CPU-intensive*, lower I/O rates allow MixApart to match stand-alone Hadoop performance starting with 0.6 data reuse ratio.

8.1.4 Running Job Mixes

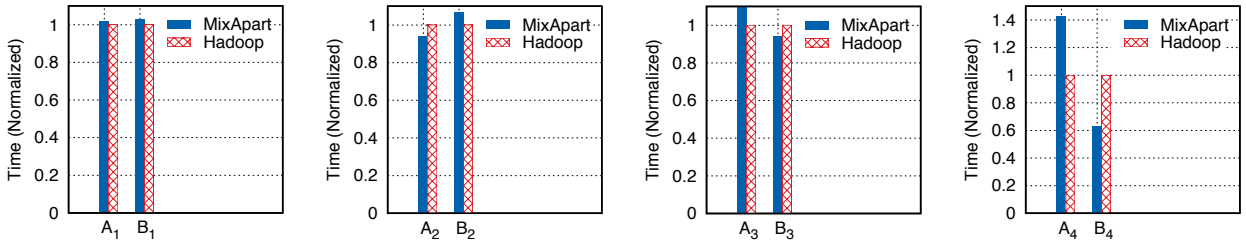
Next, we run mixes of two concurrent jobs to show the benefits of MixApart compute and data transfer schedulers. Table 1 summarizes the job mixes. To show the efficacy of MixApart, we build mixes using variations of the *pageview* job on different subsets of the Wikipedia dataset. Figure 5 shows job durations for the different mixes. We normalize job times to the time taken by Hadoop to complete a job. We do not include the HDFS data ingest time in these experiments. Despite fetching data on-demand into the cache, MixApart compares well to Hadoop.

Mix-1: Figure 5(a) shows durations when we submit two jobs, A_1 that has high data reuse and is I/O intensive, and B_1 that has low reuse and is CPU intensive. We mark A_1 to be high priority. Hence, A_1 uses the entire compute tier with its data served from the XDFS cache; the data transfer scheduler issues transfer requests for B_1's input data while A_1 is running. As most data is transferred proactively, job durations are similar. On MixApart, A_1 takes 2% more time and B_1 takes 3% more time.

Mix-2: This mix is similar to *Mix-1* but the two jobs have equal priorities. MixApart schedules A_2 to run in parallel with B_2. Input data for A_2 is entirely in the cache; for B_2, the data transfer scheduler coordinates just in time transfers from shared storage – tasks are scheduled as soon as data transfers are initiated. Figure 5(b) shows similar results with both frameworks – A_2 is 6% faster with MixApart than Hadoop and B_2 is 7% slower.

Mix-3: This mix – Figure 5(c), has jobs A_3, that has low data reuse and is I/O intensive, and B_3, with low reuse and CPU intensive. The jobs have equal priorities. MixApart schedules tasks from both jobs to run in parallel as data is being transferred from shared storage. The data transfer scheduler maximizes the number of parallel transfers, given the available storage bandwidth, using per-job I/O rates. As storage bandwidth is the bottleneck, A_3 runs 11% slower in MixApart while the CPU intensive B_3 runs 6% faster in MixApart. Being CPU intensive, the latency to transfer data to the cache is amortized by the higher CPU cost of the computation.

Mix-4: In the first three mixes, the two jobs have similar run times with Hadoop and MixApart. For MixApart, the data transfer scheduler ensures that tasks have their input data in the cache as needed, by scheduling transfers accordingly. In *Mix-4* (shown in Figure 5(d)), we submit job A_4 that has low data reuse and is CPU intensive, and job B_4 that has high data reuse and is I/O intensive. We mark A_4 to be high priority. A_4 has about 100 map tasks and it uses the entire compute tier with Hadoop; B_4 waits for A_4 to finish. In contrast, with MixApart, A_4 is bottlenecked on prefetching from shared storage, hence there are idle compute slots available; the work-conserving compute scheduler assigns tasks of B_4 to the idle compute slots. A_4 is 43% slower with MixApart than



(a) Mix-1 (b) Mix-2 (c) Mix-3 (d) Mix-4

Figure 5: **Job Mixes.** We show *job durations* for two concurrent MapReduce jobs running on Wikipedia statistics data, under various scenarios. Note that the HDFS data has been ingested prior to the experiments; adding the ingest time substantially increases the job time. Despite the handicap, MixApart performs comparably to Hadoop while providing greater storage efficiency. **Mix-1** (Figure 5(a)) runs job A_1 with high data reuse and job B_1 with low data reuse; B_1 waits for A_1 to finish. **Mix-2** (Figure 5(b)) runs A_2 with high reuse and B_2 with low reuse; A_2 and B_2 run in parallel. **Mix-3** (Fig 5(c)) runs A_3 and B_3 both with low data reuse; A_3 and B_3 run in parallel. **Mix-4** (Figure 5(d)) runs A_4 that has low data reuse and B_4 that has high reuse; A_4 and B_4 run sequentially with Hadoop and in parallel with MixApart (due to the work-conserving scheduler).

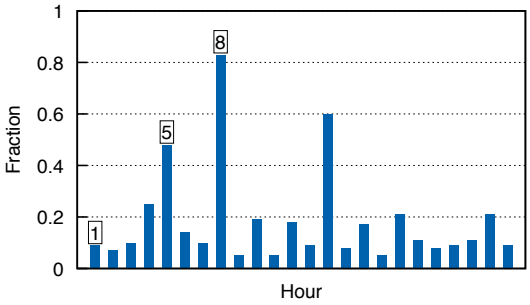


Figure 6: **Facebook Trace Characteristics.** The data reuse ratio of the trace changes over different time periods. We replay *Hour-1*, *Hour-5* and *Hour-8* to cover a broad spectrum of trace characteristics.

with Hadoop, as we ignore the data ingest into HDFS. B_4, however, runs 37% faster.

8.1.5 Running Production Traces

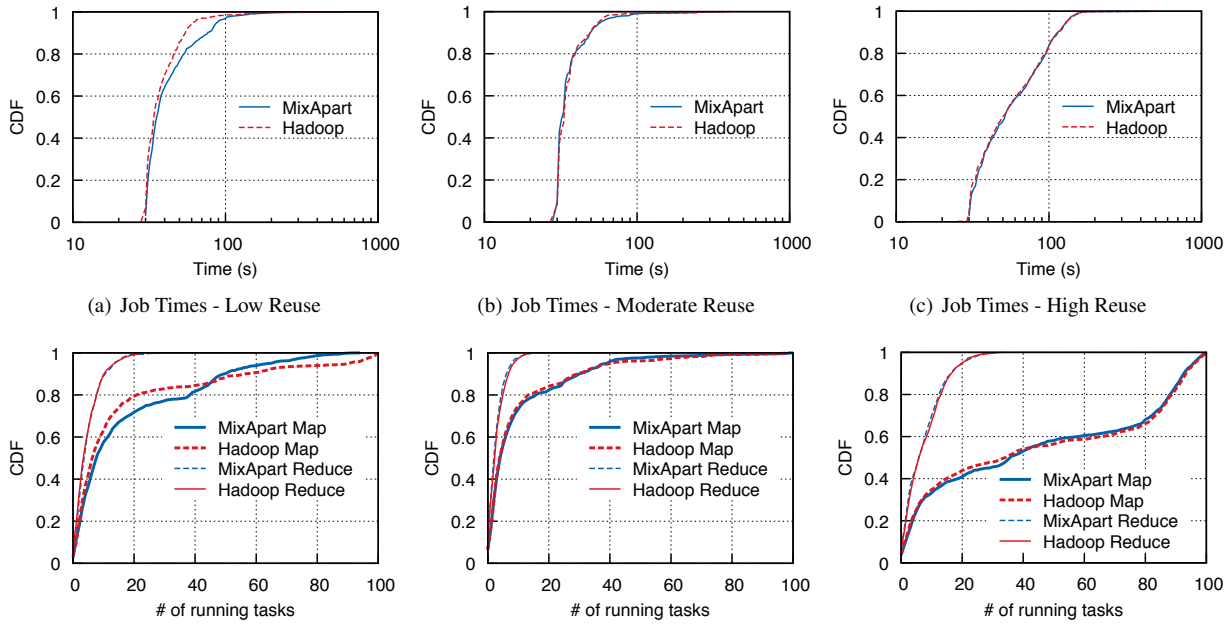
We use the SWIM framework [17] to evaluate MixApart with traces of MapReduce jobs collected from production environments; SWIM also provides a replay tool to re-run the traces in our environment. The traces preserve workload characteristics such as job input, intermediate, and output data sizes, and job submission patterns. SWIM also provides support for scaling down a trace captured from a large cluster to a smaller cluster, e.g., by scaling down job data sizes, while preserving the job compute to I/O ratios. We use the Facebook trace captured from a 3000-machine Hadoop cluster over a period of 1.5 months in 2010, consisting of 24 intervals of 1 hour job runs, sampled randomly from the original longer trace [16]. The trace is dominated by small jobs, with more than 90% of the jobs having a single Map task; the remaining jobs' input sizes range from 100s of GBs to TBs. Previous studies have shown this to be an attribute of most Hadoop environments [16].

	Low	Moderate	High
MixApart	46.3s	38.3s	64.5s
Hadoop	41.1s	38.2s	63.9s

Table 2: **Facebook Trace.** We report average job durations (in seconds) for the three Facebook trace segments. Only for the low data reuse segment, MixApart is approximately 10% slower than Hadoop, and similar otherwise.

Trace characteristics: Figure 6 shows that while the data reuse ratio varies over time, there is a moderate amount of reuse throughout the entire trace. We note that the Facebook trace only contains input path information but does not contain output path information, hence, the reuse estimates are conservative. We choose three 1-hour segments to evaluate MixApart: *Hour-1* with 0.09 data reuse ratio identified as *low reuse*; *Hour-5* with 0.48 data reuse ratio identified as *moderate reuse*; and *Hour-8* with 0.83 data reuse ratio identified as *high reuse*. We split each 1-hour trace into two sub-traces: the first 10 minutes are used to warm up the XDFS cache, and the remaining 50 minutes for the actual run. We note that a low percentage of the total data accessed in the actual run is represented by data transferred during warm-up: 1.6% for the low reuse trace, 5.9% for the moderate reuse trace, and 0.8% for the high reuse trace. We further scale the trace attributes, i.e., job sizes, to reflect the lower sized 100-core cluster that we use in our evaluation.

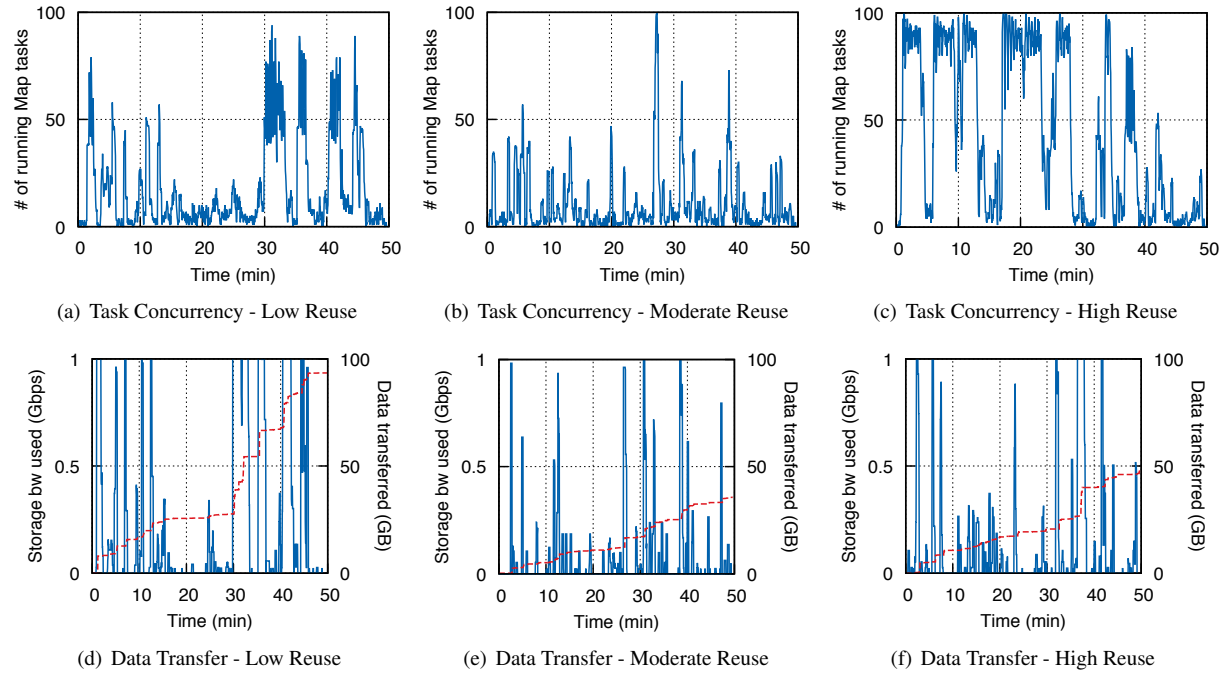
Results: Figures 7(a)-7(c) show the cumulative distribution function of job durations for Hadoop and MixApart. Table 2 shows job duration averages for each of the three trace segments. The results show that MixApart matches the performance of Hadoop for workloads containing moderate and high data reuse. For the low data reuse trace segment, 35% of the jobs run slower with MixApart – Figure 7(a). On average, MixApart is about 10% slower for the low reuse trace.



(a) Job Times - Low Reuse (b) Job Times - Moderate Reuse (c) Job Times - High Reuse

(d) Task Concurrency - Low Reuse (e) Task Concurrency - Moderate Reuse (f) Task Concurrency - High Reuse

Figure 7: **Facebook Trace.** We show job times and task concurrency for the three Facebook trace segments. Figures 7(a)-7(c) show the job running times. Figures 7(d)-7(f) show the task concurrency of both map and reduce tasks. The results show that MixApart matches the performance of Hadoop for the moderate and high data reuse trace segments.



(a) Task Concurrency - Low Reuse (b) Task Concurrency - Moderate Reuse (c) Task Concurrency - High Reuse

(d) Data Transfer - Low Reuse (e) Data Transfer - Moderate Reuse (f) Data Transfer - High Reuse

Figure 8: **Compute and Data Transfer Concurrency.** We show compute and shared storage statistics over time for the three Facebook trace segments. Figures 8(a)-8(c) show the task concurrency over time, and implicitly, the job submission rate for each trace (workload intensity). Figures 8(d)-8(f) show the storage bandwidth used at any given time (left y axis), the data transferred over time, as well as total data transferred over the duration of the trace segment (right y axis). MixApart schedules 40-50 parallel transfers to saturate the 1Gbps shared storage bandwidth. The compute tier schedules more than 50 tasks to run in parallel, due to proactive transfers and overall data reuse effects.

Figures 7(d)-7(f) show the cumulative distribution function of compute tier utilization in number of concurrent map and reduce tasks running at any given time. In general, MixApart matches the number of tasks run in parallel for both the Map and Reduce phases. The only divergence occurs when the workload has low data reuse: Hadoop achieves a higher compute tier utilization by running 90+ map tasks for 5% of the time. As expected, the Reduce phase performance is not affected.

Detailed Analysis: We gain insights into the performance of MixApart by studying the task concurrency at the compute layer and the data transfer concurrency in the cache layer. MixApart allows a fraction of the tasks to run reusing cached data, while scheduling remaining tasks to be started just in time as data is transferred from shared storage. Figure 8 shows the compute layer parallelism; we note that a majority of large jobs have map task I/O rates in the range of 20-30 Mbps, thereby enabling MixApart to schedule 40-50 parallel transfers from shared storage. This allows the compute tier to schedule more than 50 tasks to run in parallel, due to proactive transfers and overall data reuse effects. Figures 8(a)-8(c), 7(d)-7(f) also illustrate per-trace *job submission rate (workload intensity)*. The low data reuse trace has moderate workload intensity. The moderate reuse trace has low intensity, with less than 20 concurrent map tasks running for 80% of the time – Figure 7(e). The high reuse trace exhibits high workload intensity, as 30% of the time there are more than 80 map tasks running with both MixApart and Hadoop – Figure 7(f).

We also study the data transfer statistics. Figures 8(d)-8(f) show shared storage bandwidth utilization and total data transferred from shared storage over time. The low reuse trace transfers approximately 100 GB from shared storage with most of the data transfers occurring after the 30 minute mark. With higher data reuse and lower workload intensity, the moderate reuse trace transfers about 35 GB from shared storage. The high reuse trace transfers about 50 GB, due to a higher workload intensity.

Analysis of the Slowdown: We now focus on analyzing the causes of the difference between the performance of MixApart and Hadoop on the low reuse trace, where Hadoop schedules more concurrent map tasks than MixApart for around 5% of the time. We focus on a subset of jobs that create the bottleneck in MixApart.

Figure 9 plots job sizes (as number of tasks) and job durations by job id for this subset of jobs from the low reuse trace. The figure shows that, while most of these jobs are small in size, some are larger class jobs, containing 10s to 100s of tasks. The high degree of task concurrency, coupled with higher than average I/O rates

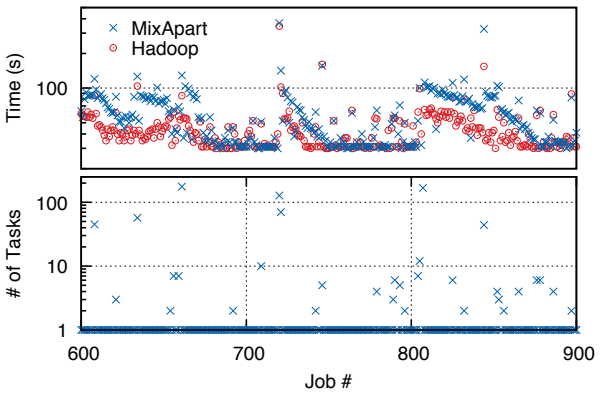


Figure 9: **Low Data Reuse Trace.** With many concurrent large I/O intensive tasks, the storage server gets bottlenecked causing subsequent tasks to be *backlogged* waiting for I/O requests to be scheduled.

for these tasks cause periods of saturation of the shared storage bandwidth we allocate for MixApart. Periods of limited concurrency at storage have some impact on the available concurrency degree within the compute tier as well. As there is low data reuse across jobs in this trace, most existing tasks for both large and small jobs are *backlogged*, waiting on I/O requests to be scheduled. The upper part of Figure 9 shows these queueing effects for both Hadoop and MixApart. However, the queueing effects in Hadoop happen due to saturation of the compute tier, while MixApart's limitation is due to the I/O scheduling policy coupled with low reuse rates.

8.2 Workload Isolation with MixApart

We study the impact of integrating data analytics and enterprise workloads in a traditional data center setup, focusing on the effects of sharing the storage system. For this purpose, we capture and replay the storage-level I/O requests of the Facebook low reuse trace; the low reuse trace is the most I/O intensive out of the three traces. As enterprise workloads, we generate I/O intensive random and sequential workloads using the flexible I/O tester [5]. We also replay a segment from enterprise storage traces collected in a production environment at Microsoft [25]; specifically, we select a 2-hour segment with high I/O intensity relative to the entire Microsoft trace set. The goal of this experiment was simply to validate the isolation between MapReduce workloads and enterprise workloads when running concurrently on shared storage.

We use quanta-based I/O scheduling, as it has been shown to guarantee performance isolation [28]. The quanta scheduler gives each workload a time slice of exclusive access to the underlying storage. We ran experiments for various quantum values and time splits and

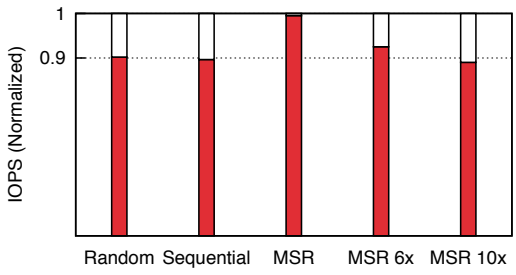


Figure 10: **Enterprise and Data Analytics Workloads on Shared Storage.** We show performance for enterprise workloads in I/O requests per second (IOPS) normalized to performance achieved with full storage access. Concurrently running enterprise and analytics workloads in MixApart causes little performance degradation in the normal case (MSR). With high I/O intensive enterprise workloads, quanta scheduling ensures performance isolation.

the results were similar. Figure 10 shows the results for a 500 ms quantum and a 90/10 time split for enterprise and analytics workloads, respectively. The random and sequential I/O intensive workloads validate the performance isolation enabled by quanta scheduling. The Microsoft traces are less I/O intensive than the random and sequential microbenchmarks. Hence, running analytics workloads concurrently with the regular MSR storage workload incurs negligible effects on these (MSR) enterprise workloads. We further replay the Microsoft trace segment at artificially generated higher speeds (MSR 6x and MSR 10x) to increase I/O intensity and show the performance impact of concurrently running data analytics in stress test enterprise situations. As with the random and sequential microbenchmarks, we see that quanta scheduling ensures performance isolation.

8.3 Overhead of Cache Invalidations

Updates to data in shared storage trigger cache invalidations, potentially reducing the effective cache hit rate for MixApart. The current system is architected for analyzing unstructured data such as logs, and/or corporate documents, produced by append-only enterprise workloads, and/or workloads with low rates of data overwrites. The append-only workloads are similar to workloads at Facebook, i.e., streams of user data, and are therefore backed by the Facebook trace evaluation. Moreover, recent studies of network file system workloads [24, 18] corroborate that overwrite rates are low in corporate environments, relative to the total enterprise data. While we do not evaluate the performance effects of cache invalidations, these effects are expected to be negligible for the types of workloads the current MixApart design targets.

8.4 Summary of Results

The results show that (i) our job trace selection and cluster sizing was appropriate for Hadoop to effectively use the compute tier with all data in HDFS, while allowing us to explore some variety in workload behavior at the same time, (ii) MixApart benefits from high data reuse and from jobs being compute intensive, on average, matching the performance of ideal Hadoop in almost all production scenarios studied, and (iii) the compute parallelism and performance achievable with MixApart is limited only when the I/O demands of all jobs saturate the shared storage bandwidth, and there is low data reuse across jobs.

9 Related Work

As the MapReduce paradigm becomes widely adopted within enterprise environments, the drawbacks of a purpose-built filesystem become glaring. Recent work has looked at techniques to interface Hadoop with enterprise filesystems [11, 27] as well as studied methods to provide enterprise storage features on top of existing MapReduce frameworks [6, 20]. Others have studied the benefits of in-memory caching for MapReduce workloads [10, 15, 30].

MixApart extends and complements existing work. We leverage the insights of Ananthanarayanan et al. [10] to argue for a disk caching layer. Moreover, we enable analytics on enterprise filesystems by leveraging a caching layer for acceleration without any changes to existing networked storage systems, while others [11, 27] modify enterprise filesystems (i.e., PVFS/GPFS) to support analytics workloads. In this sense, we believe that MixApart allows enterprise customers to leverage the infrastructure that has been deployed without additional hardware costs or downtime to existing storage systems.

Modified Cluster Storage Architectures: A body of recent efforts analyzed the aspects of incorporating Hadoop with existing storage systems [11, 27]. These works layer Hadoop's MapReduce compute tier on top of clustered filesystems – e.g., by enhancing GPFS/PVFS with large blocks and data location awareness capabilities. In contrast, with MixApart, we enable scalable, *decoupled* data analytics primarily for data stored in enterprise storage systems, through caching and prefetching.

Enhanced MapReduce Distributions: Enhanced distributions of Hadoop aim at incorporating enterprise file system features into HDFS. For instance, the Hadoop distribution from MapR Technologies [6] offers NFS access to the underlying data. In the same vein, erasure coding within HDFS has been explored recently [20]. While these attempts provide basic features needed for

enterprise environments, more time and effort are required to enhance commodity distributed file systems to the level of their enterprise counterparts.

Caching for Analytics: In-memory caching for data analytics workloads has been shown to improve performance, in general, due to data reuse [10, 15, 30]. We leverage and extend this observation to showcase the feasibility of MixApart; we improve the caching efficiency by using disk-based caches. By and large, works that optimize performance for specific job classes [10, 15, 30], can be layered on top of MixApart, just as they would be with frameworks such as Hadoop.

Location-aware Scheduling: The MixApart data-aware compute scheduler builds on previous work for location-aware compute scheduling [12, 23, 29]. In particular, we adapt the Hadoop task scheduler to be work-conserving, and to work in concert with the XDFS data transfer scheduler, by assigning tasks as soon as a transfer of data from shared storage is initiated.

10 Conclusions and Future Work

MixApart is a flexible data analytics framework that allows enterprise IT to meet its data management needs while enabling decoupled scale-out of the analytics compute cluster. MixApart achieves these goals by using an on-disk caching layer at the compute nodes and intelligent schedulers to utilize the shared storage efficiently.

We show that MixApart can reduce job durations by up to 28% compared to the traditional *ingest-then-compute* method. When the ingest phase is ignored for HDFS, MixApart closely matches the performance of Hadoop at similar compute scales. At the same time, MixApart uses a stateless disk cache without data replication within the compute cluster. We expect that the separation of concerns in this simple, decoupled design allows the most functional value in the following two realistic cases. First, MixApart can be used to support the customer option to leverage clouds for analytics, while maintaining the data within their private data center. Second, MixApart can be used to enable selective and transparent cache block refresh when the underlying enterprise data changes; this is an elegant solution for maintaining update consistency for analytics without modifying application semantics or manual interventions.

Acknowledgements

We would like to thank the program committee and the anonymous reviewers for their helpful comments. In particular, we thank our shepherd, Kimberly Keeton, for her suggestions for improving this paper. We also thank Ashvin Goel and Bianca Schroeder for their feedback. Finally, we thank Sethuraman Subbiah, Deepak Kenchammana, and the rest of the Advanced Technology Group (ATG) for their comments on the early ideas and the initial paper drafts.

References

[1] `http://dumps.wikimedia.org/other/pagecounts-raw`.

[2] Amazon Direct Connect. `http://aws.amazon.com/directconnect`.

[3] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`.

[4] Common Internet File System Protocol. `http://tools.ietf.org/html/draft-heizer-cifs-v1-spec-00`.

[5] Flexible I/O Tester. `http://linux.die.net/man/1/fio`.

[6] MapR Technologies. `http://www.mapr.com`.

[7] Network File System v4.1. `http://tools.ietf.org/html/rfc5661`.

[8] OpenAFS. `http://www.openafs.org`.

[9] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Disk-Locality in Datacenter Computing Considered Irrelevant. In *HotOS'11*.

[10] ANANTHANARAYANAN, G., GHODSI, A., WANG, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI'12*.

[11] ANANTHANARAYANAN, R., GUPTA, K., PANDEY, P., PUCHA, H., SARKAR, P., SHAH, M., AND TEWARI, R. Cloud Analytics: Do We Really Need to Reinvent the Storage Stack? In *HotCloud'09*.

[12] APACHE. Hadoop Project. `http://hadoop.apache.org`.

[13] APACHE. HDFS. `http://hadoop.apache.org/hdfs`.

[14] APACHE. Mahout. `http://mahout.apache.org`.

[15] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB'10*.

[16] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: a Cross-Industry Study of MapReduce Workloads. In *VLDB'12*.

[17] CHEN, Y., GANAPATHI, A., GRIFFITH, R., AND KATZ, R. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS'11*.

[18] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *SOSP'11*.

[19] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*.

[20] FAN, B., TANTISIRIROJ, W., XIAO, L., AND GIBSON, G. DiskReduce: RAID for Data-Intensive Scalable Computing. In *PDSW'09*.

[21] GHEMAWAT, S., GOBIOF, H., AND LEUNG, S.-T. The Google File System. In *SOSP'03*.

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys'07*.

[23] KOZUCH, M. A., RYAN, M. P., GASS, R., SCHLOSSER, S. W., O'HALLARON, D., CIPAR, J., KREVAT, E., LOPEZ, J., STROUCKEN, M., AND GANGER, G. R. Tashi: Location-aware Cluster Management. In *ACDC'09*.

[24] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX'08*.

[25] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST'09*.

[26] PATTERSON, D. A., GIBSON, G., AND KATZ, R. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD'88*.

[27] TANTISIRIROJ, W., PATIL, S., GIBSON, G., SON, S. W., LANG, S. J., AND ROSS, R. B. On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS. In *SC'11*.

[28] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance Insulation for Shared Storage Servers. In *FAST'07*.

[29] ZAHARIA, M., BORTHAKUR, D., SARMA, J. S., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys'10*.

[30] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *HotCloud'10*.

NetApp, the NetApp logo, and Go further, faster are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries. Windows is a registered trademark of Microsoft Corporation. Intel and Xeon are registered trademarks of Intel Corporation. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.