Ignem: Upward Migration of Cold Data in Big Data File Systems

Simbarashe Dzinamarira*, Florin Dinu†, T. S. Eugene Ng*

Rice University*, EPFL†

***Abstract*—This paper investigates whether migrating cold data can yield significant speedup for big data jobs that run on modern big data file systems. Our work is motivated by two observations. First, improving the input stage of a job can provide significant speedup because many jobs spend a large part of their execution reading inputs. The second observation is that the inputs for many jobs are cold. Common techniques that aim to keep hot data in memory do not benefit these jobs.**

We analyze the Google production cluster trace data and find that the key ingredients for effectively migrating cold data do exist in such production environments. Encouraged by our findings, we design and implement Ignem, a framework for migrating cold data in big data file systems. We evaluate Ignem in a series of experiments and show that it provides significant speedup for both small and large jobs. Specifically, Hive queries are accelerated by up to 34%; the mean job duration in a trace-driven workload is reduced by 12% and the task duration by nearly 40%; other standalone jobs such as sort and wordcount also improve similarly by up to 30%.

I. Introduction

In recent years, the input stage of big data analytics jobs has become a more dominant part of the jobs' execution time. This has happened for two reasons. First, the input stage typically handles much more data than subsequent stages. Second, several recent proposals have shown how to accelerate the non-input part of the job. Common techniques that aim to keep hot data in memory do not benefit the input stage of many data analytics jobs because the input data for these jobs is cold. Together, these facts make the input stage a target for optimization with a large potential for speedup.

A measurement study of SQL workloads on Spark [26] reported that at the median, queries in various workloads spend up to 19% of their execution time blocked on disk IO. Nearly 80% of the disk IO in production workloads analyzed in this study is from reading map inputs. Eliminating these slow reads by moving inputs into memory could improve query execution by up to 15% [1]. In this same study, rewriting one query in C++ reduced CPU time by 2x because this eliminated a lot of overhead from using Scala. Shortening the body of the all queries by 2x would amplify the time spent on the initial reads to 25% [2] of the overall execution time. Further optimizations in the literature [22], [11], [15], [20] that improve CPU time by an order of magnitude will see initial reads become a more dominant part of the execution of jobs. We refer the reader to Figure 7 in [11] for a breakdown of the benefits each optimization could give. Initial reads matter even for iterative machine learning jobs like Logistic regression and K-Means. Reading inputs from disk can inflate the first iteration in each job by 15x and 2.5x respectively, compared to later iterations [37].

Although there are methods to have the file system automatically identify and keep hot data in memory, such methods often do not benefit a job's input stage. Hot data refers to frequently and recently accessed data. For example, Spark [38] achieved an order of magnitude speedup over Hadoop for iterative machine learning applications by keeping repeatedly accessed data and intermediate outputs in memory. However, the initial read of data from disk is not improved by these optimizations. Triple-H [19] uses both frequency and recency of accesses to compute a temperature for each block and moves the block into faster storage when the temperature rises above a certain threshold. PACMan [5] targets hot data that is already in memory and determines which data should be kept or evicted when memory pressure rises. The assumption that motivates keeping hot data in memory is that read accesses in the near future will likely be to the data that is currently hot. However, only keeping hot data in memory does not benefit a large class of jobs whose reads are on cold data. One study shows that over 30% of tasks in a production workload belong to jobs that access singly read data [5]. These are mostly recurring jobs that process new data such as logs or user click-stream data. The new data cannot all fit in memory but has to be stored on disk before it is processed. This data is often large and it is not processed immediately [35]. When the data is eventually accessed from disk, it is cold and the reads are not accelerated because the state of the art schemes would keep or move only hot data into memory.

In this work, we first analyze the Google cluster trace data so we can answer several questions that determine how feasible it is to effectively migrate cold data upward into memory in the storage hierarchy in such production environments: Can we identify cold data that will be accessed in the near future? Once identified, can this data be migrated into memory before it is accessed? Is there enough time and residual bandwidth for migration? Upon answering these questions, other questions also emerge: How much speed-up can be expected when migration is done successfully? Can migration work be scheduled in a manner that reduces disk

[1] $19\% \times 0.8$

[2] $\frac{15\%(Read\,time)}{\frac{80\%(CPU\,time)}{2}+20\%(IO\,time)}$

seeks and therefore improves disk throughput?

Our analysis yielded encouraging results and motivated us to design Ignem, a system for migrating cold data in big data file systems. We have implemented a prototype of Ignem and we evaluate it in a series of experiments. Ignem provides significant speedup for both small and large jobs. Specifically, Hive queries are accelerated by up to 34%; the mean job duration in a trace-driven workload is reduced by 12% and the task duration by nearly 40%. Other standalone jobs such as sort and wordcount also improve similarly by up to 30%.

II. Motivation

A. Where and why data migration is useful

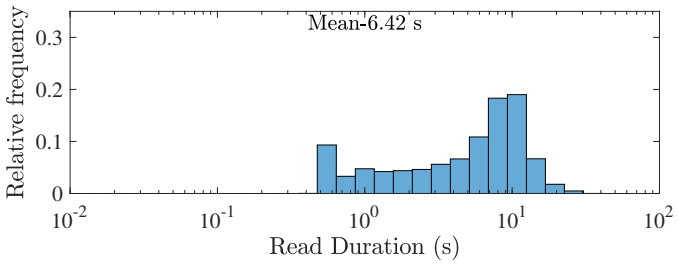
Data migration is beneficial because it can speed up the initial stage of a job, which often is a significant part of the overall job runtime. The reason for this is that the computation time in later stages is not proportional to the job input size. The initial stage often filters out or aggregates a large portion of the job input which makes the output of this stage much smaller than the job input size.

Prior work on MapReduce workloads at Google shows up to a 10:1 ratio between the job input and the output size of the map stage of the job [12]. Similarly, Rhea [17] shows a reduction of 2-20000x between input and output sizes for Hadoop mappers. Other studies also show similar data reduction [7], [9]. We further support this argument by running TPC-DS queries using the setup in Section IV and looking at the proportion of time jobs spend in the initial stage. On average, the map tasks account for 97% of total task runtime. These map tasks read inputs and filter out much of the data because of the SELECT statement and predicates in the WHERE clause. Such selectively is common amongst database queries and makes them good candidates for acceleration using data migration.

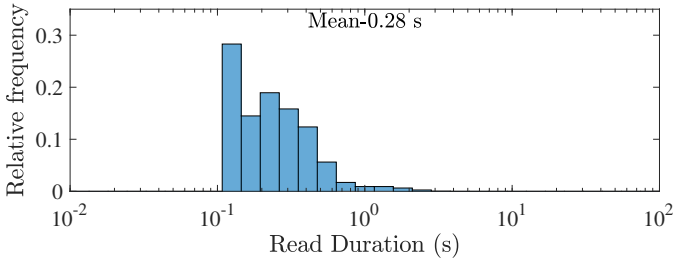
B. Migrating inputs to memory yields significant benefits

We run the SWIM workload [2] to showcase the potential benefits of migrating job input data into memory. The SWIM workload is a popular trace driven workload based on production jobs at Facebook. The workload is described in detail in Section IV. Job inputs are stored in the Hadoop Distributed File System (HDFS). We evaluate the effects of storing HDFS files on a hard disk drive (HDD), on a solid state drive (SSD), or in memory (RAM). The HDFS block size is set to 64MB.

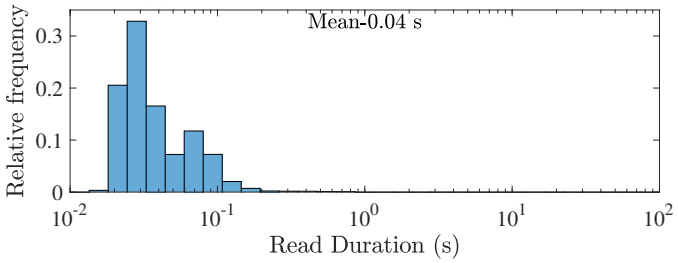
Figure 1 shows histograms for the time it takes a mapper task to read an entire HDFS input block. Reads from RAM (Figure 1c) are, on average, 160x faster than those from HDD (Figure 1a). This is because HDDs have lower sequential throughput and in addition, their performance is severely impacted by concurrent reads. Figure 2 shows a CDF of mapper task runtimes. Average task runtime for tasks that read from RAM is 23x smaller than for those reading from HDD. While significant, the speedup for task runtime is smaller than the speedup for HDFS block reads because tasks have other overheads unrelated to reading data.



(a) Block reads from HDD - Mean 6.42s



(b) Block reads from SSD - Mean 0.28s



(c) Block reads from RAM - Mean 0.04s

Fig. 1: On average, HDFS block reads from RAM are 160x faster than reads from HDD, and 7x faster than reads from SSD.

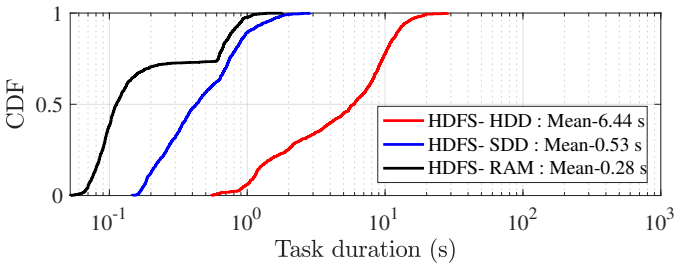


Fig. 2: Reading from RAM leads to a large speedup for mapper tasks

One approach to accelerate reads and reduce the negative effects of concurrency is to use SSDs. While HDFS block reads from SSD are faster than those from HDD, Figures 1b and 1c show that reading from RAM is still 7x faster. Similarly, task runtimes (Figure 2) are lower when reading from RAM rather than SSD. These results suggest that regardless of whether cold job input data is stored on HDDs or SSDs, migrating the data into memory is key to maximizing performance.

C. Conditions favorable for timely migration are present in production workloads

To maximize the performance benefit delivered by data migration, two conditions need to be met. First, migration should be timely, that is, it should be possible to migrate data before it is accessed. Second, there should be enough available memory to hold the migrated data while it waits to be accessed. With the support of a popular Google trace [28], we argue that both conditions hold in practice. The trace reports resource usage for over 12,000 servers during a month-long period. These resources include CPUs, memory, and disks. The trace also logs events such as the submission, scheduling, and termination of jobs and tasks.

1) Why timely migration is achievable today: Whether a block of data can be timely migrated depends on three factors: the amount of time available for migration, the amount of IO resources available for migration and the size of the data to be migrated.

For the rest of the paper, we use the term *lead-time* to refer to the amount of time available for migration. More precisely, lead-time for a task is the time between the earliest moment when migration can be triggered (usually job submission time) and the moment when the data is accessed by the task. Lead-time for a job is the time between job submission and the start of the first task in the job. Note that our definition of job lead-time is a lower bound since not all tasks in a job may start simultaneously.

Sources of lead-time The most important sources of lead-time in big data frameworks are queueing time and platform overheads.

Current systems queue tasks while they wait for the resources needed by the tasks to become available. Some systems queue tasks at the scheduler level [36] while others have queues at the node level [27]. In both cases, the rationale is to obtain high cluster utilization and resource packing efficiency by having tasks ready for execution and by having a diverse pool of task resource requirements to draw from. Our insight is that task queueing time can be leveraged for migrating data. For the Google workload, the mean and median job queueing times are 8.8 and 1.8 seconds respectively. For the purpose of this calculation job queueing time and lead-time are identical.

Per-platform overheads provide *additional* lead-time for migration. The most important source is inherent in the design of a large number of schedulers today. Centralized schedulers rely on a heartbeat mechanism to react to cluster changes and schedule new tasks. For scalability, heartbeat intervals are in the order of seconds. For example, the default heartbeat interval in Hadoop is 3 seconds. In addition to heartbeats, shipping binaries to workers [39] and JVM warm-up costs [21] can further increase lead-time.

Lead-time is sufficient As stated above, jobs in the Google workload take on average 8.8 seconds to be scheduled. Whether this lead-time is enough for data migration depends on the amount of IO that jobs perform. The Google trace reports task runtime as well as the time tasks spent blocked on disk IO. For each job in the trace, we sum the disk IO time for all its tasks. We then compare the lead-time for each job with the disk IO time. This comparison assumes the disk IO is served by one disk on a single machine. However, disk IO for a job may be parallelized across multiple disks and machines so more data could be migrated within the lead-time. Figure 3 shows the ratio between the time spent reading, and the lead-time. For 81% of jobs, the lead-time is greater than the read-time meaning there is enough time to migrate their entire inputs into memory. Note that this is despite the fact that we are using a lower bound on a job's lead-time. In addition to the 81%, other jobs whose input can only partially be migrated into memory may still benefit from migration by having some of their tasks sped up.

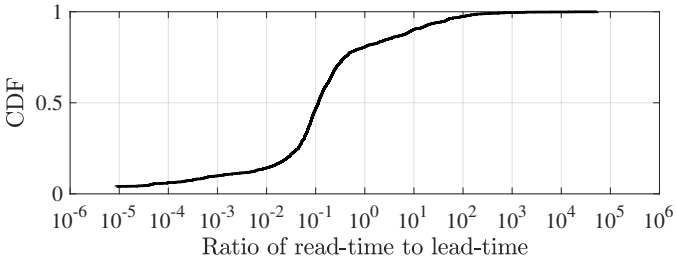


Fig. 3: For 81% of jobs in the Google trace, the lead-time is sufficient for migration.

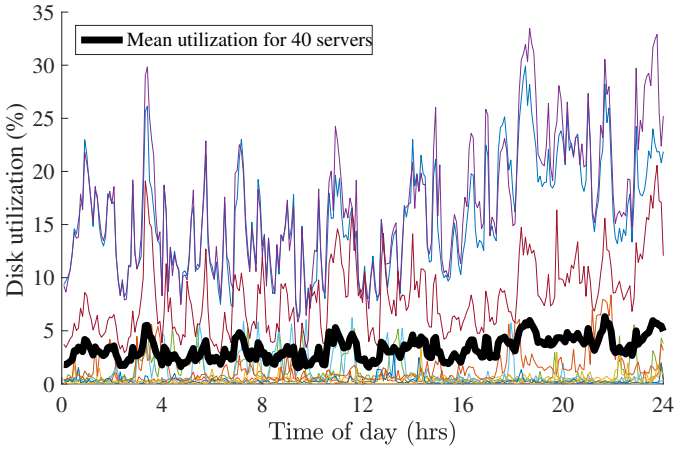


Fig. 4: Disk bandwidth utilization over a 24 hour period for servers in the Google workload. We plot individual timelines for 10 servers, and the mean utilization for 40 servers. There is abundant residual bandwidth that can be exploited for data migration.

Sufficient disk bandwidth for timely migration Our previous example showed that the lead-time was sufficient for a large number of jobs in the Google trace assuming a migration speed equal to that at which the mapper would process the data. In practice, migration can proceed faster. It is not limited by the compute rate since it happens before the data is accessed. Migration for a task may share disk resources with foreground reads and writes. Nevertheless, we argue that there is sufficient residual disk bandwidth in today's clusters to allow efficient migration.

The Google trace provides the time spent on IO for each individual tasks during intervals that are up to 5 minutes

long. Starting from this we derive disk utilization numbers for servers. We assume that the IO time for a task is uniformly distributed over the whole interval. We compute the disk utilization for a server at 1-second granularity. The disk utilization for a server for interval $[T-1s, T]$ is the sum of disk IO time for that interval for all tasks that run on that server and have an interval window that includes $[T-1s, T]$. Finally, we average server disk utilization over 5-minute windows.

Figure 4 shows the disk utilization for a number of servers in the Google cluster[28] for a 24h period. The behavior of these servers is representative of the other servers in the trace. At any point during the 24h period, the mean disk utilization of 40 randomly chosen servers is at most 5%. For all 12,000+ servers, the mean disk utilization during the 24h period is 3.1%; and 1.3% for the entire month the trace covers. This shows that current clusters are heavily over-provisioned for IO and this allows ample opportunity for migration. Prior work on a small academic cluster also shows that disks are often under-utilized [13].

2) There is enough memory to store migrated data: After data is migrated, it should remain in memory until it is accessed. Insufficient memory would result in skipped migrations or wasted work due to evictions. We perform a worst-case analysis to argue that modern servers have plenty of memory to serve migration. The Google trace shows that at on average 10 tasks run on a server at a time. Assuming the latest generation dual-socket CPUs, the number of tasks on a server at a given time is unlikely to be greater than 50. Further, assume that each of the 50 tasks is a mapper and each mapper reads a large 256MB HDFS block. This means that 12.5GB of RAM is sufficient to hold the migrated data for the maximum number of tasks that can run concurrently. This is a small amount of memory for today's servers which are provisioned with hundreds of GB of RAM. Another study's analysis of two private workloads [5] draws similar conclusions that there is sufficient RAM to keep a significant portion of the working set in RAM.

III. System Design and Implementation

This section describes the design and implementation decisions behind Ignem, our system for cold data migration. In the design discussion, we address several questions that arise when performing migration and we explain the rationale behind our design decisions. We believe our choices allow Ignem to perform migration effectively while being fault-tolerant and scalable. We then discuss how we implemented Ignem on top of HDFS, a widely popular, state-of-the-art big data file system.

A. Design

Ignem uses a master-slave architecture. The master is responsible for communicating with jobs and determining *what* data needs to be migrated. The slaves control *how* and *when* data is migrated into memory. Files are partitioned into blocks and are stored on the slaves. Slaves have only a block-level view of the data while the master has complete mappings of files to blocks and of blocks to slaves. This design is consistent with that of several of today's big data file systems [33], [16], [20].

The migration process is initiated when a client sends the Ignem master a list of files that a job will soon need to read. A client is any code that can send a request to Ignem. The master maps the files to blocks stored on the slaves and sends each slave the corresponding list of blocks. Each slave then independently reads the data blocks from disk to memory and manages when data is evicted from memory.

1) How can Ignem schedule migration work efficiently? How can it avoid disk contention and which migrations should it prioritize?: Starting migration right after job submission can be suboptimal for two reasons. First, not all migration may be equally beneficial. Second, it can result in poor disk performance when migrations for several jobs are performed concurrently.

Ignem slaves put incoming migration work into a queue before handling it. While the natural strategy is to migrate the queued blocks in a FIFO order, Ignem slaves prioritize migration for blocks belonging to jobs with smaller input sizes. This allows Ignem to improve the performance of more jobs. This prioritization also increases the likelihood of fully migrating a job's input during its lead-time. If two jobs have exactly the same input size we use job submission time as a tie-breaker. Prioritization in Ignem does not preempt the migration of a block once it has begun. To avoid disk bandwidth degradation due to concurrent reads, each slave only migrates one block at a time.

Ignem's migration is work-conserving, i.e. it does not delay pending migrations if there are no ongoing migrations. This ensures that Ignem exploits lead-time efficiently and keeps the disk well utilized.

2) How can Ignem deal with data replication?: Data replication is widely used in big data file systems. Each block of data is replicated on several servers for failure resilience. When performing migration for a replicated block the question is which and how many of the replicas to migrate to memory. Migrating several replicas wastes resources but provides additional opportunities for a task to execute on a server where its input data was migrated.

Ignem's approach is to randomly choose only one replica to migrate. This is based on the fact that network bandwidth is not a bottleneck in current data-centers [25]. Thus, even when a task cannot be scheduled on the server where its input was migrated, it can still efficiently read the block over the network. Ignem allows a task to specify locality preferences with respect to migrated blocks. This is a straightforward extension to current big data file systems because they provide APIs to allow tasks to query input locations and specify disk-based locality preferences.

3) How can Ignem avoid wasting disk bandwidth when migrating?: Migration may waste disk bandwidth in two cases. First, this can occur if the data that is migrated is never read. Ignem only migrates data for known future reads so this is not a concern in the common case. Job failures can lead

to migrated data being never read and we will explain later how Ignem deals with this. Second, under memory pressure, a migrated block could be evicted to make room for a new block. Ignem avoids this by not evicting data before it is used. The following discussion argues why we adopt this policy.

Ignem employs the *Do not harm rule* [8] which states that a block of data *A* should never be evicted from memory in order to make room for a block *B* that will be read later than *A*. Keeping *A* results in a sure hit in memory for *A*, and a potential hit/miss for *B*. Evicting *A* would result in a sure miss for *A* without any guarantee for *B*. Therefore keeping *A* is guaranteed to be at least as good as evicting it. In practice, it is hard to infer the order in which migrated blocks will be read because several tasks run concurrently on a server and their runtimes may differ. In this case, the *Do not harm rule* provides a conservative approach, assuming that *A* will be read first and thus ensures a performance improvement without wasting disk bandwidth.

4) How does Ignem avoid memory leaks in its migration buffer?: For each migrated data block, a slave maintains a reference list of job IDs for jobs that are expected to read the block. A job ID is appended to this list when the slave receives a command to migrate the block. When a job is completed, the job submitter issues an evict instruction that causes the job ID to be removed from the reference list. The evict instruction is sent via the Ignem master and it is processed in a similar manner to the migration instruction. A block is kept in memory as long as its reference list is non-empty and is evicted when it becomes empty. Evicting data as soon as no future accesses are expected gives Ignem a low memory footprint.

In the case that a job fails or is terminated before it issues the evict instruction, the reference lists containing this job's input blocks still need to be cleaned. When a memory occupancy threshold is reached for the migration buffer, the slave queries the cluster scheduler to check if the job is still running. If the slave does not receive a confirmation that the job is still active, it removes the job from all block reference lists. This cleanup mechanism helps ensure that Ignem keeps data in memory only for jobs that are still running.

As a performance optimization to keep memory usage low, we also allow Ignem to implicitly remove a job from a block's reference list as soon as the job reads the block of data. This causes data to be evicted sooner if the reference list becomes empty. A job can opt into this implicit eviction mode when the job submitter issues the migration instruction.

5) How can Ignem achieve failure resilience?: Ignem is resilient to failures. When the master fails, a new master can quickly be started and it starts handling new requests. If just the master process fails, it can be restarted on the same server and no further failure handling is required. If the server itself fails, we launch the master on a different server. Clients know how to reach the master by reading the IP address and port of the master from a small configuration file placed on each server in the cluster. After the new master is launched, we update this configuration file and broadcast it to all servers. A backup master can also be kept active at all times, and have its address pre-listed in the configuration file. When the master fails, Ignem loses its state about which blocks are in memory, so eviction commands cannot be directed to the appropriate slaves. Ignem slaves purge the reference lists for all in-memory blocks when the master fails so as to be consistent with the new masters empty state. The failure of the master only results in a temporary performance loss for those jobs whose migration had already begun.

For slave failures, we adopt resilience mechanisms similar to those for slaves in HDFS. If only the slave process fails, it can be restarted on the same server. All data that has been migrated into memory is discarded. This causes a temporary loss of performance, but after being restarted, the slaves can handle new migration commands successfully. There is no memory leak when slaves fail because the operating system automatically cleans up the blocks in memory when a slave process is terminated. If the entire server fails, the file system removes the server from the namespace map. The Ignem master queries the file system to get the locations of blocks so it will receive an updated view with only live locations for replicas of each block.

6) Can Ignem scale?: Ignem has a master-slave architecture similar to HDFS, which has been shown to scale to thousands of servers [33]. The Ignem master only has to handle migration requests from job launchers, map these requests to blocks, and send migration instructions to slaves. These operations have a very small computational load compared to what other centralized components such as the Hadoop NameNode or Yarn ResourceManager already handle at scale.

In order to reduce RPC communication overheads, Ignem sends migration commands between the master and slaves in batches. To enable Ignem to handle more clients and slaves, requests from clients to the master and commands from the master to slaves can be processed in parallel by multiple threads. The amount of extra memory required by Ignem at every slave is negligible and consists of no more than 1KB per block.

B. Implementation

We implemented Ignem as an extension of the Hadoop Distributed File System (HDFS). Ignem is backward compatible with HDFS and therefore it can be easily deployed in real-world settings. Though the rest of this section discusses Ignem in the context of HDFS, the design principles behind it can be applied in other file systems. We implemented the Ignem master within the HDFS NameNode, and the Ignem slave within the DataNode.

1) Migration mechanism at the slaves: HDFS has the capability for users to explicitly lock files in memory. We build on top of this to migrate inputs into memory. The slaves use the *mmap* and *mlock* system calls to read a file in memory. First, *mmap* maps a portion of the file to part of the virtual address space of the slave process. The *mlock* call then locks the mapped region into RAM, preventing that memory from being paged out. The *mlock* causes data to be read from disk. When data is no longer needed the *munmap* system call is

used to evict the data from memory. The input data is read-only so there is no need to write back anything to disk once data is evicted from memory.

While is it possible to migrate data onto the slave's heap, this would require changing the IO path of future reads to access data from the slave's heap instead of just opening a file. The system calls above migrate data into the buffer cache where it is accessible to other processes. This also avoids double buffering.

2) Memory management: Ignem limits the amount of migrated data to a configurable maximum threshold. If this migration memory buffer is full, migration commands are queued until buffer space is available or until they are discarded due to missed reads.

Each slave has a hash-map that maps a job's ID to the list of blocks migrated for the job. This hash-map allows Ignem to efficiently locate the blocks that need to have their reference lists modified. A job ID can be removed from a block's reference list explicitly via the eviction command when the job completes, or implicitly when the job reads the data block. Reads calls in HDFS carry the job ID so Ignem slaves can extract the job ID and independently perform this implicit eviction without contacting the Ignem master. A job chooses whether or not to enable implicit eviction when the migration command is issued.

3) Modifying applications to use Ignem: Before a job is submitted to a system such as Yarn, a piece of code we call the job-submitter is run to configure the job. Configuring a job involves specifying which classes will be run and setting the job's input and output paths. The job-submitter is the best place to insert the migration call because it is the first element in a job's lifecycle.

Inside the job-submitter one can create an instance of the file system client (DFSClient). The DFSClient is used to perform namespace operations such as opening, closing, creating and deleting files. We extended the DFSClient in HDFS with a migrate method whose main argument is a list of files to be migrated or evicted. Extra arguments are used to select the operation to perform, and whether eviction will be explicit or implicit. The DFSClient communicates with the Ignem master (which is part of the NameNode) via Remote Procedure Calls (RPC).

Some MapReduce applications such as Sort have simple job-submitters where the list of input files is easily accessible. Such applications can be easily modified to use Ignem by adding a call to the migrate method on the DFSClient. Frameworks like Hive have more complex job-submitters that submit a sequence of MapReduce jobs for different stages of the query. Modifying Hive was more involved with regards to creating the list of files to be migrated, but the API to Ignem is still a single function call. The change to Hive is a one-off change to the framework. All queries that run on the framework then get their inputs migrated transparently.

IV. Evaluation

In this section, we evaluate our prototype of Ignem. We study the benefits or migrating cold data using a workload derived from a Facebook trace, two standalone MapReduce jobs, and several Hive queries. Each job requires only a few lines of code to enable it to use Ignem's migration service. We first describe our hardware and software setup, and our workloads; then we present experimental results.

A. Hardware setup and software configuration

Hardware setup - We run our experiments on an 8 server cluster. All servers run HDFS DataNodes processes, which we extended to implement Ignem slaves. In addition to being a slave, one server also runs the HDFS NameNode and Yarn ResourceManager. The NameNode and ResourceManager have very little computation load given the size of our cluster so we can run them on a worker without any significant impact on the worker's performance. The Ignem master is implemented as part of the NameNode.

Each server has a 1TB HDD drive, 128GB of RAM and a Xeon E5-1650 CPU with 6 cores and 12 hyperthreads. We have a 10Gbps network between the servers.

File system configurations - Most of our experiments involve three configurations of the file system. The first two configurations use default HDFS with Ignem disabled. In the first configuration, all input data is stored on disk while in the second configuration we force all inputs into RAM using the vmtouch tool [3]. We call this second configuration *HDFS-Inputs-in-RAM*. Vmtouch is run after input files are generated and it locks all DataNode files in memory. Vmtouch does not affect the outputs of jobs. Finally, we also run the workloads on Ignem. For all configurations, we flush the buffer cache before running jobs to ensure the inputs are on disk unless we have explicitly locked them in memory using vmtouch.

B. Workloads

The workloads we use in this paper all consist of MapReduce jobs that use HDFS as the underlying file system. All the jobs use Apache Tez [29] as the execution engine.

1) SWIM workload: The SWIM workload [10] is a trace-based workload derived from a production Hadoop cluster at Facebook. The trace reports the input, shuffle and output data sizes of the jobs that ran on the production cluster. The arrival times for jobs are also provided. We scale down the data sizes and inter-job arrival times. We evaluate Ignem using the first 200 jobs in the SWIM trace. We scale down the sizes of job inputs to adjust for our smaller cluster. The total input size for all 200 jobs after scaling is 170GB. We also reduce the inter-job arrival time by 50%.

85% of jobs in our workload read 64MB or less and the largest jobs read up to 24GB. The abundance of short jobs and a heavy tail is a feature in other cluster traces too [4]. We chose this workload because it is realistic and it also provides a challenging scenario for Ignem. On one hand, the short jobs read very little data so optimizing reads has a limited impact on their durations. On the other hand, the large jobs

	Absolute Duration (s)	Speedup w.r.t HDFS
HDFS	14.4	
Ignem	12.7	12%
HDFS-Inputs-in-RAM	11.4	21%

TABLE I: Despite counting in fixed overheads unrelated to reading input, Ignem significantly improves the average job duration by 12%. It realizes 60% of the upper bound benefit.

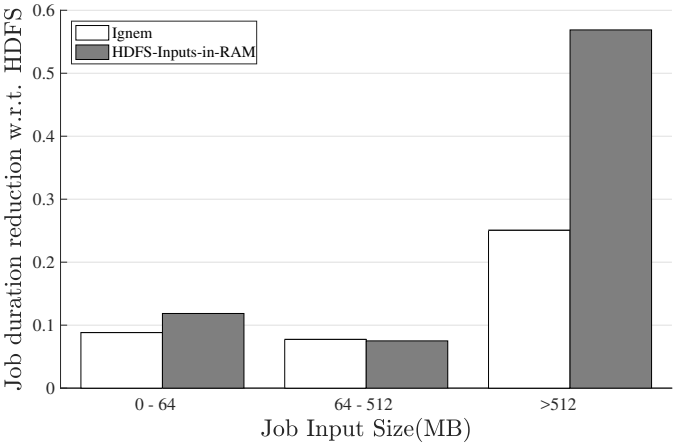


Fig. 5: Reduction in mean job duration for jobs, binned by input data size. Ignem speeds up small,medium and large jobs by 8.8%, 7.7% and 25% respectively

in the tail read so much data that there is unlikely enough time to migrate the whole input. Despite these challenges, our evaluation shows Ignem still provides significant speedup for jobs in this workload.

2) Standalone MapReduce jobs: To evaluate the benefits of migration in a more controlled setting, we also run sort and wordcount jobs by themselves. The sort experiment uses a 40GB dataset of random text. For wordcount, we vary the input size from 1 GB to 12 GB to study how the benefits of migration relate to the input size of a job and the available lead-time. We generate the wordcount input by concatenating a 400MB online text corpus [1] onto itself multiple times until we reach the target input size.

3) Hive queries: Lastly, we evaluate Ignem's benefits on several queries from the TPC-DS benchmark [24] using Hive. We added a hook into the Hive framework to enable it to instruct Ignem to migrate query inputs into memory. The hook is invoked when Hive finishes compiling each query.

C. SWIM Experimental results

1) Ignem significantly improves job duration in the SWIM workload: Table I shows the average job duration for jobs in the SWIM trace. When the SWIM workload is run with all input data in memory, the average job duration is 21% lower than when inputs are on disk. This 21% is an upper bound for what any cold data migration scheme could do. As we stated above, the SWIM workload leaves little room for improvement. Despite this, Ignem provides a speedup of 12%. This is nearly 60% of the upper bound.

	Absolute Duration (s)	Speedup w.r.t HDFS
HDFS	6.44	
Ignem	4.03	38%
HDFS-Inputs-in-RAM	0.28	96%

TABLE II: Mapper tasks in the SWIM workload run 2.6x faster when Ignem is used. The speedup is amplified further for mappers because they have less fixed overheads that are not related to reading inputs.

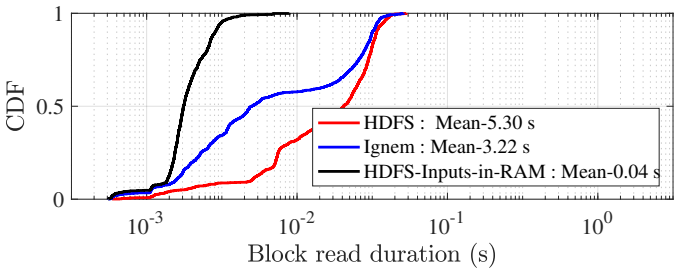
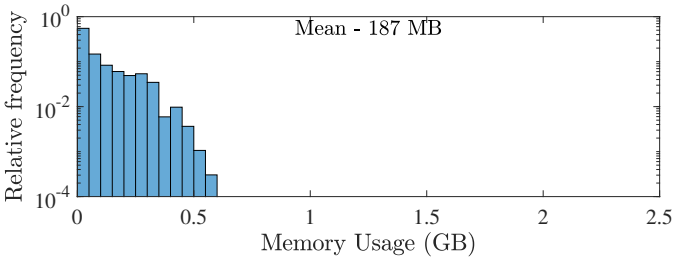


Fig. 6: Ignem significantly reduces block read durations, benefitting even blocks that are not migrated

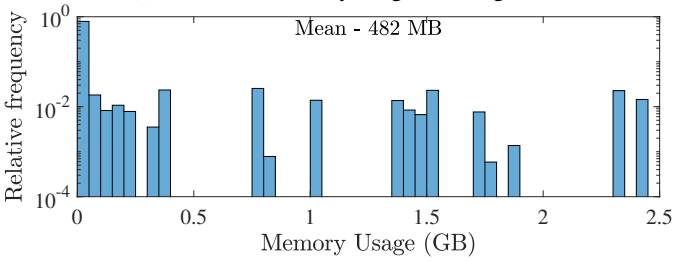
2) Larger jobs are more sensitive to read optimization: While Table I shows the overall performance over the whole workload, we also investigate how Ignem performs for different sized jobs. We divided our jobs into three bins by input size. Figure 5 shows the reduction in relative average job duration for each bin. For small jobs, Ignem provides an 8.8% speed up. Its performance is very close to that of *HDFS-Inputs-in-RAM* which means Ignem can usually migrate all inputs for these small jobs. The same is true for medium-sized jobs (64-512MB) and Ignem improves their duration by 7.7%. The fact that HDFS-Inputs-in-RAM's speedup drops for medium sizes jobs is an inadvertent artifact of our workload having few medium sized jobs and some of these jobs having high computational overhead. The computational overhead limits the job level speedup we can observe. In general though, jobs with a larger input size should experience a larger speedup when the inputs are in RAM. For the large jobs that read more than 512MB, having inputs in memory reduces their duration by nearly 60% on average. Ignem reduces the duration of these jobs by 25%. Even though Ignem cannot migrate the entire inputs of these jobs in time, the portion that is migrated has a large pay-off since IO is a more significant part of these jobs.

3) Task level gains are even more significant: Ignem's gains at the job level are diluted by parts of the job that cannot be improved by faster reads, such as shuffling, reducing and writing outputs. In this section, we zoom in to look at the speedup for map tasks since only these tasks read the input data and therefore are directly accelerated by Ignem. Table II shows Ignem improves average task duration by nearly 40%. Though this may not translate to an equally large gain for the jobs these tasks belong to, resources in the cluster are occupied by map-tasks for much less time. This allows more work to be packed into the cluster.

The lowest level of granularity we instrument are HDFS



(a) Per server memory usage under Ignem



(b) Per server memory usage of a hypothetical scheme that can migrate and evict data instantaneously

Fig. 7: A comparison of the memory usage of Ignem vs. a hypothetical scheme that performs migration instantaneously. The memory footprint of Ignem is 2.6x lower on average, yet Ignem can provide 60% of the benefit the hypothetical scheme.

block reads. Figure 6 shows a 40% reduction in the average block read time, which is similar to the benefit at the task level. These two numbers are similar because mapper tasks in the SWIM workload spend most of their time reading and perform very little computation. In Figure 6, we observe a large reduction in the block read duration for about 60% of blocks when using Ignem. This means roughly 60% of blocks are successfully migrated and read from memory by tasks. The rest of the block reads do not experience a large speedup because Ignem did not have enough lead-time to migrate them. However, there is still an improvement even for these blocks that are not migrated because Ignem reduces the disk contention these blocks experience by moving disk IO that would otherwise contend with these blocks earlier.

4) Ignem provides good performance while using very little memory: Figure 7 shows the relative frequency for the amount of memory used per server to store blocks migrated into memory. The histograms only show samples when memory usage is non-zero, in order to exclude times when the cluster is idle. We compare Ignem to a hypothetical scheme which can migrate and evict data instantaneously. The hypothetical scheme migrates the input when the job is submitted and evicts it when the job completes. This scheme cannot be implemented in practice because it is impossible to migrate data instantaneously. However, we use it a comparison point because it theoretically would provide the upper bound for speedup like HDFS-Inputs-in-RAM does. Ignem uses 2.6x less memory than the hypothetical scheme, but it is still able to provide 60% of the speedup that the hypothetical scheme would give.

	Absolute Duration (s)	Speedup w.r.t HDFS
HDFS	147	
Ignem	114	22%
HDFS-Inputs-in-RAM	75	49%

TABLE III: Sort workload results

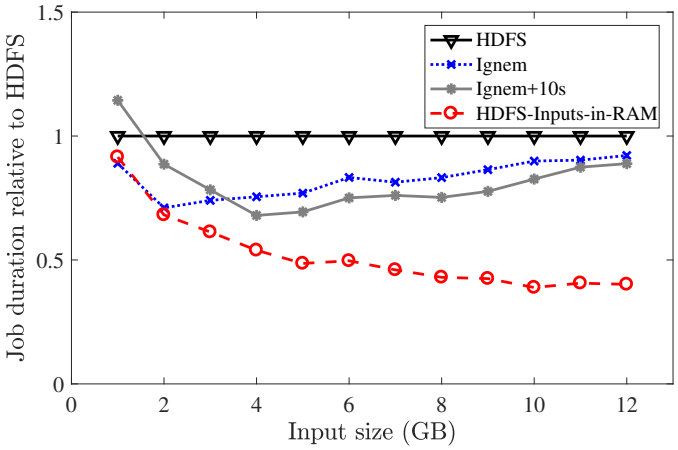


Fig. 8: Relative job durations for wordcount with different input sizes

5) Prioritizing smaller jobs helps Ignem perform well: Ignem slaves keep a queue of blocks waiting to be migrated. Instead of processing this queue in FIFO order, Ignem prioritizes migrating blocks for smaller jobs. When we disable prioritization, Ignem provides 2% less speedup w.r.t HDFS. This is nearly a 15% reduction in the benefit from Ignem in for this workload.

D. Sort workload

As shown in Table III, when its inputs are all in RAM the sort job runs nearly 2x faster. This highlights how important reads are, even for jobs that have significant computation and write a lot of data. While the buffer cache can absorb writes, reads will block on disk IO unless the data has been migrated into memory earlier. Though there is not enough lead-time to migrate the entire input into RAM, Ignem migrates part of it and reduces the job duration by 22%.

E. How the benefits of migration relate to input size and lead-time

Figure 8 shows the duration of the Wordcount with varying input sizes. When the data size is small the speedup from having inputs in RAM is smaller because reading makes up only a small part of the job. The speedup increases as reads become a larger part of the job. The speedup then plateaus when non-read activities start increasing at the same rate as the read time.

When the data size is small, Ignem can migrate the whole input into memory and matches the performance of *HDFS-Inputs-in-RAM*. Ignem keeps up until the input is too large to migrate within the lead-time. For our wordcount job, this occurs after 2GB. More generally the inflection point depends

on the disk bandwidth and how much lead-time there is. Beyond this point, the relative speed-up from migration starts to decrease. Ignem still migrates the same amount of data, but this becomes a smaller fraction of the total input. A migration scheme that can infer the Ignem speed-up curve for different jobs can potentially use this information to prioritize jobs which will benefit more.

F. The effects of lead-time. Introducing delay can speed up a job.

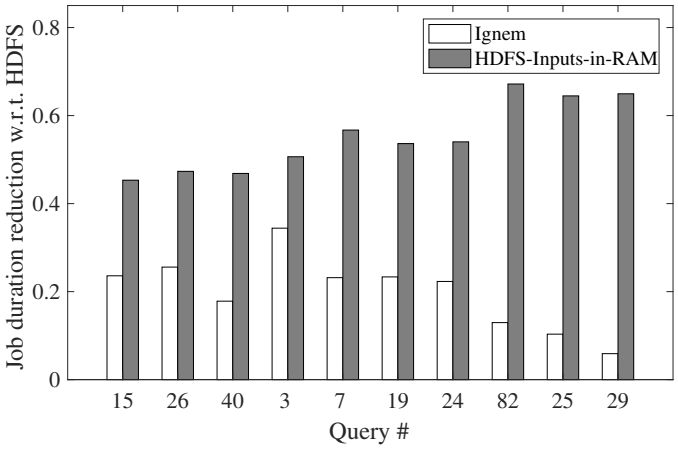
To determine how more lead-time would affect the speedup, we artificially insert some lead-time into the wordcount job. Before modification, the minimum lead-time for all blocks is 10s. We insert an additional 10s of lead-time into the jobs and plot these results in Figure 8 with the label *Ignem+10s*. We add lead-time by putting the wordcount job submitter to sleep just after the migration call, but before the job submitter finishes submitting the job. In a more natural setting, the extra lead-time may be from queueing delay at the scheduler. The sleep time is counted in the job duration.

When the input size is 1GB, Ignem+10s is 20% worse than HDFS due to the sleep period. However, at 2GB, the speedup from migration outweighs the extra 10s. Ignem+10s is now better than HDFS though not as fast as Ignem without the sleep. As the data size grows, Ignem+10s should be able to migrate more data because of the larger lead-time. This moves the minimum point in its speed-up curve further right. Surprisingly though, at 4GB, Ignem+10s outperforms Ignem. It seems counter-intuitive that adding time to a job reduces its duration. This happens because Ignem is more efficient at reading data from disk than the wordcount job itself. As described in Section III-A1, Ignem schedules migration so that only one block is read at a time, thereby avoiding high read-concurrency which degrades disk bandwidth. It is an intriguing concept that one can add delay to a system, perform work more efficiently during the delay and make up for more than the delay introduced, and so provide an overall speedup. This concept can be applied to other systems.

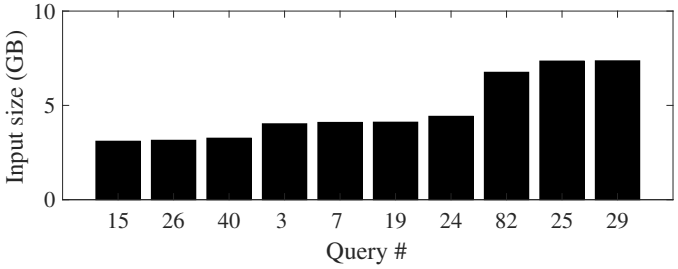
G. Hive

Finally, we augmented the Hive framework to use Ignem. In Figure 9a, Ignem improves query runtime by up to 34% for query #3, and by 20% on average. To accelerate all these queries, we only had to make a one-off modification in the Hive framework itself.

The gains from Ignem are less pronounced for queries 82, 25 & 29 in Figure 9a because these queries have a larger input size, as shown in Figure 9b. Eliminating disk reads has a larger impact when the input size is large but the amount of data that Ignem can migrate within the lead-time becomes a smaller fraction of the input size, which leads to a lower speed up. This is consistent with our discussion in Section IV-E. However, Ignem can still deliver significant speed up under these difficult circumstances.



(a) Query durations. Ignem accelerates most queries by more than 20%



(b) Query input size

Fig. 9: Hive query durations and their respective input sizes. Queries in both figures are sorted by input size.

V. Related Work

Pacman [5] leverages the all-or-nothing property of data-parallel workloads to implement a coordinated caching scheme that improves job runtime. The key insight is that a job is sped up only when the inputs of all tasks running in parallel are cached. Pacman works well for iterative workloads that read the same input repeatedly but cannot improve the performance of jobs that read singly-accessed blocks. This is because Pacman is purely a caching scheme and lacks any data migration capabilities. Nevertheless, the authors of Pacman acknowledge that 30% of all tasks in their workloads read singly-accessed blocks and Pacman cannot improve their performance. Ignem is complementary to Pacman in that it specifically targets performance improvements for singly-accessed blocks via migration.

A number of HDFS-based systems incorporate limited forms of data migration. HPMR [32] improves job performance by migrating a block from a remote rack to a server in the rack where the task processing the block is likely to execute. Thus, HMPR is complementary to Ignem. Triple-H [19] implements several data placement policies that distribute data over the tiers in a heterogeneous storage system composed of RAM, SSD, and HDD. The goal of the policies is to improve performance and load balancing. Triple-H also implements an eviction/promotion manager which evicts cold data from RAM and promotes hot data to RAM. Data is labeled hot or

cold based on past access counts. In contrast, Ignem tackles the problem of migrating cold data into RAM. Aqueduct [23] uses a control-theoretical approach to statistically guarantee a bound on the amount of impact on foreground work during a data migration, while still accomplishing the data migration in as short a time as possible. It does so by dynamically adjusting the speed of data migration guided by periodic measurements of the storage systems performance as perceived by the client applications. Aqueduct is complementary to Ignem. Aqueduct deals with how fast migration should proceed but assumes a migration plan is provided as input. In contrast, Ignem deals with creating such a migration plan.

A number of parallel file systems incorporate read-ahead prefetching, a very restricted form of data migration. GPFS [30], Lustre [31], Panache [14] as well as the Zebra Striped Network File System [18] perform prefetching for large files but only once the file has already been accessed sequentially. Recent work has shown that parallel file systems can also be made to serve the needs of data-parallel applications [6], [34] but no additional improvements have been proposed to the read-ahead prefetching scheme. HPMR [32] also performs prefetching after access but differs in that it starts the prefetch from the end of a block. In contrast to these solutions, Ignem migrates blocks *before* they are accessed making full use of the jobs' lead-time.

VI. Conclusion

This paper demonstrates that migrating cold data can deliver significant benefits for applications that use big data file systems. We first present analytic evidence that conditions in production clusters are favorable for migration. There is sufficient lead-time and residual bandwidth to migrate a significant amount of data before jobs start reading.

We then design and build Ignem to demonstrate that the potential speedup can be realized in practice. Ignem successfully migrates data and delivers large benefits across several workloads described in Section IV. Ignem reduces the average job duration in the SWIM workloads by 12%, and the average task duration by 38%. A sort job runs 20% faster under Ignem and wordcount jobs experience a speedup of up to 30%.

Beyond Ignem, our evaluation results demonstrate that the input stage of jobs is an attractive target for optimization. Ignem is only one point in a broad space that is yet to be explored thoroughly. Exploring different alternatives to some of our design decisions can produce further improvements. The experiments we ran with all inputs in RAM show that there is indeed room for further optimization.

Acknowledgement

We would like to thank the anonymous reviewers for their thoughtful feedback. This research was sponsored by the NSF under CNS-1422925 and CNS-1718980.

References

[1] Consumer Complaint Database. https://catalog.data.gov/dataset/consumer-complaint-database/resource/2f297213-7198-4be1-af1e-2d2623e7f6e9.

[2] SWIM Workloads repository. https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[3] The Virtual Memory Toucher. https://hoytech.com/vmtouch/.

[4] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *NSDI* (2013), vol. 13, pp. 185–198.

[5] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated memory caching for parallel jobs. In *Proc. NSDI 2012*.

[6] 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 *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09.

[7] Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 20.

[8] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review 23*, 1 (1995), 188–197.

[9] Chen, Y., Alspaugh, S., and Katz, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1802–1813.

[10] Chen, Y., Alspaugh, S., and Katz, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1802–1813.

[11] Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Cetintemel, U., and Zdonik, S. Tupleware: Redefining modern analytics. *arXiv preprint arXiv:1406.6667* (2014).

[12] Dean, J., and Ghemawat, S. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI* (2004).

[13] Dzinamarira, S., Dinu, F., and Ng, T. E. Pfimbi: Accelerating big data jobs through flow-controlled data replication. In *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on* (2016), IEEE, pp. 1–13.

[14] Eshel, M., Haskin, R. L., Hildebrand, D., Naik, M., Schmuck, F. B., and Tewari, R. Panache: A parallel file system cache for global file access. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010* (2010).

[15] Floratou, A., Patel, J. M., Shekita, E. J., and Tata, S. Column-oriented storage techniques for mapreduce. *Proceedings of the VLDB Endowment 4*, 7 (2011), 419–429.

[16] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03.

[17] Gkantsidis, C., Vytiniotis, D., Hodson, O., Narayanan, D., Dinu, F., and Rowstron, A. I. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI* (2013), vol. 13, pp. 2–5.

[18] Hartman, J. H., and Ousterhout, J. K. The zebra striped network file system. In *SOSP 93*.

[19] Islam, N. S., Lu, X., Wasi-ur Rahman, M., Shankar, D., and Panda, D. K. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on* (2015), IEEE, pp. 101–110.

[20] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–15.

[21] Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N., and Yuan, D. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *OSDI* (2016).

[22] Liu, Z., and Ng, T. E. Leaky buffer: A novel abstraction for relieving memory pressure from cluster data processing frameworks. *IEEE Transactions on Parallel and Distributed Systems 28*, 1 (2017), 128–140.

[23] Lu, C., Alvarez, G. A., and Wilkes, J. Aqueduct: Online data migration with performance guarantees. In *FAST* (2002).

[24] Nambiar, R. O., and Poess, M. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 1049–1058.

[25] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O. S., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012), pp. 1–15.

[26] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics framework. In *Proc. NSDI 2015*.

[27] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 36.

[28] REISS, C., WILKES, J., AND HELLERSTEIN, J. L. Google cluster-usage traces: format+ schema. *Google Inc., White Paper* (2011), 1–14.

[29] SAHA, B., SHAH, H., SETH, S., VIJAYARAGHAVAN, G., MURTHY, A., AND CURINO, C. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data* (2015), ACM, pp. 1357–1369.

[30] SCHMUCK, F., AND HASKIN, R. Gpfs: A shared-disk file system for large computing clusters. In *FAST 2002*.

[31] SCHWAN, P. Lustre: Building a file system for 1,000-node clusters. In *PROCEEDINGS OF THE LINUX SYMPOSIUM 2003*.

[32] SEO, S., JANG, I., WOO, K., KIM, I., KIM, J.-S., AND MAENG, S. HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–8.

[33] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–10.

[34] TANTISIRIROJ, W., SON, S. W., PATIL, S., LANG, S. J., GIBSON, G., AND ROSS, R. B. On the duality of data-intensive file system design: Reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11.

[35] THUSOO, A., SHAO, Z., ANTHONY, S., BORTHAKUR, D., JAIN, N., SEN SARMA, J., MURTHY, R., AND LIU, H. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 1013–1020.

[36] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[37] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12.

[38] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

[39] ZHANG, Z., LI, C., TAO, Y., YANG, R., TANG, H., AND XU, J. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. In *VLDB* (2014).