DYRS: Bandwidth-Aware Disk-to-Memory Migration of Cold Data in Big-Data File Systems

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

Rice University*, University of Sydney†

Abstract—**Migrating data into memory can significantly accelerate big-data applications by hiding low disk throughput. While prior work has mostly targeted caching frequently used data, the techniques employed do not benefit jobs that read cold data. For these jobs, the file system has to pro-actively migrate the inputs into memory. Successfully migrating cold inputs can result in a large speedup for many jobs, especially those that spend a significant part of their execution reading inputs.**

In this paper, we use data from the Google cluster trace to make the case that the conditions in production workloads are favorable for migration. We then design and implement DYRS, a framework for migrating cold data in big-data file systems. DYRS can adapt to match the available bandwidth on storage nodes, ensuring all nodes are fully utilized throughout the migration. In addition to balancing the load, DYRS optimizes the placement of each migration to maximize the number of successful migrations and eliminate stragglers at the end of a job.

We evaluate DYRS using several Hive queries, a trace-based workload from Facebook, and the Sort application. Our results show that DYRS successfully adapts to bandwidth heterogeneity and effectively migrates data. DYRS accelerates Hive queries by up to 48%, and by 36% on average. Jobs in a trace-based workload experience a speedup of 33% on average. The mapper tasks in this workload have an even greater speedup of 46%. DYRS accelerates sort jobs by up to 20%.

I. Introduction

A lot of big-data analytics applications spend a significant part of their execution in the input stage. This is primarily because of two factors. First, many jobs either filter or aggregate data before further processing, so the input stage typically reads a very large amount of data, while later stages often only process a much smaller fraction of that data. Second, there has been a series of improvements in computation frameworks that have accelerated the later stages of jobs, but they do not accelerate the input stage. Consequently, the input stage of jobs is accounting for an even larger share of jobs' run time. Unfortunately, common techniques for accelerating reads do so by keeping hot-data in memory, and do not benefit the many applications that read cold data. These applications have to incur the penalty of slow disk reads.

Prior work that analyzed production workloads from Facebook and Bing [3] reported that over 30% of tasks in these workloads read singly accessed data. These are likely recurring jobs that regularly process new data such as logs or new user-generated data like click-streams. Because this data is large and only processed periodically [27], it is persisted to disk and then has to be read cold when it is being processed. State of the art schemes to avoid disk reads do not benefit these applications. For example, Resilient Distributed Dataset (RDDs) [29] allow repeatedly accessed data to be pinned in memory. RDDs have resulted in an order of magnitude speedup for Spark over Hadoop for iterative jobs [30] but they do not accelerate reads to cold data. PACMan [3] optimizes the eviction of cached data that is already in memory but does not help speed up cold reads. Triple-H [14] analyzes the frequency and recency of accesses to data in a file system and moves popular data into faster storage to accelerate future reads. However, Triple-H only migrates hot data into memory and so it does not help with accesses to singly read cold data.

If we could accelerate reads to cold data, we expect to see a large speedup in many applications. One detailed study of SQL workloads on Hive [20] reported that queries spent about 19% of their duration blocked on IO, and 80% of that IO was spent reading data. Accelerating these reads by migrating the inputs into memory could result in a 15% speedup. When various optimizations that accelerate computation [17], [6], [11] are also applied, we expected the speedup from accelerating reads to be even larger since reading will be a proportionally larger part of the job. In the SQL study above, using C++ instead of Scala halved the CPU time. This would amplify the time spent on reads from 15% to 25%. Even for iterative jobs, accelerating the initial read can provide a significant speedup. Reading data from disk can cause the first iteration in Logistic Regression and K-Means to run 15x and 2.5x longer than later iterations respectively [29]. Reducing this initial slowdown would significantly speed up both applications.

For a more concrete comparison, we ran jobs from a trace workload from Facebook [5] with HDFS to determine the expected speedup of reading from memory. We first ran the workload with the input on disk, and then with the input pinned in RAM. At the application level, block reads from RAM were 160x faster on average than reads from the disk. We observed a large 10x speedup for map tasks that read from RAM, despite these tasks having other overheads unrelated to reads. SSDs can provide a speedup over disks, however, reads from RAM were still 7x faster than SSD.

Thus, pro-actively migrating job inputs upwards into memory has the potential to significantly accelerate those jobs that read cold data. In order to determine whether such migration is feasible in practice, we analyze the workload in the Google cluster trace. The feasibility of migration in practice depends on two conditions: 1) the system can identify cold data that will be accessed soon, and 2) there is sufficient time and residual bandwidth to migrate the data into memory before it is accessed. Our analysis suggests that both these conditions hold true, so effective migration is feasible.

However it is challenging to realize the potential of data migration in practice, particularly because the nodes in the cluster can be heterogeneous and the load on them is dynamic. The contribution of this work is the design and implementation of a migration system called DYRS that is bandwidth aware and so can adapt to heterogeneous and dynamic conditions. DYRS can adapt to match the available bandwidth on storage nodes, ensuring all nodes are fully utilized throughout the migration. In addition to balancing the load, DYRS optimizes the placement of each migration in order to maximize the number of successful migrations and eliminate stragglers at the end of a job. We implemented DYRS in a popular big-data file system and we show that is it both adaptive and effective at migrating data under dynamic conditions.

Specifically, we evaluate DYRS using several Hive queries, a trace-based workload from Facebook, and the Sort application. Our results show that DYRS successfully adapts to bandwidth heterogeneity and effectively migrates data. DYRS accelerates Hive queries by up to 48%, and by 36% on average. Jobs in the trace-based workload experience a speedup of 33% on average. The mapper tasks in this workload have an even greater speedup of 46%. Sort jobs are sped up by up to 20%.

II. Motivation

A. Which applications benefit most from migration?

Data migration is most beneficial for jobs whose initial stage is a significant part of the overall execution. This is true for many data analytics applications. The initial stage of these jobs often filters out or aggregates the input so later stages process much less data and are shorter. This data reduction amplifies the importance of reads and the potential speedup from migration.

Prior work on MapReduce workloads at Google shows ratios of up to 10:1 between map stage input and output sizes [7]. Similarly, Rhea [12] shows a reduction of 2-20000x between input and output sizes for Hadoop mappers. Other studies corroborate these findings [4], [5].

We ran several TPC-DS queries on Hive and examined the proportion of time they spent in the initial map phase. On average, the map tasks account for 97% of the total run time. 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 data analytics in general which makes such jobs good candidates for acceleration using data migration.

B. Effective migration must be heterogeneity-aware

To effectively migrate data, a migration framework should adapt to the amount of residual bandwidth on cluster nodes. This is particularly important since we observe significant heterogeneity among nodes in production clusters.

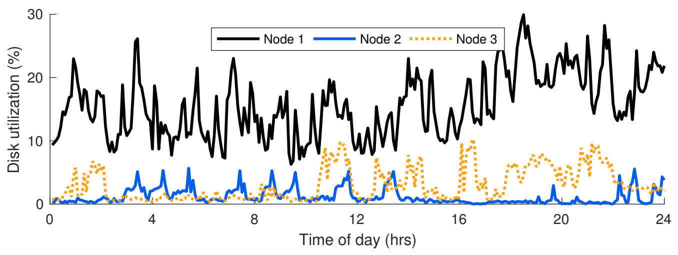


Fig. 1: Disk bandwidth utilization over a 24 hour period for three servers in the Google cluster. There is heterogeneity in the residual disk bandwidth across both nodes and time.

We analyzed the Google trace [22] and found that the disk utilization is highly heterogeneous and dynamic. The Google trace provides per-task IO time at a 5-minute granularity. We used this data to derive per-node disk utilization. We assume that each task performs IO at a constant rate. Thus, we can compute the per-second IO time for each task. The per-second disk utilization for a node is the sum of disk IO time for all tasks that are active on that node during that second. Lastly, we averaged the data to obtain per-node disk utilization at 5-minute granularity.

Figure 1 shows the disk utilization for three typical nodes in the Google trace. There is heterogeneity across both nodes and time. First, heterogeneity across nodes is seen from the fact that there are nodes that consistently have different amounts of residual disk bandwidth. For example, in Figure 1, node 1 is consistently busier than nodes 1 and 2. Its utilization is 13x and 5x that of nodes 2 and 3 on average. This can be caused by differences in the storage media or by an IO intensive application running on node 1 but not on nodes 2 and 3. Second, heterogeneity across time can be observed on each node but is more apparent on node 1. An efficient migration scheme must handle both types of heterogeneity.

C. Despite the challenge of heterogeneity, productions workloads have conditions favorable for migration

Effective migration also requires there to be enough time and residual bandwidth to read the data into memory before it is accessed. The nodes in the cluster must also have enough free memory to hold the buffered data. We again leverage the Google trace to ascertain that both these conditions are present. We analyze the time jobs spend on IO and logs of events like the submission, scheduling, and termination of jobs and tasks.

1) For most jobs, there is enough time to migrate a significant portion of the input into memory: We define lead-time as the amount of time between job submission and data access. This is the time during which migration needs to take place for it to be effective. Lead-time for a task is the time between job submission and task start time. Lead-time for a job is the time between job submission and the start of the first task in the job. It is common for tasks in a job to start at different times, therefore our definition of job lead-time is a lower bound.

Sources of lead-time: The two main sources of lead-time in big-data frameworks are queueing time and platform overheads. After job submission, a job's tasks are queued until

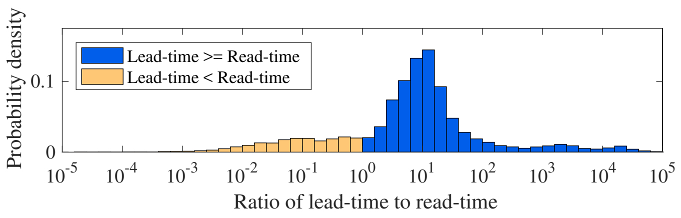


Fig. 2: 81% of jobs in the Google trace have enough lead-time to migrate the entire input into memory.

resources are available to launch them. This holds irrespective of the type of cluster scheduler: centralized [28], distributed [20] or hybrid [21]. Regardless of where tasks are queued, we can use the queueing time as lead-time to migrate job inputs into memory. In the Google trace, the mean lead-time for jobs is 8.8 seconds. This time can be used to migrate several hundred MBs of data per disk.

Platform overheads provide additional lead-time for migration. These include operations such as shipping binaries to workers and JVM warm-up costs [16] in some systems. The coordination overhead between workers and a centralized master via heartbeats may also increase lead-time [21].

Most jobs have a lot of lead-time relative to the amount of data they read: Jobs in the Google trace have a lead-time of 8.8 seconds on average. To determine whether this is sufficient for effective migration, we have to compare the lead-time to the time it would take to read the inputs into memory. For each job in the trace, we compute the time it takes to read input data into memory, which we call the read-time. Summing the time spent on IO by each task in a job gives us the read-time for the job in the trace. This simple sum likely overestimates the read-time because it includes time spent writing and does not consider that IO can be parallelized across multiple disks. Figure 2 shows a probability density function of the ratio between lead-time and read-time. Despite likely overestimating the read-time, 81% of jobs have a lead-time that is greater than the read-time, meaning there is sufficient time for migration. Even those jobs whose input is only partially migrated will be sped up by data migration.

2) Production clusters have enough residual bandwidth and free memory for cold data migration: **Servers in production environments have a lot of residual bandwidth that can be used for migration:** The amount of data that can be migrated within the lead-time depends on the amount of residual bandwidth in the cluster. The read-time we presented above may become larger if the disks are heavily utilized. Fortunately, analysis of the disk utilization in the Google trace shows that disks are often under-utilized.

Figure 3 shows a CDF of disk utilization samples from 40 servers in the Google cluster[22] over a 24h period. For 80% of these measurements, the utilization is under 4%. 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 provides ample free bandwidth for migration. Prior work on a small academic cluster also shows that disks are often under-utilized [9].

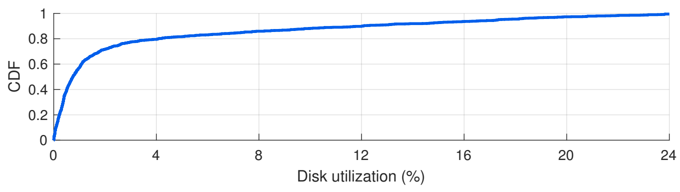


Fig. 3: CDF of disk bandwidth utilization over 24h for 40 servers in the Google workload. 80% of time utilization is under 4%. Thus, there is abundant residual disk bandwidth to use for data migration.

The working set for migration uses a relatively small amount of memory: In the Google trace, on average 10 tasks run on a server at a time. For a worst-case analysis, let's assume the latest generation dual-socket CPUs with 28 cores each, so the number of tasks on a server at a given time is unlikely to be greater than 56. Further, assume that all of the 56 tasks are mappers that each reads a large 256MB data block. Even under these conditions, 14.3GB of RAM is sufficient to hold the migrated data. 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 [3] draws similar conclusions that there is sufficient RAM to keep a significant portion of the working set in RAM.

III. Design

In this section, we walk through the decisions we made when designing DYRS. DYRS is structured in a master-slave architecture. The master is responsible for initiating migrations and deciding where these migrations will be performed. Slaves in DYRS are responsible for physically copying data into memory. When a client needs to have its input files migrated, it sends the list of filenames to the DYRS master. The master then maps the files to blocks in the file system and instructs slaves to migrate these blocks. DYRS schedules migrations using a First-In-First-Out (FIFO) policy. In future work, we plan to explore how alternative policies, and cooperation with the job scheduler, can improve performance. Once a block has been migrated, reads will be directed to the in-memory replica whether it is local or remote to the task making the read.

A. When and how to choose which replica to migrate

When the DYRS master receives a migration instruction, it constructs a list of blocks waiting to be migrated. We call these pending migrations. The master then assigns them to the slaves. We use the term binding to describe assigning a pending migration to a particular slave. Each slave also keeps a queue of migrations locally so the master can bind several blocks to the slave at once. All binding decisions are final.

1) When to choose the replica to migrate: DYRS delays the binding of migrations until as late as is possible without sacrificing performance. The delay allows the DYRS to gather the most up-to-date feedback about the performance of slave nodes and incorporate this information in order to make the best future binding decisions. In the extreme, we can bind new migrations to a slave node one at a time after the slave completes the previous migration. However, after each migration, the disk on the slave would be idle while the master is in the process of assigning it the next migration. In order to avoid disk idleness, DYRS ensures each slave has a few pending blocks queued locally. These queues should be deep enough to avoid idleness, and yet as shallow as possible to avoid binding migrations earlier than necessary. Slaves periodically query the master for more migrations whenever there is space in their local queues.

2) Selecting which replica to migrate: In DYRS, we choose to migrate only a single replica for each block regardless of how many replicas exist. This is because in most applications only a single task will read a block of data. Therefore, DYRS has to select which replica of a block to migrate. The previous section described when the choice is made; we now describe how it is made. When a slave queries the master for more work, we only bind a migration to the slave if we expect that the migration would complete soonest on it, compared to the other replica locations. Carefully selecting the replica to migrate increases the chance the data will be in memory by the time it is read. This optimization also helps avoid stragglers at the end of the migration.

Imagine if a very slow node is assigned one of the last migrations at the end of a file. That migration would likely still be in progress long after the faster nodes have completed all queued work and there is no more pending work at the master. It would have been better to leave the slow node idle and instead wait until a faster node is ready to process the migration. Assuming we can estimate how long a migration would take on each node Algorithm 1 shows how we estimate which node would complete a particular migration soonest. We then mark that node as the target for the migration. DYRS uses past migrations to estimate how long future migrations will take. Details are provided later in Section IV-A.

Algorithm 1 works as follows. We scan through the list of blocks and greedily set the target for each block as the node where assigning the block would result in the lowest new completion time. For each node, we keep a running estimate for the amount of time we expect the node to finish all the blocks that have been targeted to it. This algorithm is run regularly in a separate thread that is off the critical path of any coordination between the master and slave. When a slave queries the master for more work, the master searches its list of pending migrations for blocks that are targeted to that slave. If these exist, the master then assigns some of them to the slave. Guided by the reasoning in Section III-A1, we only assign enough migrations so that the slave does not go idle before the next time it queries for more work.

B. Ensuring efficient utilization of the disk

In Section III-A1 we described how it is necessary to keep some migration work queued on the slave in order to avoid disk under-utilization while the slave is querying the master for more work. The queue should be long enough such that it does not totally drain in the interval it takes to fetch more work. This ideal queue length can be computed by dividing the heartbeat interval by the time it takes to read a block of data using the maximum disk bandwidth.

```
Input: Estimated migration times and the number of queued
       blocks for each slave
Result: Mapping of each block to a target node
// initialize estimated finish times for each node
// assuming next pending block is assigned to this node
foreach node in DATANODES do
    finishTime[node] = migTime[node] $\times$ (numQueued[node]+1)
end
// set target for each block
foreach block in PENDING do
    locations = block.getReplicaLocations();
    target = locWithMinFinishTime(locations, finishTimes);
    block.migrationTarget = target;
    finishTime[target] = finishTime[target] + migTime[target]
end
```

Algorithm 1: Sets the target for each block as the node where its migration is expected to finish earliest.

Another way the disk can end up under-performing is if there are too many concurrent reads causing the disk to perform seeks too often. DYRS, therefore, serializes migrations and moves one block at a time into memory in order to limit disk read concurrency. Migrations are handled in FIFO order at the slaves. More sophisticated scheduling between applications can be implemented at the master. However, this is beyond the scope of this work.

C. Failure resilience

DYRS is highly resilient to failures. When there is a failure, DYRS reverts to the default behavior of the file system with no migration. The only adverse effect is the loss of the speedup from migration. Additionally, only active migrations are affected by failures. DYRS keeps only soft state so the system returns to normal quickly when DYRS' recovery mechanisms are run. The failure mechanisms in DYRS are similar to those in file systems with a master-slave architecture like HDFS [26].

1) DYRS master failure: If only the master process failed, we can restart it on the same server and it can immediately start receiving migration requests. If the server itself has failed, we have to launch the master on a new server and reroute migration requests to the new location. To reroute migration requests, the new master can broadcast an update to the clients configuration which stores the IP address of the master. Alternatively, we can maintain a live-backup of the master running and pre-list its address in the configuration file. Though the new master starts up with no state about which blocks are in memory at the slaves, its state eventually becomes consistent as slaves clean up their buffers as described in Section III-C3. The only adverse effect of the inconsistency is that even though some replicas are in memory, the master is unable to direct the block reads to them.

2) DYRS slave failure: When a slave process fails, all buffer space is reclaimed by the operating system and we can start a new process to handle new migrations. The new slave process should direct the master to drop state about blocks that were previously buffered on that server. This is because the master keeps track of where blocks are in memory so that reads can be directed to in memory replicas. The API for reading data from the worker is oblivious to whether the data is in memory or not so a block read can be served without error before the state has been dropped.

If the entire server has failed, then all data on that server will be unavailable. When a client queries DYRS for the in-memory replica of a block, DYRS only returns a choice amongst replicas on nodes that are available. A node is marked as unavailable when the file system misses several consecutive heartbeats from it. If a read occurs before the node is marked as unavailable the client can fail-over to one of the available replicas. HDFS handles DataNode failures in the same manner.

3) Job failure: For each migrated data block, the 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 and removed when the job sends an evict command to clear its ID from the blocks' reference list. If DYRS is working alongside a caching framework, the responsibility to call the evict command could be delegated to the caching framework. The evict command is handled through the DYRS master. A block is evicted from memory when its reference list is empty. If a job fails or is terminated before it issues the evict instruction, we need an alternative mechanism to clean the memory buffer and avoid memory-leaks. In DYRS, once the memory usage reaches a set threshold, the slave queries the cluster scheduler to check which jobs are active. It can then clear all inactive jobs from its blocks' reference lists and evict the blocks with empty lists from memory. This mechanism ensures DYRS keeps data in memory only for jobs that are still running. As a performance optimization to keep memory usage low, we also allow DYRS 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.

D. Scalability

DYRS has a master-slave architecture like that of HDFS. This architecture has been shown to scale to thousands of servers [26]. The DYRS master only has to 1) handle migration and eviction requests for files, 2) map the files to blocks, 3) compute the target replica for each block and, 4) respond to slave queries for migration work. The most computationally intensive task is updating the target for each pending block at the master. Fortunately, the update process is not on the critical path of the heartbeat messages between the master and slaves. During heartbeats, the master stores each slave's estimate of migration time and the number of blocks currently queued on the slave. In a separate thread, the master then uses these estimates to update the targets. The cluster administrator can control the rate of updates in order to limit their load. However, each update involves only a single pass through the list of pending migrations. Our prototype updates the targets for 50GB of pending migrations in under a millisecond.

Lastly, there is no coordination overhead between slaves as all messages are to and from the master. Each slave migrates blocks and computes its estimated migration time independently. Therefore, slaves are not a scalability bottleneck.

IV. Implementation

We have implemented DYRS within the Hadoop Distributed File System (HDFS). We were able to do so naturally because the master-slave architecture of DYRS matches that of HDFS. The DYRS master is implemented within the HDFS NameNode and the DYRS slave in the DataNode.

1) Migration mechanism at the slaves: DYRS slaves use the *mmap*, *mlock* and *munmap* system calls to migrate data from disk to the buffer cache. We favored this approach compared to migrating data onto a slave's heap for two reasons. First, the implementation is simpler and more restricted. Migration to the heap would require additional changes to the IO path for a task's reads. Second, data in the buffer cache can be accessed by multiple processes.

To start migration, *mmap* is used to map a file to the virtual address space of the DYRS slave process. Next, the *mlock* call reads the data into memory before returning. *Mlock*ed data will not be swapped to disk. Finally, the *munmap* call is used to unlock and unmap the file data and release memory back to the OS. Since the input is read-only, the OS need not write anything back to disk. The data can be simply discarded.

A. Estimating per block migration time

A critical part of the replica selection algorithm in DYRS is estimating how long migration will take on each node hosting a replica. We consider migration time to be the time it takes the *mlock* system call to return. We use an exponentially weighted moving average (EWMA) of past migration durations to minimize the effect of random fluctuations while giving more weight to recent migrations.

After a sudden drop in available disk bandwidth, the currently active migration may take a long time to complete. It is important to incorporate this signal into our estimate as soon as possible and not wait (potentially a long time) until migration completes. Thus, when the elapsed duration of an active migration becomes greater than its estimate, we update the estimate periodically (every heartbeat) until migration completes.

1) Memory management: Though DYRS pro-actively evicts data from memory as jobs finish or read the data, we also allow a hard limit to be set. When this limit is reached, migration commands are queued until buffer space is available or until they are discarded due to missed reads.

The reference lists described in Section III-C are realized as a hash-map that maps a job's ID to the list of blocks migrated for the job. This hash-map allows DYRS to efficiently locate the blocks that need to have their reference lists modified. When implicit eviction is being used, the slaves can extract the job ID directly from the read calls and do not need to contact the DYRS master. A job chooses whether or not to enable implicit eviction when the migration command is issued.

B. Running applications on DYRS

Applications run transparently on DYRS, no changes to the mapper or reducer code are necessary. We only made simple, non-application-specific changes to the framework.

In Hadoop, to make the most of the available lead-time, migration should be triggered as early as possible in the job's lifetime, ideally during job submission. Thus, we inserted the migration call in the job-submitter, the first element in a job's life cycle. Inside the job-submitter, we created an instance of the file system client (DFSClient). The DFSClient handles file operations (open, close, create, delete) and we extended it with a migration method. The arguments to this method are: a list of files, the operation to be performed (migration or eviction) and the type of eviction (explicit or implicit). The DFSClient communicates with the DYRS master via RPCs.

Frameworks like Hive submit a sequence of MapReduce jobs to complete a single query. Hive adds a query compilation phase before the job submissions. We inserted the migration call right after the query compilation. We leveraged the fact that Hive allows hooks to inject code at various steps in a query's life. We also had to map the Hive table names in the query to a list of HDFS file names. Similarly to Hadoop, the changes are done in the framework in a query-agnostic manner.

V. Evaluation

We have implemented DYRS within HDFS to evaluate its performance with real applications. We use several Hive queries, a multi-job workload derived from a Facebook trace and the Sort application. Before presenting experimental results, we first describe our hardware and software setup and the methodology we use to create heterogeneity.

A. Hardware setup and software configuration

Hardware setup - We use an 8 node cluster. One node hosts the HDFS NameNode and the Yarn Resource Manager. The other 7 nodes run HDFS DataNode processes and Yarn Node Managers. Each server has a 1TB HDD drive, 128GB of RAM and a Xeon E5-1650 CPU with 6 cores and 12 hyperthreads. There is a 10Gbps network between the servers.

Software setup - Our experiments compare DYRS against three other configurations of HDFS 2.7.3. The first two setups use default HDFS. In the first case, all inputs are stored on disk while in the second we use the vmtouch tool [1] to lock all input data in RAM. The second setup, which we call *HDFS-Inputs-in-RAM*, gives up an upper bound on the speedup we can expect. We only lock the initial input in RAM and do not modify the location of intermediate results nor the output. The third configuration we compare against is Ignem, a scheme that randomly chooses a replica of input data blocks to copy from disk to memory as soon as a job is submitted [8].

For all experiments, we flush the buffer cache before running our workloads to ensure the inputs are read from disk unless we have explicitly locked them in memory as in the HDFS-Inputs-in-RAM configuration. For the HDFS-Inputs-in-RAM setup, we still flush the buffer cache to ensure background syncing of the inputs to disk completes before we launch a workload. All our experiments are run on Apache Tez 0.9.0 coupled with Hadoop Yarn 2.7.3.

B. Workloads

We select three workloads to evaluate DYRS. We use Hive to study how DYRS handles complex, multi-job queries in isolation. The SWIM workload shows the behavior of DYRS in a complex and concurrent multi-job setting. Finally, Sort is a popular operation in many data transformation pipelines.

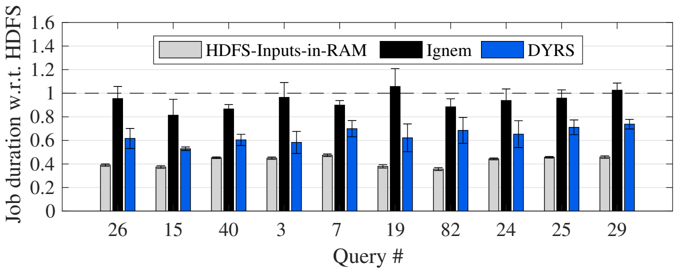
1) Hive queries: We use a set of ten queries from the TPC-DS[19] benchmark to evaluate DYRS on Hive 2.3.2. The TPC-DS dataset has more queries written in SQL, but we could only find ten that had been translated in HiveQL which is required to run them on Hive. We run each query independently on all four file system configurations. Hive queries are commonly used by data analysts to analyze large amounts of tabular data. By migrating data while a query is queued to run, a framework like DYRS improves the turn-around time for the analysis.

2) SWIM workload: The SWIM workload [5] is a trace-based workload derived from a production Hadoop cluster at Facebook. Jobs are sized (input, shuffle and output data size) and submitted according to the trace. We use the first 200 jobs in the trace. We scale down the job input sizes to fit on our 8-node cluster. The scaled cumulative job input size across all 200 jobs is 170GB. To have multiple jobs running concurrently we reduced job inter-arrival times by 75%. The distribution of job input sizes is heavy-tailed which is typical of production clusters [2]: 85% of jobs read little data (less than 64MB) but most of the data is read by a few large jobs (up to 24GB).

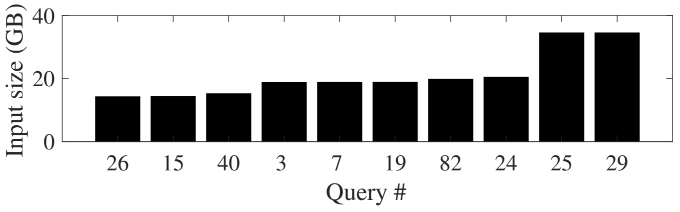
3) Sort job: We run sort across a range of different data sizes and with varying amounts of lead-time to study how the benefits of migration relate to the input size of a job and the available lead-time. We also use the sort application to understand and visualize the adaptive nature of DYRS.

C. Creating bandwidth heterogeneity

In production environments, the residual bandwidth available for migration is likely to be heterogeneous. This can be either due to fixed factors like different disk models on servers or due to dynamic factors like running applications. We introduce heterogeneity to evaluate if DYRS can effectively migrate cold data in a heterogeneous cluster. We reduce the residual bandwidth available on a node by running two Linux dd jobs that repeatedly read from two files on the disk. To ensure that dd reads come from the disk and not memory, we first flush the buffer cache and then use the "I_DIRECT" flag in dd to prevent buffering. For the set of experiments evaluating dynamic heterogeneity, we use a custom C++ application to generate different patterns of interference on one or two nodes.



(a) Query durations. DYRS accelerates most queries by more than 25%.



(b) Query input size.

Fig. 4: Hive query durations and their respective input sizes. Queries in both figures are sorted by input size.

	Absolute Duration (s)	Speedup w.r.t HDFS
HDFS	31.5	
HDFS-Inputs-in-RAM	16.9	46%
Ignem	66.4	-111%
DYRS	20.9	33%

TABLE I: Average job-duration and speedup across all jobs in the SWIM workload.

D. Hive query results

Figure 4a shows the durations of Hive queries normalized to default HDFS. HDFS-Inputs-in-RAM speeds up execution by 50% on average which shows Hive queries can benefit significantly from faster reads. Most queries perform aggregations or filtering early in their execution so the input stage is the dominant part of the queries. DYRS realizes the potential speedup and improves query runtime by up to 48% for query #15, and by 36% on average. While DYRS produces a large speedup, Ignem makes the queries run slower because its replica selection strategy does not avoid the slow node in the cluster. When the input size of the queries increases, the proportion of data that can be migrated within the lead-time decreases because the lead-time is constant. Despite this, DYRS provides over 25% speedup for the largest queries.

E. SWIM workload results

1) DYRS accelerates jobs of all sizes: We ran the SWIM workload to study how DYRS performs in a more realistic and complex setup. Table I shows the overall performance over the whole workload. DYRS accelerates jobs by 33% on average while Ignem results in a slowdown of more than 2x.

Figure 5 shows job duration binned by job input size. DYRS provides significant speedup across all job sizes: 34%, 47% and 26% for small, medium and large jobs respectively. Medium sized jobs have a bigger speedup than smaller ones because non-read overheads are a proportionally smaller part of their end-to-end runtime. For large jobs, a relatively smaller

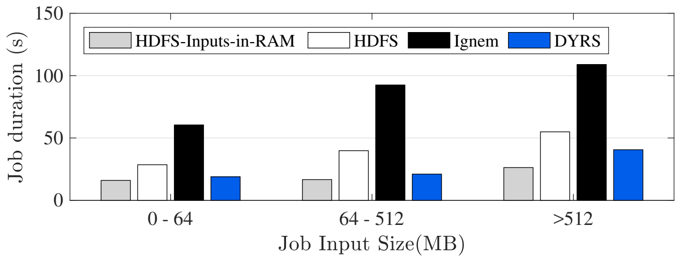


Fig. 5: DYRS speeds up small, medium and large jobs by 34%, 47% and 26% respectively. Across all job sizes, the average speedup is 33%.

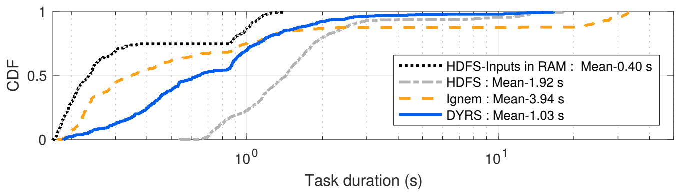
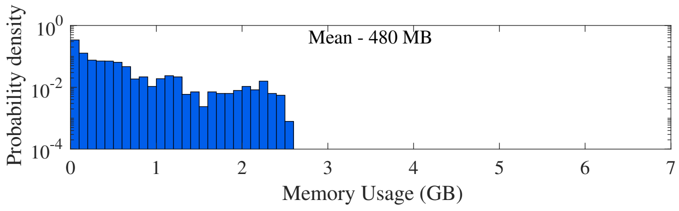


Fig. 6: DYRS significantly reduces map task duration.

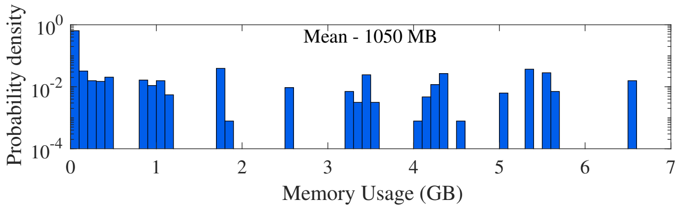
amount of the input can be migrated successfully compared to medium sized jobs, therefore, we see a smaller speedup. For small and medium-sized jobs, DYRS realizes over 75% of the potential speedup obtained by HDFS-Inputs-in-RAM.

2) Mapper tasks complete much faster under DYRS, improving cluster utilization: Looking at end-to-end job duration masks some of the benefits of DYRS since the end-to-end duration includes the shuffle and reduce phases that cannot be accelerated by migration. Figure 6 shows the speedup for mapper tasks in the SWIM workload. Mapper tasks run 1.8x faster under DYRS than with HDFS. This improves overall resource utilization as IO-bound mapper tasks spend less time holding CPU slots and memory. Ignem overloads the slow node while leaving the faster ones underutilized, which results in very short tasks on the fast nodes and very long ones on the slow node. DYRS, on the other hand, keeps all nodes well utilized. Though there are fewer very short tasks with DYRS, the average completion time is better.

3) DYRS obtains a large speedup while keeping a low memory footprint.: In this section, we estimate the amount of memory needed to achieve performance similar to HDFS-Inputs-in-RAM and compare this to DYRS. The performance of HDFS-Inputs-in-RAM can be achieved with minimal memory usage by a hypothetical scheme that migrates the input instantly when the job is submitted and evicts it when the job completes. Figure 7 shows the distribution for the amount of memory used on individual servers to store blocks migrated into memory for both DYRS and the hypothetical scheme. DYRS can only migrate 45% as much data as this hypothetical scheme but provides 72% of the speedup HDFS-Inputs-in-RAM provides in Table I. There is a diminishing return in speedup from using more memory because of the non-read parts of jobs. DYRS uses less memory because in reality there is limited bandwidth for migration, but also because DYRS pro-actively evicts data as soon as it has been read just as the hypothetical scheme does.



(a) Per server memory usage under DYRS.



(b) Per server memory usage of a hypothetical scheme that can migrate and evict data instantaneously and achieves the same performance as HDFS-Inputs-in-RAM.

Fig. 7: Memory usage in DYRS vs. a hypothetical scheme based on HDFS-Inputs-in-RAM. The y-axes are in log scale.

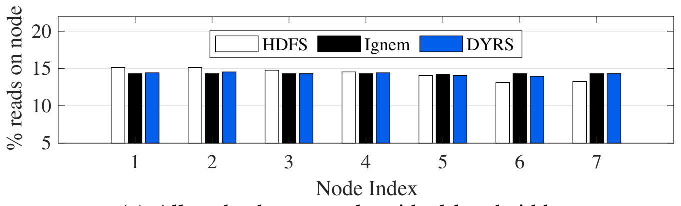
Interference Pattern	Figure	Sort job runtime (s)
Node #1 only: Persistently active	9a	137
Node #1 only: Alternates every 10s	9b	127
Node #1 only: Alternates every 20s	9c	129
Node #1 and #2: Alternates every 10s	9d	135
Node #1 and #2: Alternates every 20s	9e	137

TABLE II: DYRS effectively uses residual bandwidth regardless of the interference pattern so setups with the same overall amount of interference have similar runtimes. The row shading indicates setups with the same overall amount of interference.

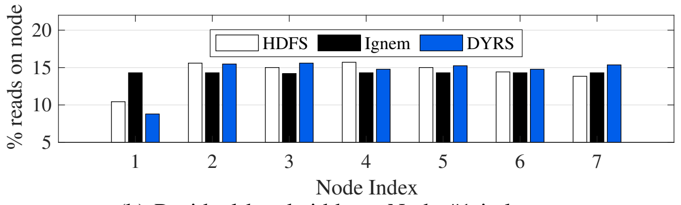
F. Sort job: Effective migration needs to be adaptive

1) DYRS adapts the number of migrations on each node to match the available bandwidth: In this section, we analyze why Ignem fails to provide speedup when there is a handicapped node in the cluster. We ran a Sort job and recorded the number of reads on each data node. Figure 8a shows the distribution of reads for a homogeneous cluster when we do not introduce any interference. As expected, each node receives a similar number of blocks. In Figure 8b one node is slowed down. Ignem still distributes the migration load equally. Ignem does not use historical data to guide its load distribution nor can it leverage the current node status because it binds migrations to replicas immediately upon receiving the migration command. DYRS, on the other hand, delays its binding. The completion of earlier migrations can, therefore, inform binding decisions for later ones. This delayed binding gives DYRS the ability to adapt to the residual bandwidth on each node. For default HDFS, tasks are placed on a node only when previous ones completed. This provides implicit feedback which results in fewer tasks on the slow node.

2) DYRS can quickly track and adapt to bandwidth changes: Heterogeneity in a cluster may arise from fixed factors such as hardware differences, or from the dynamics of the workload on the cluster. DYRS has to be adaptive to both types of heterogeneity. We now show how DYRS closely tracks the amount of residual bandwidth on each node. The

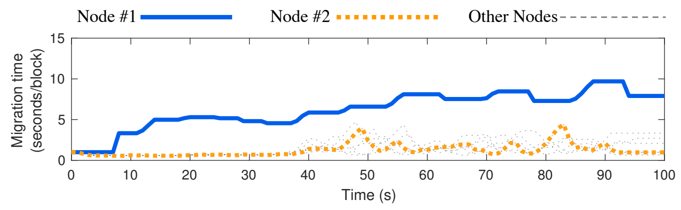


(a) All nodes have equal residual bandwidth.

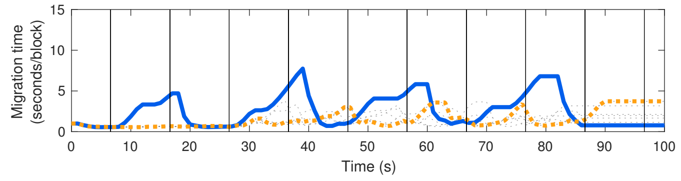


(b) Residual bandwidth on Node #1 is lower.

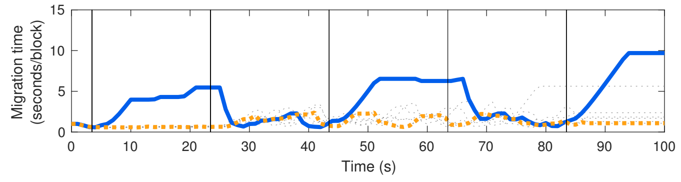
Fig. 8: Distribution of reads on DataNodes. DYRS and HDFS adapt to node heterogeneity, unlike Ignem which always balances the migration load equally.



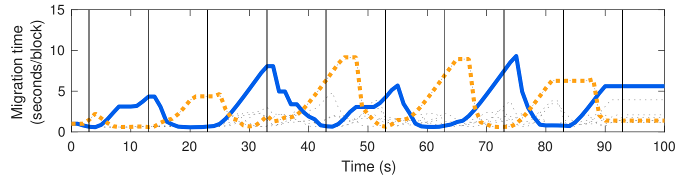
(a) Persistent interference on Node #1.



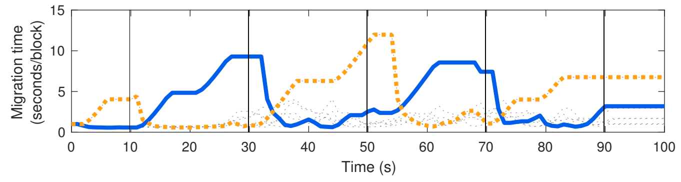
(b) Interference on Node #1 turned ON/OFF every 10 seconds.



(c) Interference on Node #1 turned ON/OFF every 20 seconds.



(d) Interference on Nodes #1 & #2 alternating every 10 seconds.



(e) Interference on Nodes #1 & #2 alternating every 20 seconds.

Fig. 9: Impact of interference on estimated migration time. DYRS adapts quickly.

metric we track is the estimated time to migrate a single block. Figure 9 shows the migration time estimate as we apply different patterns of interference. For easy comparison, we highlight the trendlines for two nodes in our cluster.

In Figure 9a, there is persistent interference on Node #1. DYRS' estimate for migration time is correctly higher on Node #1 than on Node #2. We expect the estimates to have some fluctuations because hard disk bandwidth varies a lot as tasks start and finish. In Figure 9b and Figure 9c we make the interference on Node #1 active/inactive every 10 and 20 seconds respectively. DYRS' estimate follows this pattern correctly. In an earlier prototype, we only updated the estimate upon the completion of a migration which resulted in a slow update after the residual bandwidth suddenly dropped. The extra update described in Section IV-A makes DYRS respond quicker to slowdowns. Though the frequency of alternation between active/inactive is different in Figure 9b and Figure 9c, there is interference only 50% of the time in both experiments so we should expect the runtimes of the sort jobs to be the same. The runtimes in Figures 9b and 9c should also be lower than that in Figure 9a since there is less interference. Table II shows these expected comparisons hold in practice.

For Figure 9d and Figure 9e we now introduce interference on both nodes #1 and #2, and cycle between active/inactive periodically. When interference is active on Node #1 it is inactive on Node #2 and vice versa. The figures show that DYRS correctly tracks the estimated block completion times. Each slave computes its own estimate independently so DYRS can track the migration time estimates for all nodes in the cluster with minimal overhead. Table II shows that the jobs in Figure 9d and Figure 9e have the same duration. These two also have a similar duration to Figure 9a. This is expected since all three experiments always have one node worth of interference at any time. In experiments with the same overall amount of interference, DYRS produces the same runtime because it is able to quickly adapt to changing load and fully utilize any residual bandwidth.

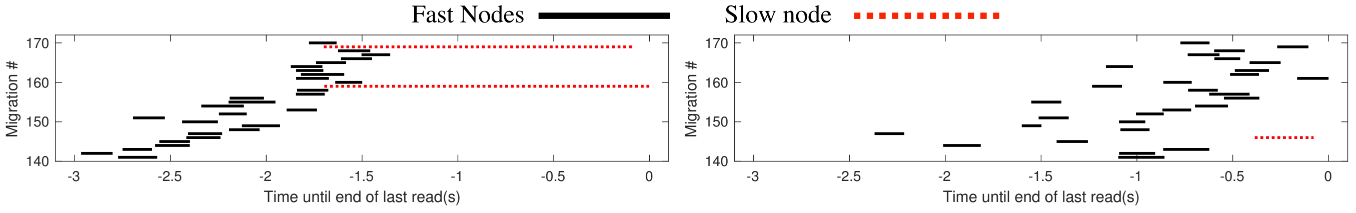
3) DYRS minimizes the risk of stragglers at the end migration: Figure 10 shows timelines for the last 30 blocks read in Sort job with 10GB of input. We compare DYRS against a naive load balancing scheme without DYRS' straggler avoidance. We mark time in reference to the last read so that we can easily visualize how eliminating stragglers would affect the makespan for reads. In DYRS, a node is only assigned a block if we expect that block to finish earliest on that node. We consider the time each node is expected to finish the block in question given the work that is either already queued on the node, or targeted towards it. Because of this, slow nodes only get assigned migrations when there are still a lot of outstanding migrations to keep faster nodes occupied. We can observe this in Figure 10b. In contrast, if a migration scheme were to simply assign migrations to any node with free slots in its local queue, some of the last few migrations can end up on a slow node as can be seen in Figure 10a. The last few migrations have a high risk of becoming stragglers, especially when they are assigned to a slow node.

4) Sort job: How migration is affected by input size and lead-time: In this section, we ran Sort jobs with various lead-times and input sizes to study how DYRS performs in these different settings. We report both the duration of the map phase and the end-to-end job duration which includes the lead-time.

Figure 11a shows that as we increase the data size but keep the lead-time constant, the relative speedup for the map-phase shrinks. This is expected because the amount of data we can migrate is mostly determined by the lead-time. The speedup from migration becomes less significant the longer the job runs. If we artificially introduce more lead-time, we can migrate more data and hence see a bigger speedup. However, additional lead-time could increase the end-to-end job duration if the speedup from more migrations does not offset the extra lead-time. Figure 11b shows that for shorter jobs, artificially inserting lead-time increases end-to-end job duration. However, for longer jobs, the end-to-end duration does not change despite the extra lead-time. DYRS effectively uses the lead-time to migrate data and the speedup from migration makes up for the additional lead-time. This improves overall utilization since mapper tasks spend less time holding resources like CPU slots and memory while blocked on reads.

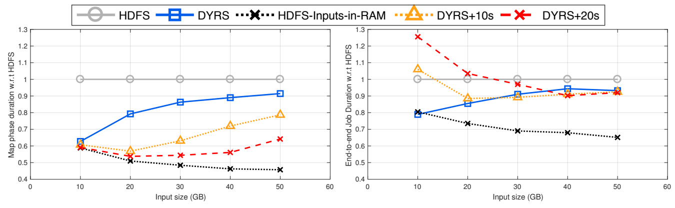
VI. Related Work

Ignem [8] is a disk-to-memory migration scheme that randomly chooses a replica of input data blocks to copy from disk to memory as soon as a job is submitted. This approach suits the case where the node bandwidths are homogeneous. Unfortunately, we observe significant bandwidth heterogeneity among nodes in production clusters. As Section V showed, Ignem could perform worse than default HDFS under heterogeneous bandwidth scenarios. In contrast, DYRS can adapt quickly to match the available bandwidth on storage nodes, ensuring all nodes are fully utilized throughout the migration, and in addition to balancing the load, DYRS optimizes the placement of each migration in order to maximize the number of successful migrations and eliminate stragglers at the end of a job. Pacman [3] is a caching scheme that coordinates caching across a distributed file system. Its coordination is based on the insight that jobs are only sped up when the inputs of all tasks in a wave are cached. Pacman only manages data that is already in memory so it does not improve the performance of cold reads. However, the authors of Pacman acknowledge that 30% of all tasks in their workloads read singly-accessed data and Pacman cannot improve their performance. DYRS fills this gap and complements Pacman by targeting cold reads and pro-actively migrating the data into memory. Triple-H [14] manages the placement of data within a tiered file system composed of RAM, SSD, and HDD. In addition to data placement when jobs write outputs, Triple-H monitors accesses to files and moves popular/hot files into fast storage and less popular ones to slower storage tiers. Triple-H promotes hot-data into memory to speed up future reads. DYRS, on the other hand, attempts to accelerate the initial reads to speed up jobs that read singly accessed data. HPMR [25] migrates data across the network to a server on the same



(a) Naive load balancing. (b) DYRS

Fig. 10: Blocks reads at the end of a Sort job. DYRS avoids stragglers by assigning the last few migrations to faster nodes.



(a) Map-phase duration. (b) End-to-end duration including lead-time.

Fig. 11: Sort application. Artificially inserting lead-time can improve utilization without hurting end-to-end job runtime.

rack a task will execute. DYRS would complement HPMR by migrating the data into memory. Aqueduct [18] is a system that controls the rate of background tasks like migration to limit their impact on foreground operations. When the cluster is busy, the techniques in Aqueduct can complement DYRS to control its effects on foreground tasks. GPFS [23], Lustre [24], Panache [10] as well as Zebra [13] perform prefetching for large files but only once the file has already been accessed sequentially. In contrast to these prefetching solutions, DYRS migrates blocks *before* they are accessed, making full use of the jobs' lead-time. Alluxio, formely Tachyon [15], allows users to manually load inputs into memory. However, it cannot perform load balancing or effectively select replicas to migrate. DYRS solves these problems and its API allows migration to fully exploit the lead-time.

VII. Conclusion

We have presented DYRS, a migration scheme for cold data that is bandwidth aware. DYRS' design incorporates methods that address the performance characteristics of production clusters. Experimentally, we have shown that despite bandwidth heterogeneity, DYRS accelerates Hive queries by up to 48%, and by 36% on average; jobs in a SWIM trace-based workload experience a speedup of 33% on average; sort jobs are sped up by up to 20%. These benefits are due to DYRS' ability to quickly adapt to changes in the amount of residual bandwidth on nodes and to optimize the placement of migrations to eliminate stragglers.

Acknowledgement

We would like to thank the anonymous reviewers for their thoughtful feedback. This research was sponsored by the NSF under CNS-1422925, CNS-1718980, CNS-1801884, and CNS-1815525.

References

[1] The Virtual Memory Toucher. https://hoytech.com/vmtouch/.
[2] Ananthanarayanan, G., et al. Effective straggler mitigation: Attack of the clones. In *NSDI 2013*.
[3] Ananthanarayanan, G., et al. PACMan: Coordinated memory caching for parallel jobs. In *NSDI 2012*.
[4] Appuswamy, R., et al. Scale-up vs scale-out for hadoop: Time to rethink? In *SoCC 2013*.
[5] Chen, Y., et al. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *VLDB 2012*.
[6] Crotty, A., et al. Tupleware: Redefining modern analytics. *arXiv preprint arXiv:1406.6667* (2014).
[7] Dean, J., and Ghemawat, S. Mapreduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*.
[8] Dzinamarira, S., Dinu, F., and Ng, T. E. Ignem: Upward migration of cold data in big data file systems. In *ICDCS 2018*.
[9] Dzinamarira, S., Dinu, F., and Ng, T. E. Pfimbi: Accelerating big data jobs through flow-controlled data replication. In *MSST 2016*.
[10] Eshel, M., et al. Panache: A parallel file system cache for global file access. In *Fast 2010* (2010).
[11] Floratou, A., et al. Column-oriented storage techniques for mapreduce. In *VLDB 2011*.
[12] Gkantsidis, C., et al. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI 2013*.
[13] Hartman, J. H., and Ousterhout, J. K. The zebra striped network file system. In *SOSP 93*.
[14] Islam, N. S., et al. Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *CCGrid 2015*.
[15] Li, H., et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC 2014*.
[16] Lion, D., et al. 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*.
[17] Liu, Z., and Ng, T. E. Leaky buffer: A novel abstraction for relieving memory pressure from cluster data processing frameworks. *IEEE Trans. on Parallel and Distributed Systems 28*, 1 (2017), 128–140.
[18] Lu, C., Alvarez, G. A., and Wilkes, J. Aqueduct: Online data migration with performance guarantees. In *FAST 2002*.
[19] Nambiar, R. O., et al. The making of tpc-ds. In *VLDB 2006*.
[20] Ousterhout, K., et al. Sparrow: distributed, low latency scheduling. In *SOSP 2013*.
[21] Rasley, J., et al. Efficient queue management for cluster scheduling. In *Eurosys 2016*.
[22] Reiss, C., et al. Google cluster-usage traces. *Google Inc.* (2011).
[23] Schmuck, F., and Haskin, R. Gpfs: A shared-disk file system for large computing clusters. In *FAST 2002*.
[24] Schwan, P. Lustre: Building a file system for 1,000-node clusters. In *PROCEEDINGS OF THE LINUX SYMPOSIUM 2003*.
[25] Seo, S., et al. HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In *CLUSTER 2009*.
[26] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop Distributed File System. In *MSST 2010*.
[27] Thusoo, A., et al. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD 2010*.
[28] Vavilapalli, V. K., et al. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC 2013*.
[29] Zaharia, M., et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 2012*.
[30] Zaharia, M., et al. Spark: cluster computing with working sets. In *HotCloud 2010*.