VeloxDFS: Streaming Access to Distributed Datasets to Reduce Disk Seeks

Sunghwan Ahn[1,2)] Hyeongjun Park[1)] V. A. Bolea Sanchez[3)] Deukyeon Hwang[5)] Wonbae Kim[4)]
Alan Sussman[6)] Beomseok Nam[1)]
SungKyunKwan University[1)] SK Hynix[2)] Kitware, Inc[3)] UNIST[4)]
University of Texas, Austin[5)] University of Maryland, College Park[6)]

***Abstract*—In this work, we design and implement *VeloxDFS*, a distributed file system for ETL MapReduce frameworks, which has compatible APIs with HDFS. VeloxDFS is a decentralized, consistent, hash-based file system that dynamically adjusts the size of partitioned blocks. Rather than the conventional static and coarse-grained partitioning scheme, VeloxDFS employs a fine-grained logical partitioning scheme and provides an abstraction of various sized blocks based on the I/O consumption rate. VeloxDFS avoids I/O contention and straggler problems by employing a block stream manager that coordinates the I/O requests of multiple tasks during runtime. By reducing the I/O contention of concurrent tasks, VeloxDFS enables *overcommit scheduling* that schedules a larger number of tasks than the available physical CPU cores, leveraging the OS scheduler to improve the computing resource utilization of a cluster. Our extensive performance study shows that VeloxDFS with the over-commit scheduling policy shows up to 1.7x higher job processing throughput than HDFS for multiple concurrent workloads.**

***Index Terms*—Distributed File System, Storage Management, IO Load Balancing**

I. Introduction

In the past decade, with the exponential data accumulation, Hadoop MapReduce and various big data processing frameworks, such as Spark, and Hive have been developed and improved. Hadoop and HDFS's significance as an ETL (extract, transform, and load) platform has grown in relation to the diversity of new frameworks. While significant efforts have been made to improve various data processing frameworks, not much effort has been made to improve the performance of the Hadoop Distributed File System (HDFS).

One of the well known limitations of ETL in distributed data processing frameworks running on top of HDFS is the load imbalance problem, often referred to as the *straggler* [1] or *skew* problem [2]. There are two common cases that cause the skew problem: (1) The assignment of uneven numbers of partitioned blocks in multi-tenant clusters, and (2) uneven computation times of partitioned blocks, i.e., records in some blocks require significantly longer times to process than others. For both skew types, HDFS is incapable of providing any method to help balance the load [2, 3, 4]. To mitigate the skew problem without the help from HDFS, several data processing frameworks proposed speculative execution [1, 2]. However, speculative execution is not effective in resolving the record level skew problem because applications make scheduling decisions based on the coarse-grained partitioned blocks in HDFS. That is, even if the speculative scheduling policy runs a straggling task in a different physical host, it will still require a longer computation time than others.

To resolve the record-level skew problem, we may reduce the size of tasks as was done by Monotasks[5], Riffle[6], and HDFS-InputSplit[4, 7, 8]. However, without modification of underlying file systems, such a fine-grained sub-tasks may introduce significant drawbacks. For example, small files require more disk seek operations than larger files, and a large number of small files not only place a significant managerial overhead on the Linux file system but also on the distributed directory service called NameNode in HDFS. Fine-grained tasks can be problematic in YARN (Yet Another Resource Negotiator)[9], a generic resource manager where the ApplicationMaster creates a container for each task. Hence, without overhauling the YARN resource manager, fine-grained tasks may suffer from high overhead incurred by initialization and authentication operations.

What complicates the skew problem and its data processing frameworks is that HDFS does not hide but exposes the details of partitioned blocks to data processing frameworks. This is because the Google File System [10], the ancestor of HDFS, was co-designed with the MapReduce [1] framework. A rationale behind such a tightly coupled design is the fact that "moving computation is cheaper than moving data". While the tightly coupled co-design helps minimize network congestion and improves the efficiency of the system, there is room for improvement by hiding the details of partitioning and providing an abstraction of partitioned blocks without compromising the co-design principle.

The root cause of the load imbalance is that HDFS provides equal-sized large blocks to data processing frameworks. However, with fixed-sized large blocks it is hard to achieve good load balance across tasks. If one task takes less time than others, it is better to increase the size of its block so that the task completion times can be balanced. However, HDFS does not dynamically adjust the block size. Moreover, when multiple tasks make requests for partitioned blocks, the I/O requests are not coordinated by HDFS instead they compete with each other for concurrent access to different blocks even if they belong to the same user file. Such concurrent accesses to different blocks in HDFS result in many disk seek operations. Such uncoordinated disk access becomes more serious in a multi-tenant cluster. i.e., applications make I/O scheduling decisions without considering the disk access

patterns of other concurrent applications.

In light of these observations, we design and implement an HDFS-compatible distributed file system - *VeloxDFS*, which hides the details of physical partitioning and provides a logical block abstraction to applications. With the logical block abstraction, load balancing is done by the distributed file system instead of by applications. VeloxDFS runs a disk access coordinator - *block stream manager* (BSM), which controls concurrent disk accesses by multiple clients. The BSM reduces disk seek operations by serializing concurrent disk accesses and balances the task completion times by dynamically changing the logical block sizes.

The main contributions of VeloxDFS are as follows.

- VeloxDFS logically partitions a user file into fine-grained small blocks but physically stores them in a single file. By storing partitioned blocks in a single physical file per node, VeloxDFS reduces the managerial overhead of the Linux file system metadata. Besides, by combining and storing partitioned blocks in a single physical file, VeloxDFS performs sequential I/O operations, reducing the number of disk seeks for multiple block requests of concurrent tasks.
- VeloxDFS implements *block stream manager* (BSM) that coordinates the concurrent I/O requests from multiple clients. The block stream manager removes the responsibility from applications of making load balancing decisions as the block stream manager balances the lifespan of tasks by adjusting the size of logical blocks based on I/O consumption rate.
- By reducing disk I/O contention, VeloxDFS enables DFS clients to benefit from *overcommit* scheduling in YARN. With overcommit scheduling each data node can run a larger number of concurrent jobs than the number of available cores, improving resource utilization and incurring minimal degradation of disk I/O throughput.
- From our performance study, we show that HDFS suffers from an unnecessarily large number of disk seek operations, which causes performance problems, especially when the YARN scheduler overcommits a larger number of jobs than the number of physical cores in data nodes. In contrast, VeloxDFS effectively coordinates concurrent I/O requests minimizing disk seek operations and takes advantage of overcommit scheduling to improve the computing resource utilization of the cluster.

The rest of the paper is organized as follows. In Section II, we present the shortcomings and challenges of HDFS. In Section III we present the design of VeloxDFS. In Section IV, we evaluate the performance of VeloxDFS against HDFS. Finally, we conclude the paper in Section VI.

II. Background: HDFS

HDFS is a distributed file system designed for distributed and parallel applications that process very large data sets that range from Gigabytes to Terabytes [11]. HDFS is designed for batch processing that need streaming access to datasets. To provide high throughput streaming access to files, HDFS relaxes POSIX requirements.

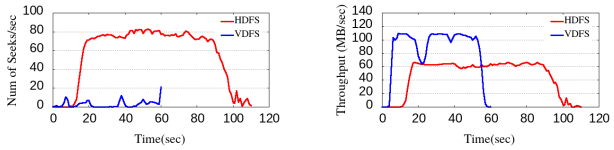
HDFS partitions a file into equal-sized blocks because the fixed block size makes the file management easy in a distributed cluster. To spend less time on disk seek operations but more time on data transfer, HDFS partitions an input file into large blocks of 128 MB by default. With such a large block size, disk seek overhead is expected to account for a small portion of disk access time.

However, HDFS, in its current form, is far from satisfactory in that it fails to benefit from sequential I/O due to random seeks between multiple blocks requested by concurrent tasks. Data processing frameworks on HDFS often start multiple tasks simultaneously and each task independently reads its own block, which results in disk seek operations and degrades the I/O throughput. In current HDFS design, such I/O contention problem is not considered. Consider an example where the YARN scheduler schedules eight map tasks to a data node. If there is only one core available in the node, each task will read its own block sequentially. However, if more than eight cores are available, all eight blocks will be accessed simultaneously. HDFS does not provide a way of controlling such concurrent disk accesses. Since all eight blocks are stored as separate physical files, concurrent accesses to these files require excessive disk seek operations. As the number of cores in data nodes increases, the disk I/O contention problem becomes more serious.

When a data node has multiple disks, HDFS distributes data blocks in a round-robin fashion so that it can benefit from parallel disk I/O. However, it is not common to have a larger number of disks than the number of cores in a data node. Therefore, multiple tasks often compete with each other to read their own blocks from the same disk, which results in disk seek operations and degrades the I/O throughput. As manycore systems become more prevalent, such disk I/O contention problem can become more and more serious. While a recent study has investigated the manycore scalability of Linux file systems [12], not much work has focused on the I/O contention problem of HDFS.

Another problem with the static block partitioning of HDFS is that some blocks require a significantly longer time to process than others. To make matters worse, the number of blocks processed by each worker node can be different. To mitigate such skew problems, various approaches [13, 1, 2] have been investigated in the past literature. However, to the best of our knowledge, none of the previous works looked into the problem in the distributed file systems layer. For best performance, HDFS needs to divide a file across data nodes in a way that parallel tasks take equal times to finish their work. However, it is almost impossible for HDFS to predict future I/O access patterns and proactively partition a file in a way that it can balance task completion times. Therefore, HDFS is designed to expose the partition information to its applications so that clients can make their own task scheduling decisions. However, coarse-grained partitions often fail to balance the load and also fail to leverage sequential disk accesses.

Figure 1 shows the disk seek frequency and I/O throughput of distributed file systems when we run a Hadoop `grep` job



(a) Disk Seek Frequency per Node (b) IO Throughput per Node

Fig. 1. *Disk Seeks Caused by Concurrent Tasks*

that processes a 100 GB input file on a 20 data node cluster that we describe in Section IV-A. Since sixteen tasks are simultaneously scheduled in each data node, they compete for the disk I/O and HDFS does not exhibit sequential disk access patterns, i.e., it performs about 80 disk seek operations per second. As a result, the disk throughput of HDFS is lower than 70 MB/sec, which is far below than the ideal sequential read performance of our testbed HDD (156MB/sec). Compared to HDFS, our VeloxDFS performs very few disk seek operations and achieves about 36.4% higher disk I/O throughput that HDFS (110 MB/sec vs. 70 MB/sec). We present the design of VeloxDFS and how it achieves a higher performance than HDFS in Section III.

III. VeloxDFS: Design and Implementation

VeloxDFS has many similarities with HDFS in that it targets applications that need streaming access to datasets. VeloxDFS is not a general purpose distributed file system but it is designed for the Hadoop ecosystem, which does not support POSIX but instead write-once-read-many semantics. As in HDFS, VeloxDFS partitions a large file into small blocks of a configured size and they are distributed over multiple nodes in a cluster. Although data blocks are distributed, VeloxDFS does not have a central metadata directory service (MDS) unlike HDFS that employs *NameNode*. Instead, VeloxDFS employs a consistent hashing as in DH-HDFS [14] for better scalability.

VeloxDFS contains three components - (i) a decentralized metadata directory service (MDS) that manages block partition information, (ii) a block stream manager (BSM) per node that controls disk access for clients, and (iii) the client library. In this section, we describe the design and implementation of these components.

A. Decentralized Metadata Directory Service

While HDFS employs a central NameNode that manages the namespace and file metadata, VeloxDFS manages the file metadata in a decentralized fashion. In VeloxDFS, each data node runs its own MDS, which forms a consistent hash ring structure. To enable $O(1)$ request routing on top of a consistent hash ring, each node keeps track of all data nodes and their key ranges as in Cassandra [15] and Dynamo [16].

As in HDFS, VeloxDFS partitions a user file into multiple blocks and distribute them across a set of data nodes. To manage the locations of partitioned blocks that belong to a single user file, VeloxDFS selects a data node using the hash key of a user file name and stores the metadata about the partitioned blocks, i.e., location (which data node has which block), file owner, permission, etc, in the MDS of the selected data node. When a client opens, closes, or renames a file, the client runs a hash function using the file name to determine which MDS has the file metadata. Once the client obtains the list of data nodes from the MDS, it communicates with the BSMs in those data nodes to read or write partitioned blocks.

B. Block Stream Manager

VeloxDFS partitions a user file into fine-grained small blocks, e.g., 8 MB by default, to mitigate the skew problem presented in Section II. However, there exist a few challenges that need to be addressed when we employ such fine-grained partitioning.

- (C1) First, one of the problems with fine-grained partitioning is that a large number of small blocks may result in a large number of concurrent small I/O requests, which will increase the number of expensive disk seek operations. Besides, if we store each small block in a separate file as in HDFS, the underlying Linux file system has to manage a large amount of file system metadata.
- (C2) Second, with fine-grained partitioning, a list of large number of small blocks need to be managed by the MDS. If we use 8 MB blocks rather than 128 MB, the amount of metadata managed by the MDS will be amplified by 16x, which will aggravate the scalability problem of distributed file systems.
- (C3) Third, fine-grained partitioning may result in a large number of tasks in YARN application frameworks such as MapReduce since they are designed to create a task for each partitioned block. A large number of tasks place a significant amount of scheduling overhead on the Application Master in YARN because each task requires a dedicated container, which performs expensive initialization and authentication [4].

To enable fine-grained partitioning while addressing these challenges, we design and implement the BSM in VeloxDFS.

To address *C1* and *C2*, VeloxDFS does not store each partitioned block in a separate file. Instead, it combines all the blocks assigned to the same data node and stores them in a single file, which we refer to as *block-combined file*. For example, suppose a 4 GB user file is partitioned into 8 MB blocks and distributed across four data nodes, such that each data node stores 128 blocks. Instead of creating 128 files per data node, each data node creates a single 1 GB block-combined file (128×8MB). In such a way, VeloxDFS decreases the amount of metadata in the file system.

When a client requests a particular block, the client acquires the file offset of the block from the BSM in addition to the file path. When multiple clients on the same data node request partitioned blocks of the same user file simultaneously, VeloxDFS creates a file input stream for each client, reads multiple blocks from the block-combined file sequentially, and provides blocks to each stream in a round robin fashion. We note that if there are multiple disks in a data node, VeloxDFS can distribute partitioned blocks across multiple block-combined files, i.e.,

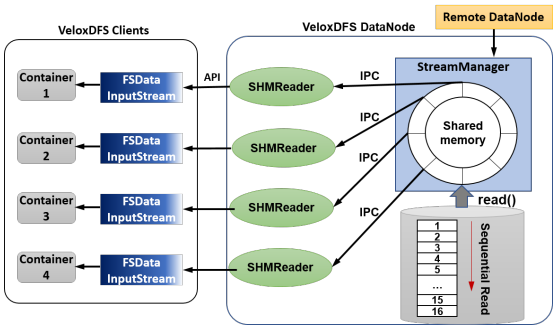


Fig. 2. Streaming Access to Disks for Concurrent Tasks in VeloxDFS

one file per disk, and evenly distribute partitioned blocks across disks to benefit from disk parallelism.

Figure 2 illustrates how VeloxDFS is different from HDFS when accessing partitioned blocks. When HDFS clients request blocks, each of them directly calls the `read()` system call through the HDFS API. However, VeloxDFS does not allow clients to directly access disks, since only a background BSM process accesses disks. Therefore, VeloxDFS clients read blocks from the shared memory provided by BSM. The BSM fills the shared memory buffer as it reads data blocks sequentially, and clients consume data blocks in the shared memory buffer as in the producer-consumer problem. If a client runs faster than others, it will fetch data blocks faster and consume more data blocks. As such, VeloxDFS resolves the load balancing issue across clients in a single data node and reads input files always in a sequential fashion even if multiple clients request I/O simultaneously.

To address *C3*, VeloxDFS employs logical blocks and avoids creating a large number of tasks/containers. Unlike HDFS, which assigns a physical block of 128 MB to each client, VeloxDFS creates and assigns a logical block of unknown size, i.e., a byte stream, to each client.

We note that the MapReduce AppMaster determines the number of concurrent tasks based on the number of physical blocks in data nodes. I.e., if AppMaster decides to process N blocks in a particular data node, it spawns N containers in the data node. If the number of available cores in the data node is smaller than the number of blocks, some tasks have to wait until previously scheduled tasks finish so that some cores become available. However, in VeloxDFS, the number of logical blocks is independent of the number of partitioned blocks and determined solely by the number of available cores. Since VeloxDFS partitions a user file into fine-grained blocks, the number of blocks is often much larger than the number of available cores. To prevent the YARN scheduler from creating as many clients as the number of partitioned blocks, VeloxDFS informs the YARN scheduler that it has just as many blocks as the number of available cores. With the logical block abstraction, the BSM can minimize the number of concurrent clients.

Consider the example shown in Figure 2. Although the data node has 16 blocks, only four client tasks will be scheduled by the Application Master since VeloxDFS tells the Application Master that it has four logical blocks. Then, the BSM starts reading the 16 blocks from disks and stores them in its shared memory buffer until it is full. The concurrent four tasks compete for the blocks in the shared memory buffer and consume them as needed. Using condition variables as in classic producer-consumer systems, the BSM finds out how many blocks have been consumed and reads more blocks from disks and store them in the buffer so that clients can keep reading. For multiple jobs that read different input files, VeloxDFS spawns multiple BSM processes and allocate the shared memory buffer space for each input file. We note that if we set the shared memory buffer size too small, clients can be stalled waiting for more blocks. If we set the buffer size too large, scarce memory space can be wasted. In Section IV, we show that VeloxDFS shows good performance with a small 100 MB buffer space for a BSM.

C. Block Distribution and Replication

VeloxDFS partitions a file and replicates partitioned blocks across multiple data nodes. One of the drawbacks of fine-grained partitioning is that the probability of a single record crossing the boundary of blocks increase. If the number of records that span multiple blocks stored in two different data nodes increases, remote I/O becomes necessary to obtain a complete record, which can decrease I/O throughput. Therefore, VeloxDFS stores a group of contiguous blocks in the same data node. In the example, we store two contiguous blocks in the same data node. In our implementation, we group sixteen contiguous 8 MB blocks and assign them to a single node so that each data node does not have block-combined files less than 128 MB.

To reliably store large files, VeloxDFS replicates partitioned blocks across multiple data nodes. In HDFS, the NameNode makes replication decisions for each block and manages the location information of replicas. As in HDFS, VeloxDFS may replicate individual blocks on random nodes considering rack locality. However, because replicating individual blocks will increase the metadata management overhead, we made VeloxDFS replicate the entire block-combined file across multiple data nodes. When VeloxDFS replicates a block-combined file, the locations of replicas need to be stored in its corresponding MDS. However, since VeloxDFS runs on top of the consistent hash ring overlay network, VeloxDFS replicates a block-combined file in the predecessor and successor nodes by default, such that the locations of replicated files do not need to be stored in MDS.

D. Recovery

In a large scale cluster, node failures are not uncommon. The replication strategy described above is rack-unaware and it can be used when nodes are in a single rack. VeloxDFS uses ZooKeeper to monitor node status, configuration, and I/O load of data nodes. If a data node fails, either its predecessor or successor takes over the faulty data node utilizing the replicated files and MDS. The take-over node will request

the faulty node's neighbor to create replica files to keep as many replica files as the specified replication factor. Such a take-over mechanism guarantees recoverability unless all three consecutive nodes fail at the same time.

If a cluster occupies multiple racks, VeloxDFS should employ rack-aware replication policies as in Cassandra, i.e., if a take-over node is in the same rack or same datacenter, we walk the ring until we reach the first node in another rack or datacenter. A drawback of such a network topology aware replication scheme is that each MDS has to store not just the location of one block-combined file but also replicated block-combined files. We note that the fault tolerance level that VeloxDFS guarantees is no different from that of Cassandra and Dynamo as all of them employ a consistent hashing overlay network.

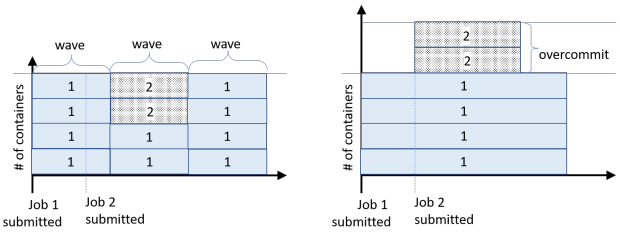
E. I/O Scheduling for Inter-node Balancing

Even if we assign an equal number of partitioned blocks to DFS clients, some blocks may require more computations than others and take longer times. To mitigate the skew problem, VeloxDFS employs two types of load balancing schemes, i.e., i) intra-node balancing and ii) inter-node balancing.

Intra-node Balancing: Suppose multiple tasks are processing data blocks of the same user input file on the same data node. In one task processes data blocks more slowly than others, it will fetch a fewer number of data blocks from the BSM. Therefore, intra-node balancing can be achieved across containers in the same data node.

Inter-node Balancing by Block Stealing: The YARN scheduler assumes that data nodes are homogeneous and tasks from the same job take the same time. However, YARN applications often suffer from *stragglers* due to various reasons [1, 2, 13]. To resolve this problem, VeloxDFS implements a *block stealing* policy. If tasks in a data node finish processing their blocks, its BSM communicates with the BSMs of its predecessor and successor nodes on its consistent hash ring. If it finds its predecessor or successor node is running slowly, the BSM steals unprocessed blocks from the slow node. This stealing policy makes fast nodes process a larger number of blocks than slow nodes in the cluster. Note that this stealing policy requires a locking mechanism to prevent multiple clients from processing the same block. For this purpose, MDS runs a distributed locking service using ZooKeeper. In particular, each BSM becomes a ZooKeeper client and receives the partition information from a corresponding MDS. Although the distributed lock is an expensive mechanism, the locking overhead does not place significant overhead on VeloxDFS because it steals blocks only when it finds a neighbor node is significantly running slowly. If all nodes make linear progress, MDS does not steal blocks and thus, does not acquire any lock from ZooKeeper.

If BSMs steal blocks only from successor and predecessor nodes, the block stealing policy does not cause remote disk I/O because VeloxDFS replicates block-combined files in a predecessor and a successor node. When a node steals blocks from neighbor nodes, it steals blocks in the reversed order, i.e., from the end of file, such that it can reduce the lock contention. We note that this stealing policy causes disk seek operations because the stealing reads blocks backward. To reduce the number of disks seek operations, we store blocks in replicated block-combined files in reversed order.



(a) *Fair Scheduler with HDFS* (b) *Overcommit Scheduler with VeloxDFS*

Fig. 3. Concurrent Job Scheduling

F. Multi-Wave vs. Overcommit Scheduling

A MapReduce job normally runs in multiple *waves* of map tasks and reduce tasks. If a job processes a user file partitioned into N blocks, the MapReduce Application Master has to make N scheduling decisions. If a larger number of tasks than the number of available cores are scheduled on a data node, only as many tasks as the number of available cores can run in parallel in the first map wave, and the other tasks must wait in the queue for subsequent waves.

The YARN scheduler supports three scheduling policies - FIFO, Capacity, and Fair scheduler. With these three policies, containers declare the amount of memory and CPU they need to perform tasks. If a task is assigned to an available CPU core, it is guaranteed that the task will not be preempted by other tasks waiting in the queue.

Since YARN scheduling policies are non-preemptive from the perspective of each task, the FIFO scheduler is not suitable for concurrent jobs because short jobs can starve while long jobs are running. To mitigate this problem, the Capacity scheduler allows multiple jobs to share a cluster by reserving a fraction of cluster resources for each job. However, the Capacity scheduler under-utilizes cluster resources unless there are a large number of concurrent jobs. The Fair scheduler makes scheduling decisions per wave. I.e., if a task finishes processing its block of 128 MB, the freed-up resources are given to another job's waiting task so that each job can get the same amount of computing resources.

Such block-based and non-preemptive scheduling policies allow each task to fully utilize its granted computing resources. However, a user job often over-estimates its memory requirement because if a container exceeds its memory allocation it is immediately terminated by YARN. Therefore, user tasks often do not fully utilize their granted resources. To take advantage of the fact that containers do not always make full use of granted resources, the overcommit scheduling policy has been investigated to improve resource utilization in a cluster [17].

Unlike HDFS, VeloxDFS dynamically changes the size of logical blocks so that a single wave of tasks consumes all the

partitioned blocks. However, a drawback of such single wave scheduling is that it does not take into account the balance of resource allocation across multiple jobs. I.e., if a long running job is submitted before a short job, the short job has to wait until the long running tasks finish.

To mitigate this problem, the size of a logical block can be limited, which ensures multiple waves. Alternatively, when concurrent jobs are waiting for a longer time than a threshold in the queue, we can make the BSM stop providing more blocks to current tasks so that they yield to waiting tasks. However, this raises a question about the threshold values. What is the optimal upper limit size for logical blocks and when each task yields to a waiting task? We do not have an answer to this question and this problem seems NP-hard. Instead of proposing another heuristic performance tuning algorithm, we propose to leverage the OS scheduler.

The YARN scheduler was designed to allocate resources to distributed and parallel applications, but it is not designed to leverage the operating system's CPU scheduling. That is, the number of cores has to be set by a YARN administrator and the general recommendation is to set it to the number of physical cores on the node [17]. With such settings, there is not much room for the operating systems to schedule concurrent tasks. If we run a larger number of tasks than the number of cores in a data node using HDFS, i.e., if we overcommit a data node with a large number of tasks, the system throughput degrades due to severe disk I/O contention. However, VeloxDFS determines the number of concurrent I/O requests not by the number of concurrent tasks but by the number of jobs that access different user input files. Therefore, the overhead of overcommit scheduling in VeloxDFS is much smaller than for HDFS.

Overcommit scheduling for VeloxDFS does not require modifications to the existing YARN scheduler. I.e., we use the Fair scheduler for overcommit scheduling. If there are C cores in a data node, VeloxDFS informs the YARN scheduler that it has $C \times overcommit\_factor$ virtual cores. However, VeloxDFS does not allow more than C tasks of the same job to run concurrently. VeloxDFS achieves this by telling the YARN scheduler that it has no more than C logical blocks for the same job. Consider the example shown in Figure 3(a). When job `1` is submitted, the NameNode in HDFS informs the Application Master how its input file is partitioned, distributed, and replicated. Suppose the Application Master decides to run 10 tasks in a data node based on the partition information. However, the data node has only four cores available. Hence, four containers will be created and the rest of the tasks must wait in the queue. While the first wave of tasks are being processed, another job `2` arrives. If Fair scheduling is employed, job `2` will occupy two cores after the first map wave finishes.

However, if overcommit scheduling is used, as shown in Figure 3(b), the MDS informs the Application Master that it has only four logical blocks although the data node has more than 4 blocks. Once the four tasks start, the BSM will provide all the partitioned blocks in the data node to the four tasks. When job `2` that requires a 256 MB input file arrives, the MDS will inform the Application Master that it has two logical blocks for the job. Although VeloxDFS employs fine-grained partitioning, the minimum logical block size each task requires is 128 MB unless the block is the last block of the user file. Since YARN considers $C \times (overcommit\_factor - 1)$ virtual cores to be idle, it will immediately schedule two tasks on the data node. That is, if we set the overcommit factor to k, at least k jobs can concurrently run and they will compete for C physical cores and perform context-switching according to the OS scheduling priority.

IV. Performance Evaluation

We evaluate the performance of VeloxDFS and compare it against HDFS 3.1.2 using Hadoop 2.7 and Spark 3.0.1. We implemented VeloxDFS in about 18,000 lines of C++ code and 6,200 lines of Java code. [1]

A. Experimental Setup

We run experiments on a 22-node Linux cluster. Each node has dual Intel Xeon Octa-core E5-2640 v3 processors (2.60GHz, hyper-threading enabled), 32 GB DDR4 ECC memory, and two 7200rpm 1 TB HDDs - one for OS (Ubuntu 18.04) and the other for distributed file systems. The maximum memory size for YARN is set to 27 GB. 22 nodes are connected via a single gigabit Ethernet switch. 20 nodes are used as data nodes, and the other two nodes are used for primary and secondary NameNodes for HDFS. Although VeloxDFS does not need the two NameNodes, it requires a ZooKeeper node to monitor the status of data nodes and to provide a distributed lock service for block stealing. We set the number of slots to 16 per data node except for overcommit scheduling, thus both HDFS and VeloxDFS can run 320 tasks concurrently. We set the replication factor to 3 for both HDFS and VeloxDFS and we use Fair scheduling unless stated otherwise.

B. Experimental Results

1) Performance with Varying Block Size: In the first set of experiments shown in Figure 4, we run a single `Grep` job that processes a 100 GB input file while we vary the size of the partitioned blocks. `Grep` is an I/O intensive workload. Since we evaluate the performance of distributed file systems, we first show the performance of an I/O intensive workload and then a computation-intensive workload later.

In HDFS, fine-grained partitioning results in a large number of map tasks, each of which reads its own partitioned input file without coordination. Therefore, HDFS fails to take advantage of sequential disk access. In addition, if we set the block size small in HDFS, it results in a large number of containers and the job suffers from container initialization and authentication overhead. As a result, the job processing time increases as we decrease the block size.

Unlike HDFS, VeloxDFS benefits from a BSM that coordinates concurrent I/O requests for a block-combined file.

[1] The code is available at https://github.com/DICL/VeloxDFS.

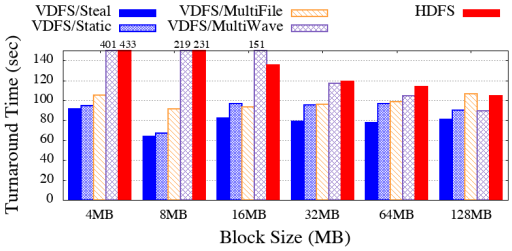
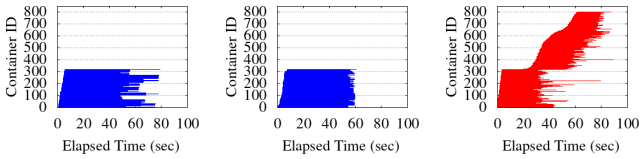


Fig. 4. Job Turnaround Time with Varying Block Size



(a) VeloxDFS (Static, 64MB Blocks) (b) VeloxDFS (Stealing, 64MB Blocks) (c) HDFS

Fig. 5. Containers Lifespan for a Single `Grep` Job

To quantify the performance effect of each VeloxDFS design choice, we implemented four variants of VeloxDFS.

In the first variant, denoted as `VDFS/MultiFile`, we store each partitioned block in a separate physical file as in HDFS. In this implementation, a BSM spawns multiple threads and each thread does a `read()` system call for each partitioned block file independently from each other. As a result, this variant incurs a large number of disk seek operations. We note that this is similar to how HDFS is processing I/O requests. However, unlike HDFS, `VDFS/MultiFile` does not generate multiple map waves but a single container consumes multiple blocks. In the experiments, `VDFS/MultiFile` shows similar performance to HDFS when the block size is 128 MB. As the block size decreases, HDFS suffers from a larger number of containers and the job turnaround time increases sharply whereas `VDFS/MultiFile` does not increase the number of containers and the job turnaround time slightly decreases due to better load balancing with a finer-grained partitioning. We note that the number of disk seek operations is rather insensitive to the block size but depends on the number of concurrent tasks that issue disk I/O requests.

Another variant of VeloxDFS, denoted as `VDFS/Multi-Wave`, stores all partitioned blocks in a single block-combined file, but enables multiple map waves by limiting the maximum size of logical blocks. That is, even if there exist some blocks that are not processed in the shared memory buffer, the BSM stops feeding a task if the task has processed a predefined number of bytes. By enforcing such a limit on logical block size, VeloxDFS can enable multiple map waves as in HDFS. In the experiments shown in Figure 4, the number of map waves increases from 3 to 81 as we decrease the block size from 128 MB to 4 MB. As a result, the job turnaround time of `VDFS/MultiWave` suffers from the overhead of a large number of containers.

In `VDFS/Static` and `VDFS/Steal`, we store all partitioned blocks in a single block-combined file and disable multiple waves so that only a single wave of map tasks processes all partitioned blocks. We enable and disable the block stealing policy in `VDFS/Steal` and `VDFS/Static`, respectively. Figure 4 shows that both `VDFS/Static` and `VDFS/Steal` improve the job response times because small block sizes help improve load balancing and a single map wave reduces the container overhead. Since `VDFS/Steal` balances the task lifespans more aggressively by stealing some blocks from remote data nodes, it improves the job turnaround time by up to 19.8% compared to `VDFS/Static`. We note that the disk seek frequency and I/O throughput of `VDFS/Steal` and HDFS for these experiments is shown in Figure 1. Since VeloxDFS shows the best performance with 8 MB blocks, which is 39% faster than the best performance of HDFS with 128 MB blocks, we set the partitioned block size of VeloxDFS to 8 MB for the rest of experiments unless stated otherwise.

2) Load Balancing: Stealing Policy: In the experiments shown in Figure 5, we measured the lifespan of tasks (YARN containers) when we submit a `Grep` job that processes an input file of 100 GB. In HDFS, 800 containers are scheduled because the 100 GB input file is partitioned into 800 blocks. However, the MapReduce Application Master creates only 320 containers for VeloxDFS since there are 320 physical cores in the cluster. In the experiments, we set the block size to 64 MB although it is not the best configuration for VeloxDFS. We use such a coarse-grained partitioning to emphasize the effect of the block stealing policy. With the 64 MB block size, the 100 GB input file is partitioned into 1600 blocks and each data node in the 20 node cluster stores 80 blocks. Since there are 16 cores per node, each container processes 5 blocks, i.e., 320 MB, if we do not enable the block stealing policy.

Figure 5(a) shows that there exist some tasks that take a longer time than others. If the block stealing policy is enabled, faster tasks steal blocks from slow tasks, which helps improve the balance of task lifespans, as shown in Figure 5(b). As a result, all the tasks with VeloxDFS that enables the block stealing policy complete in 59 seconds whereas there are some tasks that take longer than 70 seconds if we do not enable the block stealing policy.

Figure 5(c) shows that a MapReduce job with HDFS processes tasks in multiple waves, i.e., 320 tasks are scheduled in the first wave, another 320 tasks are scheduled in the second wave, and then the remaining 160 tasks are scheduled in the third wave. We note that most of the tasks in the first map wave complete in about 35 seconds. However, a few tasks in the first map wave take longer than 50 seconds and there exists one task that takes longer than 80 seconds. Due to the unbalanced lifespans of tasks and multiple waves, the job turnaround time of HDFS is 98 seconds, which is about 29% slower than VeloxDFS with the block stealing policy enabled.

C. Concurrent Execution: Overcommit Scheduling

Since VeloxDFS does not require multiple waves of tasks, different job scheduling policies such as overcommit scheduling need to be investigated for fairness when multiple jobs share the cluster. In the experiments shown in Figure 6, we

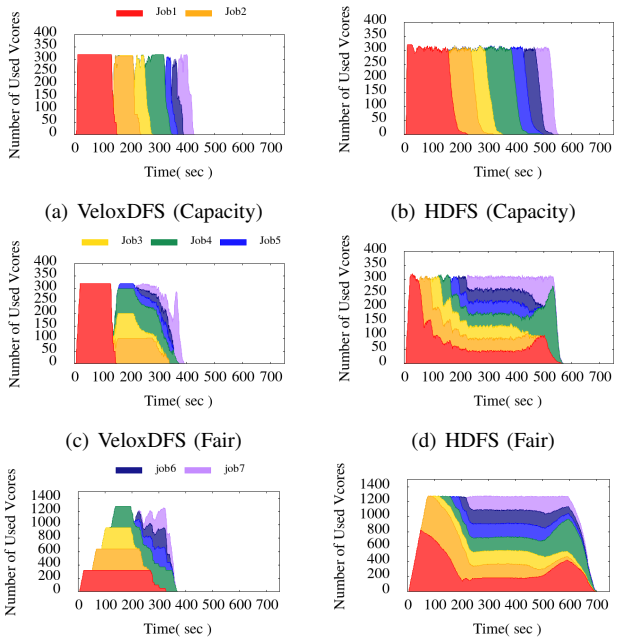


Fig. 6. *Lifespan of Containers for Concurrent Workload (Job1:Grep 200GB submitted at 0 sec, Job2:Grep 100GB at 50 sec, Job3: WordCount 50GB at 90 sec, Job4: WordCount 100GB at 120 sec, Job5: AggregatedWordCount 50GB at 160 sec, Job6: Grep 50GB at 190 sec, Job7: WordCount 50GB at 220 sec.)*

submit seven MapReduce applications. Their job submission times and input file sizes are shown in the caption. We note that each of these jobs accesses a different input file.

When Capacity scheduling is used, which is designed to maximize the throughput and the utilization of the cluster, both HDFS and VeloxDFS dedicate the entire cluster to each job being executed in the order they arrive. Due to the superior performance of VeloxDFS, the batch completion time of VeloxDFS is 23% lower than HDFS (429 sec vs. 554 sec).

If the Fair scheduling policy is used, HDFS that enables multiple waves balances the cluster resource utilization for all running applications. That is, all jobs get started after waiting for only one single wave of the previously scheduled tasks. However, VeloxDFS does not allow multiple waves unless we limit the logical block size. As a result, the jobs that arrive while the first long running job is being executed wait until the first job completes. When the first job completes, the scheduler finds four jobs are waiting and they start sharing the cluster according to the Fair scheduling policy. In the experiments, two more jobs arrive while the four jobs are running. These two jobs get started when the previously scheduled jobs release CPU cores.

It is noteworthy that more than half of cores become idle for a while after the first job completes when we employ the fair scheduling policy. This is not a problem of VeloxDFS but a scheduling overhead that comes from the YARN scheduler. That is, the Fair scheduler makes job scheduling decisions based on the progress of multiple map waves. However, since VeloxDFS does not have multiple map waves, the scheduler

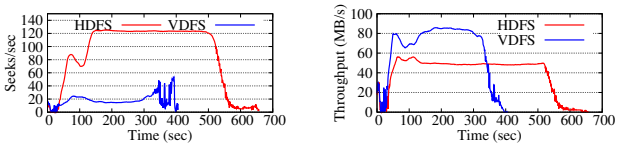


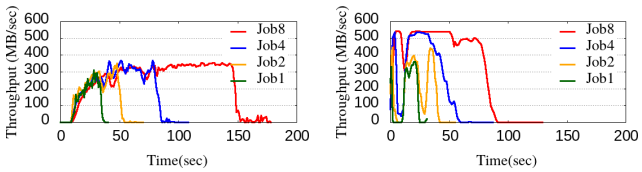
Fig. 7. *Spark Performance on HDFS and VeloxDFS*

cannot predict when the containers will release CPU cores. Therefore, the job scheduler cannot pre-allocate resources for waiting jobs. Currently we do not have a solution to this problem because it is a YARN scheduler issue. Hence, we leave this as future work.

In the experiments shown in Figure 6 (c) and (f), we allow a 4x larger number of tasks (1280) than the number of physical cores (320) to run in the cluster, i.e., we set the overcommit factor to 4. Since the first job processes a 200 GB input file, HDFS will inform the YARN scheduler that it has 1600 blocks (200 GB/128 MB = 1600). With the overcommit scheduling policy, the YARN scheduler keeps scheduling containers even after the first 320 containers are scheduled as shown in Figure 6 (f). It is also noteworthy that the Fair scheduling policy has a very high container scheduling latency and it fails to schedule 1280 tasks for the first 50 seconds. At 50 seconds, when not all 1600 but only about 800 tasks of the first job are scheduled, the second job arrives and the Fair scheduler assigns the remaining 480 slots to the second job. We note that the overcommit scheduling policy is no different from the Fair scheduling policy except that the overcommit scheduling increases the number of virtual cores. When the third job arrives at 90 seconds, it finds out that there are no available slots and it waits until previously scheduled tasks release CPU resources. Although we set the overcommit factor to 4, the legacy HDFS does not allow four jobs to run in a fair way but the first job overuses CPU resources.

We note that the lifespans of tasks with the overcommit scheduling policy with HDFS are much longer than with Capacity or Fair scheduling policy because of disk I/O contention and container overhead. If we use the Capacity or Fair scheduling policy, all seven jobs complete in 550 or 570 seconds, respectively. However, if overcommit scheduling is used, all seven jobs finish in 700 seconds, which is about 27% higher than with the Capacity scheduling policy.

In contrast, if we use the overcommit scheduling policy with VeloxDFS, the batch completion time is even shorter than Capacity and Fair scheduling policies. VeloxDFS does not create a larger number of logical blocks than the number of physical cores, although we set the number of available cores per data node (*vcore*) to a multiple of the number of physical cores. Therefore, Figure 6 (c) shows that, when a single job is running, the overcommit scheduler runs exactly the same number of tasks as the number of physical cores, as in Capacity scheduling. Later, when the second job is submitted, the YARN scheduler creates another 320 containers since it



(a) HDFS (Fair) (b) VeloxDFS (Overcommit)

Fig. 8. *Hadoop MR: AVG. I/O Throughput per Node (SSD)*

believes data nodes have a 4x larger number of CPU cores. Compared to Capacity scheduling, the overcommitted tasks take a longer time to complete, i.e., the second `Grep` job takes 185 seconds since it competes with three other concurrent jobs, but it takes 78 seconds in Figure 6 (a) when the entire cluster is assigned to the job. This slowdown is because of the context-switching overhead, but it should be noted that the job execution time is not quadrupled even if four jobs are concurrently running. This is because MR tasks alternate between CPU burst and I/O burst cycles and the operating system's CPU scheduler appropriately schedules concurrent tasks to fully utilize CPU availability.

1) Throughput of Concurrent Jobs with Spark: In the experiments shown in Figure 7(a), we compare the performance of VeloxDFS against HDFS using Spark. We launch Spark workers on 18 data nodes, and submit 8 `WordCount` jobs, each reading a different 50 GB input file. We let each job use one executor per data node, and the number of cores assigned to each executor is set to 8. To enable overcommit scheduling, we set the number of `vcores` of the data node to 64, i.e., the overcommit factor is set to 3.2.

Figure 7(a) shows that VeloxDFS performs fewer than 20 seek operations per second during most of the execution time while HDFS performs about 120. As a result, the peak I/O throughput of HDFS in each data node is only 55 MB/sec whereas that of VeloxDFS is 83 MB/sec, as shown in Figure 7(b). As a result, all eight jobs finish in 400 and 650 seconds with VeloxDFS and HDFS, respectively.

2) Throughput of Concurrent Jobs with SSDs: In the experiments shown in Figure 8, we evaluate the performance of VeloxDFS on a small 11 node cluster that has SSDs as we vary the number of concurrent Hadoop `Grep` jobs that read their own 50 GB input files. This second testbed cluster has dual Intel Xeon 10 core Gold 5115 processors (2.4 GHz), 64 GB DDR4 ECC memory, a 7200rpm 2TB HDD for OS (Ubuntu 18.04) and a 512 GB Samsung Evo 860 SSD for the distributed file system. The 11 nodes are connected via a gigabit Ethernet switch and 10 nodes are used as data nodes.

Although the overhead of reading non-contiguous blocks on an SSD is not as high as for an HDD, VeloxDFS still outperforms HDFS because sequential reads are faster than random reads even for SSDs. Besides, VeloxDFS shows superior load balancing behavior. When we run a single job, VeloxDFS performs almost no random read operations. But as we increase the number of concurrent jobs, it performs more disk seek operations because the number of BSMs increases. As a result, when we submit 2, 4, and 8 concurrent jobs, VeloxDFS performs up to 259, 367, and 486 random read operations per second, respectively. Although these numbers are significantly larger than the number of seeks for the single job workload, HDFS performs a much larger number of disk seek operations for the same workload, i.e., 741, 814, and 762 disk seek operations. It is noteworthy that HDFS performs more than 400 disk seek operations even when we run only two concurrent jobs. As a result, the peak I/O throughput of HDFS in each data node is only 310, 350, 368, and 354 MB/sec whereas that of VeloxDFS is 362, 443, 537, and 541 MB/sec when the number of concurrent jobs is 1, 2, 4, and 8, respectively. Note that the maximum sequential read throughput of our testbed SSD is 550 MB/sec.

V. Related Work

It has been reported that the NameNode in HDFS is often flooded by a huge number of file metadata requests [18]. To mitigate this problem, the centralized NameNode design has been revisited in several previous studies including HopsFS [19], Flat Datacenter Storage [20], DH-HDFS [14], and EclipseMR [21]. In addition to scalability, there have been efforts to improve the fault tolerance of HDFS such as HARDFS [22], Salus [23], and DH-HDFS [14]. The architecture of VeloxDFS is similar to those of DH-HDFS and EclipseMR [21] in that they employ the distributed hash tables (DHT). However, the main focus of VeloxDFS is not on the scalability and fault tolerance, but on the excessive disk seek operations and the skew problem.

Monotasks [5] is an execution model that decomposes a job into *monotasks*. A monotask is a small unit of work that is allowed to use only a single resource, i.e., CPU, disk, or network. In Monotasks, each resource has a dedicated scheduler, which schedules monotasks for that particular resource. The disk scheduler in Monotasks is similar to VeloxDFS in that it tries to reduce disk I/O contention. However, Monotasks allows only one *disk monotask* to access a disk at any time. I.e., it reads 128MB HDFS blocks sequentially, one at a time. With this approach, a single job's concurrent tasks are serialized if they require HDFS blocks on the same disk. As a result, Monotasks fails to achieve dramatic performance improvement since it has a different form of task pipelining and the pipeline stages do not overlap significantly. Unlike Monotasks, VeloxDFS employs stream managers, which allow multiple tasks to read their own logical blocks simultaneously while performing sequential I/O operations.

The I/O overhead of shuffle phase has been pointed out as one of the most significant factors that degrade the performance of MapReduce in numerous studies including Riffle [6] and ThemisMR [24]. Riffle [6] proposes to split jobs into smaller tasks for better parallelism. But all-to-all data transfer between small tasks becomes a serious performance bottleneck in shuffle phase. To reduce the overhead of shuffling with small tasks, Riffle proposed to merge fragmented intermediate shuffle files into larger blocks, which converts small, random disk I/O requests into sequential accesses.

Over the past decade, there has been a tremendous amount of research effort to develop various scheduling policies at the application framework level. In contrast, VeloxDFS is designed to support application frameworks with block stream managers that make load balancing decisions at the file system level. For a similar purpose, Kim et al.[8, 7] proposed to reduce the YARN container initialization overhead by controlling the HDFS *input split* size, i.e., increasing the input split size reduces the number of tasks and reduces the YARN container initialization overhead. However, [8, 7] did not investigate the effect of concurrent tasks on the number of disk seeks.

ThemisMR [24] is a MapReduce framework that reads and writes data records to disk exactly twice. As such, it guarantees job-level fault tolerance rather than task-level fault tolerance. ThemisMR also employs a centralized, per-node disk scheduler that stores records to disks in large batches to avoid expensive disk seek operations. Unlike Riffle and ThemisMR, VeloxDFS performs sequential I/O not only for output records but also for input files. Also, VeloxDFS is a generic distributed file system, which can be used not only by a MapReduce programming model but also by other generic data processing engines including various Yarn applications.

VI. Conclusion

In this work, we have designed and implemented *VeloxDFS*, a novel HDFS compatible file system. VeloxDFS coordinates concurrent I/O requests from multiple clients to minimize the number of disk seek operations. VeloxDFS dynamically adjusts the size of partitioned blocks to reflect I/O consumption rate and mitigates the skew problem. By reducing the I/O contention from concurrent tasks, VeloxDFS enables overcommit scheduling that schedules a larger number of tasks than there are physical CPU cores, leveraging the OS scheduler to improve the computing resource utilization of cluster. Our performance study shows that VeloxDFS is up to 1.7x faster than HDFS for multiple concurrent workloads.

Acknowledgment

This research was supported in part by Samsung Electronics, and also by IITP (grant No. 2018-0-00549, 2021-0-01817, and 2021-0-00862), and Electronics and Telecommunications Research Institute(ETRI) grant (grant No. 20ZS1310) funded by the Korean government. The first and second authors are co-first authors. They have contributed equally to the design and implementation of the techniques presented in the paper. The third author implemented most of the codebase for VeloxDFS with the help of the rest of the student authors. The corresponding author is Beomseok Nam.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI '04), 2004.

[2] Y. Kwon, M. Balaziniska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in MapReduce applications," in ACM SIGMOD International Conference on Management of Data (SIGMOD), 2012, pp. 25–36.

[3] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in MapReduce workloads using progressive sampling," in Proceedings of the Third ACM Symposium on Cloud Computing (SOCC '12), 2012.

[4] W. Kim, Y. ri Choi, and B. Nam, "Mitigating yarn container overhead with input splits," in Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017.

[5] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), 2017, p. 184–200.

[6] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: Optimized shuffle service for large-scale data analytics," in Proceedings of the Thirteenth EuroSys Conference (EuroSys '18), 2018.

[7] W. Kim, Y.-R. Choi, and B. Nam, "Coalescing hdfs blocks to avoid recurring yarn container overhead," in 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 214–221.

[8] W. Kim, Y.-r. Choi, and B. Nam, "Mitigating yarn container overhead with input splits," in 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 204–207.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in Proceedings of the Third ACM Symposium on Cloud Computing (SOCC '13), 2013.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), 2003.

[11] R. Appuswamy, C. Gkantsidis, D. Narayana, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for Hadoop: Time to rethink?" in 4th annual Symposium on Cloud Computing (SOCC '13), 2013.

[12] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding manycore scalability of file systems," in Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16), 2016.

[13] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: Light Weight Data Skew MItigation in MapReduce," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 9, pp. 2520–2533, 2015.

[14] P. A. Misra, I. Goiri, J. Kace, and R. Bianchini, "Scaling distributed file systems in resource-harvesting datacenters," in 2017 USENIX Annual Technical Conference (USENIX ATC '17), 2017.

[15] A. Lakshman and P. Malik, "Cassandra: Structured storage system on a p2p network," in Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC '09), 2009, p. 5.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07), 2007, p. 205–220.

[17] J. Lowe, "Investigating the Effects of Overcommitting YARN resources," Hadoop Summit 2016.

[18] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook messages case study," in USENIX Conference on File and Storage Technologies (FAST '14), 2014.

[19] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in 15th USENIX Conference on File and Storage Technologies (FAST 17), 2017.

[20] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in Proceedings of the Symposium on Operating System Design and Implementation (OSDI '12), 2012.

[21] V. A. B. Sanchez, W. Kim, Y. Eom, K. Jin, M. Nam, D. Hwang, J.-S. Kim, and B. Nam, "Eclipsemr: Distributed and parallel task processing with consistent hashing," in 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 322–332.

[22] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "HARDFS: Hardening HDFS with selective and lightweight versioning," in Proceedings of USENIX Conference on File and Storage Technologies (FAST '13), 2013.

[23] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the salus scalable block store," in Proceedings of the Symposium on Networked System Design and Implementation (NSDI '13), 2013.

[24] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, "Themis: An i/o-efficient mapreduce," in Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12), 2012.