Trident: Task Scheduling over Tiered Storage Systems in Big Data Platforms

Herodotos Herodotou
Cyprus University of Technology
Limassol, Cyprus
herodotos.herodotou@cut.ac.cy

Elena Kakoulli
Cyprus University of Technology
Limassol, Cyprus
elena.kakoulli@cut.ac.cy

ABSTRACT

The recent advancements in storage technologies have popularized the use of tiered storage systems in data-intensive compute clusters. The Hadoop Distributed File System (HDFS), for example, now supports storing data in memory, SSDs, and HDDs, while OctopusFS and hatS offer fine-grained storage tiering solutions. However, the task schedulers of big data platforms (such as Hadoop and Spark) will assign tasks to available resources only based on data locality information, and completely ignore the fact that local data is now stored on a variety of storage media with different performance characteristics. This paper presents Trident, a principled task scheduling approach that is designed to make optimal task assignment decisions based on both locality and storage tier information. Trident formulates task scheduling as a minimum cost maximum matching problem in a bipartite graph and uses a standard solver for finding the optimal solution. In addition, Trident utilizes two novel pruning algorithms for bounding the size of the graph, while still guaranteeing optimality. Trident is implemented in both Spark and Hadoop, and evaluated extensively using a realistic workload derived from Facebook traces as well as an industry-validated benchmark, demonstrating significant benefits in terms of application performance and cluster efficiency.

PVLDB Reference Format:
Herodotos Herodotou and Elena Kakoulli. Trident: Task Scheduling over Tiered Storage Systems in Big Data Platforms. PVLDB, 14(9): 1570-1582, 2021.
doi:10.14778/3461535.3461545

1 INTRODUCTION

Big data processing platforms such as Apache Hadoop [7] and Spark [8] are now widely used for processing large amounts of data in distributed clusters for a wide variety of applications, including web-scale data mining, online analytics, and machine learning. Such applications are typically executed in phases composed of many parallel tasks. One of the most important components of these platforms is their *task scheduler*, which is responsible for scheduling the application tasks to the available resources located on the cluster nodes, since it directly impacts the processing time and resources utilization [35]. The ultimate goal of a task scheduler is to find an optimal task distribution that will minimize the mean execution time of the scheduled tasks and maximize the utilization of the allocated resources [18].

A plethora of scheduling algorithms have been proposed in recent years for achieving a variety of complementary objectives such as better resource utilization [26, 32, 38], fairness [25, 41], workload balancing [12, 39], and data management [1, 4, 47]. One of the key strategies towards optimizing performance, employed by almost all schedulers regardless of their other objectives, is to schedule the tasks as close as possible to the data they intent to process (i.e., on the same cluster node). Achieving this *data locality* can significantly reduce network transfers (since moving code in a cluster is much cheaper than moving large data blocks) and improve both application performance and cluster utilization [18, 35].

At the same time, the rapid development of storage technologies and the introduction of new storage media (e.g., NVRAM, SSDs) [29] is stimulating the emergence of *tiered storage systems* in cluster computing. For example, HDFS, the de facto distributed file system used with Hadoop and Spark deployments, has extended its architecture to support storing data in memory and SSD devices, in addition to HDD devices [20]. OctopusFS [27] and hatS [28] extended HDFS to support fine-grained storage tiering with new policies governing how file blocks are replicated and stored across both the cluster nodes and the storage tiers. In addition, in-memory distributed file systems such as Alluxio [3] and GridGain [19] are used for storing or caching HDFS data in cluster memory.

While tiered storage systems have been shown to improve the overall cluster performance [21, 27, 28], current task schedulers are oblivious to the presence of storage tiers and the performance differences among them. In the past, scheduling a data-local task meant the task would read its input data from a locally-attached HDD device. In the era of tiered storage, however, that task might read data from a different storage media such as memory or SSD. Hence, the execution times of a set of data-local tasks can vary significantly depending on the I/O performance of the storage media each task is reading from. In addition, as the bandwidth of the interconnection network keeps increasing (and headed towards InfiniBand), the execution time bottleneck of tasks is shifting towards the I/O performance of the storage devices [5, 31].

Very few works address (fully or partly) the task scheduling problem over tiered storage. In the presence of an in-memory caching tier, some systems like PACMan [6], BigSQL [16], and Quartet [14] will simply prioritize scheduling memory-local tasks over data-local tasks (as they assume that only two tiers exist). H-Scheduler [31] is perhaps the only other storage-aware scheduler implemented for Spark over tiered HDFS. H-Scheduler employs a heuristic algorithm for scheduling tasks to available resources that, unlike our approach, does not guarantee an optimal task assignment.

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461545

In this paper, we introduce *Trident, a principled task scheduling approach that can exploit the storage type information provided by the underlying tiered storage system for making optimal scheduling decisions in both a locality-aware and a storage-tier-aware manner*. Specifically, the scheduling problem is encoded into a bipartite graph of tasks and available cluster resources, where the edge weights represent the cost of reading data based on locality and storage tier information. The problem is then formulated as a minimum cost maximum matching optimization problem, which is solved using a standard solver for finding the optimal task assignments. More concretely, our **contributions** are as follows:

(1) We formally define the problem of task scheduling over tiered storage and formulate it as a minimum cost maximum matching problem in a bipartite graph.
(2) We introduce two pruning algorithms for bounding the size of the graph and reducing the scheduling time by up to an order of magnitude without affecting the optimality of the solution.
(3) We extend YARN's resource request model with a general notion of locality preferences to account for storage tiers.
(4) We implemented the Trident scheduler in both Spark and Hadoop, showing the generality and practicality of our approach.
(5) We performed an extensive evaluation using a realistic workload and an industry-validated benchmark, showcasing significant benefits for application performance and cluster efficiency.

Outline. The rest of this paper is organized as follows. Section 2 reviews prior related work. Section 3 formally defines the task scheduling problem over tiered storage. Sections 4 and 5 present the details of our task scheduling approach when implemented in Spark and Hadoop, respectively. Section 6 describes our evaluation methodology and results, while Section 7 concludes the paper.

2 RELATED WORK

Multiple scheduling algorithms have been proposed in the past and are presented in various comprehensive surveys [18, 35]. In this section, we discuss the most relevant ones. Hadoop offers three schedulers out-of-the-box: (1) *FIFO* (First In First Out), which assigns tasks to resources in order of job submission [30]; (2) *Capacity*, which allocates resources to jobs under constraints of allocated capacities per job queue [33]; and (3) *Fair*, which assigns resources to jobs such that they get, on average, an equal share of resources over time [44]. Similarly, Spark supports FIFO and Fair scheduling. In terms of data locality, the three schedulers behave in a similar manner: given some available resources on a node, they will try to assign (in order) data-local, then rack-local, then remote tasks.

Several studies focus on improving data locality rates. Delay Scheduling [45] will have the next job wait for a small amount of time if it cannot launch a data-local task, in at attempt to increase the job's data locality. Delay scheduling is actually offered as a configurable option in all aforementioned schedulers. Wang et al. [42] focus on striking the right balance between data locality and load balancing using stochastic network theory, in order to simultaneously maximize throughput and minimize delay. Scarlett [4] and DARE [1] employ a proactive and reactive approach, respectively, for changing the number of data replicas based on access frequencies in an attempt to improve data locality. Unlike Trident, none of the above approaches support tiered storage.

A set of approaches tackle the issue of task scheduling over heterogeneous clusters that contain nodes with different CPU, memory, and I/O capabilities. One common theme involved is estimating the task execution times in order to correctly identify slow tasks (on less capable nodes) and re-execute them. LATE [47] adopts a static method to compute the progress of tasks, SAMR [9] calculates progress of tasks dynamically using historical information, and ESAMR [36] extends SAMR to employ k-means clustering for generating more accurate estimations. Tarazu [2] and PIKACHU [17] use dynamic load rebalancing to schedule tasks after identifying the fast and slow nodes at runtime. More recently, RUPAM [43] employed a heuristic for heterogeneity-aware task scheduling, which considers both task-level and hardware characteristics while preserving data locality. While the aforementioned approaches work over heterogeneous clusters, they ignore the heterogeneity resulting from different locally-attached storage devices.

The increasing memory sizes is motivating the use of distributed memory caching systems in cluster computing. PACMan [6] and Big SQL [16] utilize memory caching policies for storing data in cluster memory for speeding up job execution. In terms of task scheduling, they simply prioritize assigning memory-local tasks over data-local tasks. Quartet [14] also utilizes memory caching and focuses on data reuse across jobs. The Quartet scheduler follows a rule-based approach: schedule a task T on node N if (i) T is memory-local on N or (ii) T is node-local on N but not memory-local anywhere else. Otherwise, fall back to default scheduling with delay enabled. For comparison purposes, we implemented Quartet and extended its approach to search for SSD-local tasks first before HDD-local tasks.

H-Scheduler [31] is the only other storage-aware task scheduler designed to work over a tiered storage system such as HDFS (with tiering enabled). The key idea of H-Scheduler is to classify the tasks by both data locality and storage types, and redefine their scheduling priorities. Specifically, given available resources on some cluster node, schedule tasks based on the following priorities: local memory > local SSD > local HDD > remote HDD > remote SSD > remote memory [31]. The main issue with H-Scheduler and Quartet is that their heuristic methodology implements a best-effort approach that (in many cases) leads to sub-optimal or even poor task assignments, as we will see in Section 6. On the other hand, *the principled scheduling approach of Trident guarantees that the optimal task assignments (as formalized in Section 3.1) will always be achieved.*

3 TASK SCHEDULING OVER TIERED STORAGE

Distributed file systems such as HDFS [34] and OctopusFS [27] store data as files that are split into large blocks (128MB by default). The blocks are replicated and distributed across the cluster nodes, and stored on locally-attached HDDs. When tiering is enabled on HDFS or OctopusFS is used, the block replicas can be stored on different storage media. For example, Figure 1 shows how 3 blocks (B_1–B_3) are stored across 4 nodes (N_1–N_4), with some replicas residing in memory, SSDs, or HDDs. Various heterogeneous tiering scenarios are also supported such as a tier not spanning all nodes (e.g., not all nodes have SSDs) or having two tiers of the same storage type with different performance characteristics (e.g., PCIe vs SATA SSDs) [27].

Data processing platforms, such as Hadoop and Spark, are responsible for allocating resources to applications and scheduling

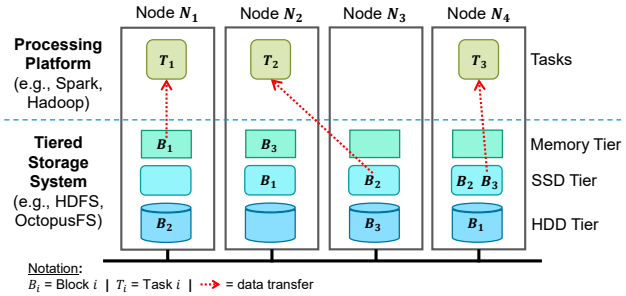

Figure 1: Example of task execution over tiered storage. Task T_1 is memory-local, T_2 is rack-local, and T_3 is SSD-local.

tasks for execution. In the example of Figure 1, assuming task T_i wants to process the corresponding data block B_i, the scheduler was able to achieve two data-local tasks (T_1 and T_3) and one rack-local task (T_2). When the storage tiers are considered, we can further classify the tasks T_1 and T_3 as memory-local and SSD-local, respectively. While current schedulers only take into account data locality, we argue (and show) that considering the storage tiers is crucial for taking full advantage of the benefits offered by tiered storage.

3.1 Problem Definition

A typical compute cluster consists of a set of *nodes* $N = \{N_1, ..., N_r\}$ arranged in a rack network topology. The cluster offers a set of *resources* $R = \{R_1, ..., R_m\}$ for executing tasks. A resource represents a logical bundle of physical resources (e.g., ⟨1 CPU, 2GB RAM⟩), such as a container in Hadoop or an Executor slot in Spark, and it is bound to a particular node. For each resource R_j, we define its location as $L(R_j) = N_k$, where $N_k \in N$. Finally, a set of *tasks* $T = \{T_1, ..., T_n\}$ require resources for executing on the cluster.

In a traditional big data environment (i.e., in the absence of tiered storage), each task contains a list of preferred node locations based on the locality of the data it will process. For example, if a task will process a data block that is replicated on nodes N_1, N_2, and N_4, its list of preferred locations contains these three nodes. However, when the data is stored in a tiered storage system such as OctopusFS or tiered HDFS, the storage tier of each block is also available. Hence, we define the *task's preferred locations* $P(T_i) = [\langle N_k, p_k^i \rangle]$ as a list of pairs, where the first entry in a pair represents the node $N_k \in N$ and the second entry $p_k^i \in \mathbb{R}$ represents the storage tier. We define p_k^i as a numeric score that represents the cost of reading the data from that storage tier. Hence, the lower the score the better for performance. Various metrics can be used for setting the scores but their absolute values are not as important as representing the relative performance across the tiers. For example, if the I/O bandwidth of memory, SSD, and HDD media is 3200, 400, and 160 MB/s, respectively, then the scores 1, 8, and 20 would capture the relative cost of reading data from those three tiers.

Scheduling a task T_i on a resource R_j will incur an *assignment cost* C based on the following cost function:

$$C(T_i, R_j) = \begin{cases} p_k^i & \text{if } L(R_j) = N_k \text{ in } P(T_i) \\ c_1 + p_k^i & \text{if } L(R_j) \text{ in same rack as some } N_k \text{ in } P(T_i) \\ c_2 & \text{otherwise (with } c_2 \gg c_1) \end{cases} \quad (1)$$

Table 1: List of notations.

Notation	Explanation
$N_k \in N$	A node N_k from a set of nodes N
$T_i \in T$	A task T_i from a set of tasks T
$R_j \in R$	A resource R_j (container/slot) from a set of resources R
$L(R_j)$	Location (i.e., node) of resource R_j
$P(T_i)$	Preferred locations (i.e., $\langle N_k, p_k^i \rangle$ pairs) of task T_i
p_k^i	Storage tier preference score for task T_i on node N_k
$C(T_i, R_j)$	Cost of scheduling task T_i on resource R_j
d	Default replication factor of the underlying file system

According to Equation 1, if the location of resource R_j is one of the nodes N_k in T_i's preferred locations, the cost will equal the corresponding tier preference score p_k^i. Alternatively, if R_j is on the same rack as one of the nodes N_k in T_i's preferred locations, the cost will equal the corresponding preference score p_k^i plus a constant c_1, which represents the network transfer cost within a rack. Otherwise, the cost will equal a constant c_2, representing the network transfer cost across racks. The inter-rack network cost is often much higher than the intra-rack cost and dwarfs the local reading I/O cost; hence, we do not add a preference score to c_2.

Using the above definitions, we can setup the task scheduling problem as a constrained optimization problem:

$$\textbf{Minimize} \sum_{(T_i,R_j) \in T \times R} x_{i,j}\, C(T_i, R_j) \quad (2)$$

$$\textbf{Subject to } x_{i,j} = \{0, 1\}\,, \forall\, (T_i, R_j) \in T \times R$$

$$\sum_{R_j \in R} x_{i,j} = 1\,, \forall\, T_i \in T$$

$$\sum_{T_i \in T} x_{i,j} = 1\,, \forall\, R_j \in R$$

The goal is to find all assignments (i.e., $\langle T_i, R_j \rangle$ pairs) that will minimize the sum of the corresponding assignment costs. The variable $x_{i,j}$ is 1 if T_i is assigned to R_j and 0 otherwise. The second constrain guarantees that each task will only be assigned to one resource, while the last constrain guarantees that each resource will only be assigned to one task. The above formulation requires that the number of tasks n equals the number of resources m. If $n < m$, the third constrain must be relaxed to $\sum_{T_i \in T} x_{i,j} \leq 1$ (i.e., some resources will not be assigned), while if $n > m$, the second constrain must be relaxed to $\sum_{R_j \in R} x_{i,j} \leq 1$ (i.e., some tasks will not be assigned). Table 1 summarizes the notation used in this section.

3.2 Min Cost Max Matching Formulation

The task scheduling problem defined above can be encoded as a bipartite graph $G = (T, R, E)$. The vertex sets T and R correspond to the tasks and resources, respectively, and together form the vertices of graph G. Each edge (T_i, R_j) in the edge set E connects a vertex $T_i \in T$ to a vertex $R_j \in R$ and has a weight (or cost) as defined in Equation 1. The constrained optimization problem formulated in Equation 2 is *equivalent* to finding a maximum matching $M = \{(T_i, R_j)\}$ with $(T_i, R_j) \in T \times R$ in the bipartite graph G that minimizes the total cost function:

$$\sum_{(T_i,R_j) \in M} C(T_i, R_j) \quad (3)$$

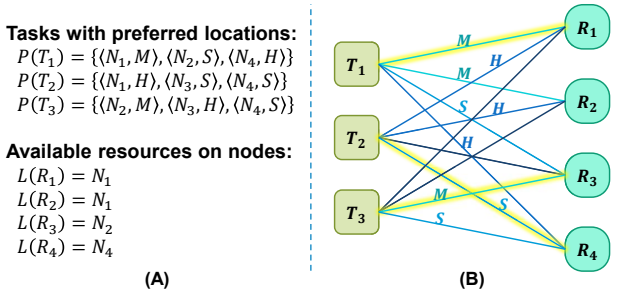

Figure 2: (A) Example of 3 tasks with preferred locations and 4 available resources. (B) Corresponding bipartite graph. M, S, and H represent the tier preference scores (or costs) for memory, SSD, and HDD, respectively. The cost for rack-local assignments are not shown for clarity. The yellow highlighted edges represent the optimal task assignments.

By definition, a matching is a subset of edges $M \subseteq E$ such that for all vertices v in G, at most one edge of M is incident on v. Hence, at most one task will be assigned to one resource and vice versa. A maximum matching is a matching of maximum cardinality, i.e., it contains as many edges as possible. Since any task can potentially be assigned to any resource, maximum matching will contain either all tasks or all resources, depending on which set is smaller. Hence, the constrains listed in Equation 2 are all satisfied. Consequently, by solving the minimum cost maximum matching problem on G, we are guaranteed to assign as many tasks as possible to resources and to attain the lowest total assignment cost possible.

Figure 2(A) illustrates an example with three tasks and four available resources located on three distinct nodes. Each task T_i has a list of 3 preferred locations (i.e., ⟨node, tier⟩ pairs) according to the storage location of the corresponding data block B_i shown in Figure 1. For ease of reference, each tier preference score is indicated using the constants M, S, and D, corresponding to the memory, SSD, and HDD tiers of the underlying storage system, with $M < S < D$. The created bipartite graph with 7 vertices (3 for tasks and 4 for resources) is visualized in Figure 2(B). Each edge corresponds to a potential assignment of a task to a resource and is annotated with the cost computed using Equation 1. The goal of task scheduling in this example is to select the three edges that form a maximum matching and minimize the total cost. The optimal task assignment (see highlighted edges in Figure 2(B)) consists of two memory-local tasks (T_1 on R_1 and T_3 on R_3) and one SSD-local task (T_2 on R_4).

Several standard solvers can be used for solving the minimum cost maximum matching problem and finding the optimal task assignment, including the Simplex algorithm, the Ford-Fulkerson method, and the Hungarian Algorithm [13]. While the Simplex algorithm is known to perform well for small problems, it is not efficient for larger problems because its pivoting operations become expensive. Its average run time complexity is $O(\max(n, m)^3)$, where n is the number of tasks and m is the number of resources, but has a combinatorial worse case complexity. The Ford-Fulkerson method requires converting the problem into a minimum cost maximum flow problem, with complexity $O((n + m)nm)$ in this setting. We have chosen to use the *Hungarian Algorithm* as it runs in a strongly polynomial time with low hidden constant factors, which makes it more efficient in practice. Specifically, its complexity is $O(nmx+x^2\lg(x))$, where $x = \min(n, m)$. Below, we introduce two vertex pruning algorithms that reduce the complexity to $O(\min(n, m)^3)$, while still guaranteeing an optimal solution. Approximation algorithms are also available for finding a near-optimal solution with a lower complexity [15]. However, the scheduling time of our overall approach is so low (as evaluated in Section 6.4) that we opted for finding the optimal solution with the Hungarian Algorithm.

Algorithm 1 Compute and prune available resources

```
1:  procedure ComputeResources(tasks[], resourceSets[])
2:      resources = ∅                          ▷ List of available resource slots
3:      if totalAvailableSlots(resourceSets) ≥ d × tasks.length
4:          nodesToTasks = ∅                   ▷ Map node to local task count
5:          for each T_i in tasks do
6:              for each N_k in P(T_i) do
7:                  nodesToTasks.get(N_k).increment
8:          for each S_j in resourceSets do
9:              if nodesToTasks.contains(S_j.node)
10:                 availSlots = computeAvailableSlots(S_j)
11:                 localTasks = nodesToTasks.get(S_j.node)
12:                 maxSlots = min{availSlots, localTasks}
13:                 add maxSlots entries to resources
14:         if resources.length ≥ tasks.length
15:             return resources               ▷ Found enough resources
16:     for each S_j in resourceSets do
17:         availSlots = computeAvailableSlots(S_j)
18:         add availSlots entries to resources
19:     return resources                       ▷ Return all available resource slots
```

3.3 Resource and Task Pruning Algorithms

In many cases, the number of tasks ready for execution does not equal the number of available resources. For example, a small job executing on a large cluster will have much fewer tasks than available resources, while a large job executing on a small or busy cluster will have much more tasks than available resources. Next, we describe two algorithms for pruning excess resources or tasks that reduce the graph size and lead to a more efficient execution of the Hungarian Algorithm, without affecting the optimality of the solution.

Pruning Excess Resources. Algorithm 1 shows the process of computing and pruning the available resources in a cluster. The input consists of a list of tasks and a list of resource sets. A *resource set* represents a bundle of resources available on a node, which can be divided into resources (i.e., slots) for running the tasks. The pruning of excess resources is enabled when the total available slots across all resource sets is d times higher than the number of tasks (line 3), where d is the default replication factor of the file system. The rational for this limit will be explained after the algorithm's description. First, the lists with the preferred locations of all tasks are traversed for counting the number of local tasks that can potentially be executed on each node (lines 4-7). The counts are stored in a map for easy reference. Next, each resource set S_j is considered (line 8). If S_j can be used to run at least one data-local task (line 9), then we need to compute the maximum number of slots (*maxSlots*) that can be created from S_j for running data-local tasks. *maxSlots* will equal the minimum of (a) the total number of

available slots from S_j, and (b) the number of tasks that contain S_j's node in their preferred locations list (lines 10-12). Finally, *maxSlots* entries (i.e., resource slots from S_j) are added in the list of available resources (line 13). After traversing all resource sets, if the list of available resources has more entries than tasks, the list is returned and the process completes (lines 14-15). Otherwise, resource slots for all remaining available resources are added in the result list (lines 16-18). This final step (lines 16-18) is also performed when the number of total available slots is less than d times the number of tasks for returning all available resources.

Suppose 3 tasks are ready for execution and their preferred locations are as shown in Figure 2(A). Further, the cluster consists of 6 nodes (N_1–N_6), each with enough resources to create 3 slots. Hence, there are a total of 18 available slots and the pruning will take place. First, the number of possible data-local tasks will be computed as $\{N_1 : 2, N_2 : 2, N_3 : 2, N_4 : 3\}$. For the resource set of N_1, even though there are 3 available slots, only 2 will be added in the result list as only 2 can host data-local tasks. The same is true for the resource sets of N_2 and N_3. For N_4, all 3 available slots will be added in the result list. Finally, since N_5 and N_6 do not appear in the tasks' preferred locations, no slots will be added in the result list. Overall, only 9 out of the 18 possible resource slots will be considered for the downstream task assignments. In fact, even if the number of available resource slots were much higher, 9 is the largest number of slots this process will return for this example. In general, n tasks and m resources (with $n \ll m$) will lead to a graph with $n + m$ vertices and nm edges without pruning, but only $4n$ vertices and $3n^2$ edges with pruning, showcasing that a *massive pruning of excess resources is possible for small jobs.*

Next, consider a different example where the same 3 tasks are present but only 2 slots are available on each of the nodes N_1–N_4. If pruning were to take place (lines 3-15), all 8 slots would be added in the result list, rendering the pruning process pointless. This behavior is expected since each task will typically have 3 preferred locations (since each file block has 3 replicas by default) spread across several nodes. Hence, by enabling pruning only when the number of available slots is greater than d times the tasks, we avoid going through a pruning process that will have no to little benefits.

Even though a large number of available resources may be pruned, *the optimality of the task assignments is still guaranteed* because (a) the excluded resources cannot lead to data-local assignments, and (b) the retained resources that can lead to data-local assignments are more than the tasks. Hence, the excluded resources would not have appeared in the final task assignments.

Pruning Excess Tasks. Algorithm 2 shows the process of pruning excess tasks in the presence of few available resources. In particular, pruning is enabled when the number of tasks ready for execution is higher than d times the total number of available resources (line 2). The rational is similar to before: avoid going through the process of pruning tasks when pruning is not expected to significantly reduce (if any) the number of tasks. When pruning is enabled, the available resources are traversed for collecting the set of distinct nodes (*resourceNodes*) they are located on (lines 3-5). Next, each task is added in the result list only if at least one of its preferred locations is contained in *resourceNodes* (lines 7-9). If the selected tasks are more than the available resources, the result list is returned and the process completes (lines 10-11). In the opposite case, or when pruning is not enabled, the list with all tasks is returned (line 12).

Algorithm 2 Prune tasks available for execution

```
1:  procedure PruneTasks(tasks[], resources[])
2:      if tasks.length ≥ d × resources.length
3:          resourceNodes = ∅                    ▷ Set of nodes with resources
4:          for each R_j in resources do
5:              resourceNodes.add(L(R_j))
6:          selectedTasks = ∅                    ▷ List of selected tasks
7:          for each T_i in tasks do
8:              if resourceNodes.containsAny(P(T_i))
9:                  selectedTasks.add(T_i)
10:         if selectedTasks.length ≥ resources.length
11:             return selectedTasks             ▷ Found enough tasks
12:     return tasks                             ▷ Return all tasks
```

Continuing with the example with the 3 tasks (recall Figure 2(A)), suppose that only one resource slot is available on node N_2. In this case, only tasks T_1 and T_3 will be included in the selected tasks list as they record N_2 in their preferred locations; task T_2 will be excluded. The optimality of the task assignments is safeguarded because the excluded tasks (which cannot lead to data-local assignments) would have never been selected by the Hungarian Algorithm since there are still more (data-local) tasks than available resources.

4 TASK SCHEDULING IN APACHE SPARK

Spark is a data processing framework that utilizes a restricted form of distributed shared memory, called Resilient Distributed Datasets (RDDs), for enabling in-memory computations in a fault-tolerant way [46]. Spark applications invoke a set of coarse-grained deterministic transformations (e.g., map, filter, join) and actions (e.g., count, collect) on RDDs for implementing their business logic.

4.1 Current Task Scheduling

A Spark application executes as a set of independent *Executor* processes coordinated by the *Driver* process. Initially, the Driver connects to a cluster manager (either Spark's Standalone Manager [8], Hadoop YARN [40], or Mesos [23]) and receives resource allocations on cluster nodes that are used for running the Executors. The Driver is responsible for the application's task scheduling and placement logic, while the Executors are responsible for running the tasks and storing RDD data over the entire duration of the application.

Internally, an application is divided into *jobs* based on RDD actions. A job is a directed acyclic graph of *stages* built based on the RDD lineage graph. Finally, a stage consists of a set of parallel *tasks*. Whenever a stage S is ready for execution (i.e., its input data is available), the *Task Scheduler* is responsible for assigning S's tasks to the available resources (or slots) of the Executors. The default scheduling algorithm is as follows. Given some available slots on Executor E running on some node N, look for a task that needs to process a data partition cached on E, thus creating a *process-local* assignment. Otherwise, look for a task that needs to process a data block stored on N, thus creating a *data-local* assignment. Otherwise, make a random assignment if the task has no locality preferences, or a rack-local assignment, or a remote assignment, in that order.

4.2 Trident Scheduler in Spark

The Trident Scheduler is proposed to replace the current Task Scheduler in the Spark Driver in order to take advantage of the storage tier information of the processed data. The input to Trident consists of (i) a list of tasks belonging to the same stage along with their preferred locations, and (ii) the list of Executors, each with its available resource set. Given this input, Trident utilizes Algorithms 1 and 2 to select which resources and tasks to use, builds the bipartite graph as described in Section 3.2, and uses the Hungarian Algorithm for making the optimal task assignments.

The Spark execution model creates two additional noteworthy scheduling scenarios that are naturally handled by our graph encoding. First, the preferred location of a task T can be an Executor E containing a cached data partition created during a previous stage execution. Assigning task T to E achieves the best locality possible as it leads to a process-local execution. In this case, we set the tier preference score to zero, thereby guiding Trident in favoring process-local assignments over all other. Second, tasks that will read input from multiple locations during a shuffle (e.g., tasks executing a reduceByKey) have no locality preference. Since the current Task Scheduler schedules such tasks before making any rack-local (or lower) assignments, we set their assignment cost to a number lower than the network cost c_1 to ensure that Trident has the same behavior. In conclusion, *our graph-based formulation can easily generalize to a variety of locality preferences for task assignment.*

5 TASK SCHEDULING IN APACHE HADOOP

Apache Hadoop consists of a resource management layer and a processing layer [40]. The former follows a master/worker architecture, where the *Resource Manager* is responsible for allocating resources to running applications and the *Node Managers* run on the worker nodes and manage the user processes on those nodes. In the processing layer, an *Application Master* manages an application's life-cycle, negotiates resources from the Resource Manager, and works with the Node Managers to execute and monitor the tasks. Allocated resources have the form of *Containers*, which represent a specification of node resources (e.g., CPU cores, memory).

5.1 Current Resource and Task Scheduling

When it starts, the Application Master (AM) creates first a set of *resource requests* based on the input data to process and the tasks it plans to execute. A resource request contains: (a) a priority within the application (e.g., map or reduce); (b) a locality preference (either a host name, a rack name, or '$*$'); (c) resources per container (e.g., 1 CPU, 2 GB RAM); and (d) number of containers [40]. The AM uses heartbeats (every 1 second by default) to send the requests to the Resource Manager (RM) and to receive the allocated containers.

The RM receives the resource requests and stores them internally. It also receives periodic heartbeats from Node Managers containing their resource availability. Upon a node heartbeat, the RM uses a pluggable *Scheduler* to allocate the node resources to applications. The order of applications and the amount of resources to allocate to each one depend on the type of scheduler (e.g., FIFO vs. Fair). Data locality is taken into account only when it is time to allocate resources to a particular application at a specific request priority. At that point, the scheduler tries to allocate data-local containers; otherwise, rack-local containers; otherwise, remote ones. The current schedulers also support the option of doing the allocations asynchronously (e.g., every 100ms) based on all available resources in the cluster, but do so in the same manner as described above.

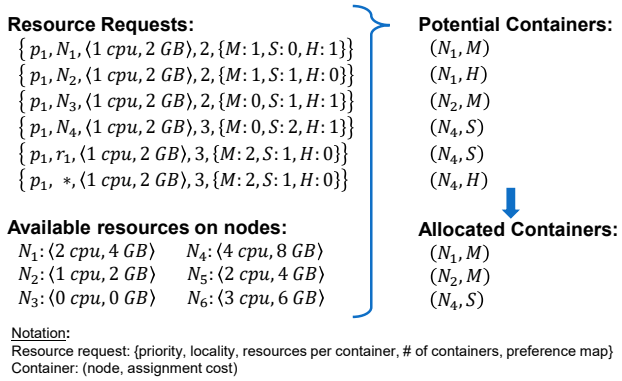

Figure 3: Example of resource requests based on the preferred locations of the tasks shown in Figure 2(A), and allocated containers after executing Algorithm 3.

Upon a heartbeat from an AM, the RM returns the allocated containers. The AM will then assign tasks to the allocated containers while taking data locality into account following the same strategy as above: first data-local, then rack-local, then remote. Finally, the tasks are sent to the Node Managers for execution.

Therefore, *scheduling based on locality preferences in Hadoop actually takes place at two distinct locations*: (1) the RM for allocating containers to applications, and (2) the AM for assigning tasks to the allocated containers. Consequently, the Trident scheduler in Hadoop consists of two components, one running in the RM (Section 5.3) and one running in the AM (Section 5.4).

5.2 Extending YARN's Resource Request Model

Given a list of tasks with preferred locations, the AM will generate the resource requests as follows. For each distinct node N (or rack R) that appears in the preferred locations, a request will be created for N (or R), where the number of containers will equal the number of times N (or R) is found in the preferred locations. Finally, a '$*$' (i.e., anywhere) request will be created, with the number of containers equal to the number of tasks.

In order for the RM's Scheduler to consider storage tier preferences, we need to extend the resource request model to include them. We do so by introducing the notion of a *preference map* in the resource request, which maps each tier preference score (recall Section 3.1) to the number of containers requested for that score (i.e, tier). Consider the example in Figure 2(A) containing 3 tasks with preferred locations. Tasks T_1 and T_2 list node N_1 in their preferred locations, which leads to the creation of one resource request for 2 containers on N_1, as shown in Figure 3. The preference map will contain the entries $\{M:1, H:1\}$ because for T_1, N_1 is paired with the score M and for T_2, N_1 is paired with H. Regarding node N_2, the preferred locations for T_1 and T_3 contain the pairs $\langle N_2, S\rangle$ and $\langle N_2, M\rangle$, respectively. Hence, the resource request for N_2 asks for 2 containers, with preference map $\{M:1, S:1\}$. This information can then be used by the scheduler for making better decisions. For

example, instead of allocating 2 containers on N_1 (which would lead to 2 data-local containers, 1 memory-local and 1 HDD-local), it would be better to allocate 1 container on N_1 and 1 on N_2, which would lead to 2 memory-local containers.

When computing the preference map for a rack-local resource request, we only count the per-task lowest scores that are paired with nodes belonging to that rack. The rational is to match the default behavior of the underlying tiered storage systems that direct a rack-local read to the highest tier (with the lowest score). In the example of Figure 2(A), all nodes belong to the same rack (r_1) and the lowest scores for T_1, T_2, and T_3 are M, S, and M, respectively. Hence, the preference map contains $\{M : 2, S : 1\}$. The same process is performed for computing the preference map for the '$*$' request.

It is important to note that the current resource request model forms a "lossy compression of the application preferences" [40], which makes the communication and storage of requests more efficient, while allowing applications to express their needs clearly [40]. However, the exact preferred locations of the tasks cannot be mapped from the resource requests back to the individual tasks. Hence, the Trident's component running in the RM will follow a different scheduling approach rather than using the Hungarian Algorithm, described next.

5.3 Trident's Scheduling Component in the RM

The Scheduler in the RM is responsible for allocating resources to the various running applications by effectively making three decisions: (1) for which application to allocate resources next; (2) how many resources to allocate to that application; (3) which available resources (i.e., containers) to allocate. The first two decisions are subject to various notions or constraints of capacities, queues, fairness, etc. [35]. Locality is only taken into consideration during the third decision, when a specific amount of available resources are allocated to a particular application. Hence, our Trident Scheduler, which focuses only on the third decision, can be incorporated into a variety of existing schedulers, including FIFO, Capacity, and Fair, for making assignments based on both node locality and storage tier information.

Algorithm 3 shows Trident's container allocation process, running in YARN's Resource Manager. The input is a list of resource requests submitted by an application (with a particular priority) and the number of maximum containers to allocate based on queue capacity, fairness, etc. The high level idea is to first build a list of potential containers to allocate based on locality preferences and then allocate the containers with the lowest assignment cost. The total number of containers is computed as the minimum of the total number of requested containers and the maximum allowed containers (line 3). Next, for each resource request S_i^n that references a particular cluster node N_k (lines 4-5), we compute the number of available containers on N_k based on N_k's available resources and S_i^n's requested resources per container (line 6). The number of containers on N_k (*numConts*) will equal the minimum between the number of available containers and the number of requested containers in S_i^n (line 7). Finally, *numConts* containers will be added in the list of potential containers (line 8). The assignment cost for each container is also computed based on the preference map in S_i^n in procedure *AddContainers*, which will be described later.

Algorithm 3 Allocate containers in YARN's Resource Manager

```
1:  procedure AllocateContainers(requests[], maxContainers)
2:      containers = ∅                          ▷ List of potential containers
3:      totalConts = min{getTotalConts(requests), maxContainers}
4:      for each node-local request S_i^n in requests do
5:          N_k = getNode(S_i^n)
6:          availConts = computeAvailableContainers(N_k, S_i^n)
7:          numConts = min{availConts, S_i^n.numContainers}
8:          addContainers(containers, numConts, N_k, S_i^n, 0)
9:      if containers.length < totalConts
10:         for each rack-local request S_i^r in requests do
11:             for each N_k in getNodes(S_i^r) do
12:                 availConts = computeAvailableContainers(N_k, S_i^r)
13:                 numConts = min{availConts, S_i^r.numContainers}
14:                 addContainers(containers, numConts, N_k, S_i^r, c_1)
15:                 if containers.length ≥ totalConts
16:                     break double for loop
17:     if containers.length < totalConts
18:         add remaining containers to containers with cost c_2
19:     sort(containers)                        ▷ Sort containers on cost
20:     return containers.take(totalConts)
21: procedure AddContainers(containers, numConts, node,
                            request, rackCost)
22:     iter = request.preferenceMap.getSortedIterator()
23:     currCount = 0
24:     for i = 0 to numConts do
25:         if currCount == 0
26:             currEntry = iter.next
27:             currCost = currEntry.getKey
28:             currCount = currEntry.getValue
29:         containers.add(Container(node, currCost + rackCost))
30:         currCount = currCount − 1
```

If the number of potential containers so far is less than the number of needed containers (line 9), a similar process is followed for the resource requests with rack locality. Specifically, for each rack-local resource request (line 10) and for each node in the corresponding rack (line 11), the appropriate number of containers is added in the list of potential containers (lines 12-14). In this case, the network cost c_1 is added to the assignment cost of each container (recall Equation 1). As soon as the number of needed containers is reached, the double for loop is exited for efficiency purposes (lines 15-16). If the number of collected containers is still below the needed ones, the remaining potential containers from random nodes are added in the list with assignment cost equal to c_2 (lines 17-18).

Due to the aggregate form of the resource requests (recall Section 5.2), it is possible that the number of potential containers is higher than the needed containers (*totalConts*), even after the first loop iteration of node-local requests (lines 4-8). In common scenarios where data blocks are replicated 3 times, this number will typically equal 3 times *totalConts*. This behavior is desirable in order to consider all storage tier preferences of the requests. Hence, our last step is to select the *totalConts* containers with the smallest assignment cost from the list of potential containers (lines 19-20). The *sort* on line 19 dominates the complexity of Algorithm 3 as

$O(nlg(n))$, where n is the number of potential containers, which typically equals 3 times the number of total requested containers.

The *AddContainers* procedure in Algorithm 3 is responsible for creating and adding a number of potential containers on a node. The assignment costs for each container depend on the tier preferences of the resource request. The key intuition of the algorithm (lines 22-30) is to assign the lowest cost first as many times as requested. Next, assign the second lowest cost as many times as requested, and so on. For example, suppose the preference map contains $\{M : 2, S : 3, H : 1\}$ and 3 containers are needed. The 3 corresponding assignment costs to containers will equal M, M, and S.

Figure 3 shows a complete example with (i) the resource requests generated based on the tasks preferred locations from Figure 2(A), and (ii) the current available resources in the cluster with 6 nodes. The resource request on N_1 asks for 2 containers, which can fit in the available resources of N_1. Hence, two potential containers are created, one with cost M and one with cost H. The request on N_2 also asks for 2 containers but only 1 can fit there; thus, 1 container is created with cost M. There are no available resources on N_3 so no containers are created there. Finally, even though 4 containers can fit on N_4, the corresponding request asks for only 3, which leads to the creation of 3 potential containers with costs S, S, and H. At this point, the list of potential containers contains 6 entries, which are more than the 3 total requested containers. Finally, this list is sorted based on increasing assignment cost and the first 3 containers are allocated to the application, leading to the best resource allocation based on preferred (node and storage tier) locality.

5.4 Trident's Scheduling Component in the AM

As soon as the MapReduce Application Master receives a set of allocated containers, it needs to assign tasks to them. The Trident Scheduler replaces the default scheduler for making optimal storage-tier-aware assignments. In particular, Trident will build the bipartite graph containing map tasks (the only type of tasks with locality preferences in MapReduce) and the allocated containers along with the assignment costs, as described in Section 3.2. In the case of MapReduce, the number of allocated containers will always be less than or equal to the number of tasks. Hence, only Algorithm 2 is implemented in MapReduce for pruning excess tasks when only a few containers are allocated. Finally, Trident will employ the Hungarian Algorithm for finding the optimal task assignments on the allocated containers. Regarding reduce tasks (which do not have locality preferences), they are randomly assigned to their allocated containers in the same manner performed by the default scheduler.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness and efficiency of the Trident Scheduler in making both locality-aware and storage-tier-aware scheduling decisions for improving application performance and cluster efficiency. Our evaluation methodology is as follows:

(1) We study the effect of Trident when scheduling a real-world MapReduce workload from Facebook (Section 6.1).
(2) We investigate the impact of input data size and application characteristics on task scheduling using an industry-validated benchmark on both Hadoop (Section 6.2) and Spark (Section 6.3).
(3) We evaluate the scheduling time needed by Trident (Section 6.4).

Table 2: Facebook job size distributions, binned by data sizes.

Bin	Data size	% of Jobs	% of Resources	% of I/O
A	0-128MB	74.4%	25.0%	3.2%
B	128-512MB	16.2%	12.2%	16.1%
C	0.5-1GB	4.0%	7.3%	12.0%
D	1-2GB	3.0%	13.4%	19.3%
E	2-5GB	1.6%	20.8%	21.9%
F	5-10GB	0.8%	21.4%	27.5%

Experimental Setup. Our evaluation is performed on an 11-node cluster running CentOS Linux 7.2 with 1 Master and 10 Workers. The Master node has a 64-bit, 8-core, 3.2GHz CPU, 64GB RAM, and a 2.1TB RAID 5 storage configuration. Each Worker node has a 64-bit, 8-core, 2.4GHz CPU, 24GB RAM, one 120GB SATA SSD, and three 500GB SAS HDDs. We implemented our approach in Apache Hadoop v2.7.7 and Apache Spark v2.4.6. For the underlying file systems we used HDFS v2.7.7 (without enabling tiering) as a baseline and OctopusFS [27], a tiered file system that extends and is backwards compatible to HDFS. OctopusFS was configured to use 3 storage tiers with 4GB of memory, 64GB of SSD, and 3×320GB of HDD space on each Worker node. The default replication factor is 3 and the default block size is 128MB for both file systems.

Implementation. Hadoop and Spark were modified to propagate the storage tier information from the input file readers to the schedulers, in the same way node locations are propagated. Trident was implemented as a pluggable component overriding the scheduling interfaces provided by the two systems. Overall, we added 2875 lines of Java code to Hadoop and 944 lines of Scala code to Spark.

Schedulers. In addition to our *Trident Scheduler*, we implemented two more schedulers from recent literature, namely *H-Scheduler* [31] and *Quartet* [14] (as described in Section 2), within the MapReduce Application Master and the Spark Driver. When running the Hadoop experiments, we used the Capacity Scheduler in YARN's Resource Manager, as was done in [14]. For comparison purposes, we also tested both Hadoop's and Spark's *Default* task schedulers, which do not take storage tier into consideration.

Performance Metrics. The various schedulers are compared using three performance metrics: (1) the *data locality* of tasks, i.e., the percentage of memory-, SSD-, HDD-, and rack-local tasks scheduled in each scenario; (2) the *reduction in completion time* of jobs compared to the baseline; and (3) the *improvement in cluster efficiency*, defined as finishing the jobs by using the least amount of resources [6, 21]. The cluster efficiency is computed by summing the individual runtimes of all tasks in a job. All results shown are averaged over 3 repetitions.

6.1 Evaluation with Facebook Workload

This part of the evaluation is based only on a MapReduce workload as it is derived from real-world production traces from a 600-node Hadoop cluster deployed at Facebook [11]. With the traces, we used the SWIM tool [37] to generate and replay a realistic and representative workload that preserves the original workload characteristics, including the distribution of input sizes and skewed popularity of data [6]. The workload comprises 1000 jobs scheduled for execution over 6 hours and processing 92GB of input data. When using

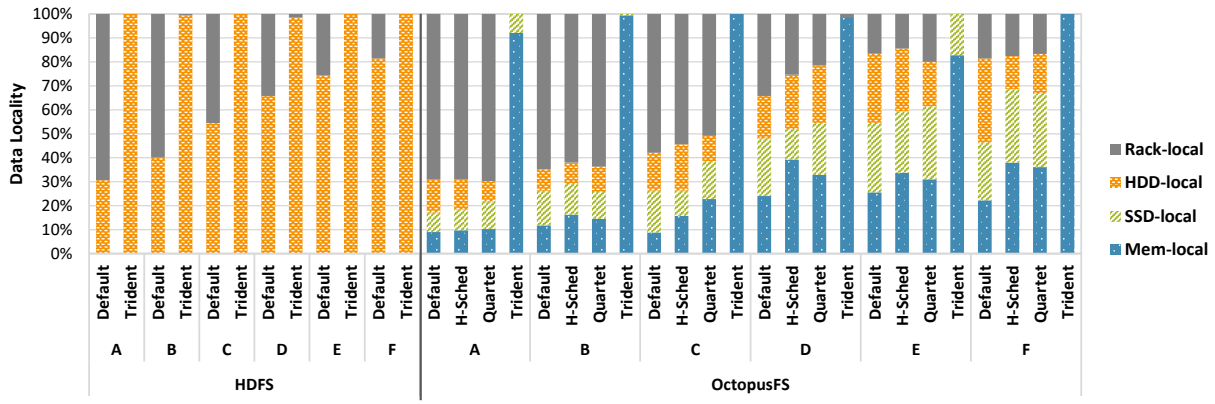

Figure 4: Data locality rates for all schedulers over the two file systems, broken down into the six Facebook workload bins.

OctopusFS, we enabled its Least Recently Used eviction policy so that later jobs in the workload can take advantage of the memory tier (since the aggregate capacity of the memory tier is 40GB) [21].

To differentiate the effect of task scheduling on different jobs, we split them into six bins based on their input data size. Table 2 shows the distribution of jobs by count, cluster resources they consume, and amount of I/O they generate. As noted in previous studies [6, 10], the jobs exhibit a heavy-tailed distribution of input sizes. Even though small jobs that process <128MB of data dominate the workload (74.4%), they only account for 25% of the resources consumed and perform only 3.2% of the overall I/O. On the other hand, jobs processing over 1GB of data account for over 54% of resources and over 68% of I/O. More in-depth workload statistics can be found in [21].

We executed the workload on Hadoop over HDFS (without tiering) using the Default and Trident schedulers as well as on Hadoop over OctopusFS using all four schedulers. Figure 4 shows the data locality rates for all schedulers over the 2 file systems, broken down according to the job bins. With HDFS and the Default Scheduler, there is a clear increasing trend in the percent of data-local tasks (note that all data is stored on HDDs) as the job size increases. The achieved data locality is low at 30-40% for small jobs (Bins A, B) for a combination of reasons: (i) the cluster is busy, (ii) these jobs have only a few tasks to run, and (iii) the scheduler considers one node at a time for task assignments. Hence, it is unlikely that any given node will be contained in the tasks' preferred locations. With increasing job sizes (and number of tasks), there are more opportunities for data-local scheduling and the data locality percentage increases up to 81%. The Trident Scheduler, on the other hand, considers all available resources together at all times, and hence it is *able to achieve almost 100% of data locality for all job sizes.*

With OctopusFS, the trend of data-local tasks for the Default Scheduler is the same as with HDFS (see Figure 4). As the Default Scheduler ignores the storage tier, those data-local tasks are (roughly) divided equally into memory-, SSD-, and HDD-local tasks. The H-Scheduler and Quartet have similar or slightly higher overall data-locality rates compared to the Default Scheduler. For small jobs, since there are little opportunities for data-local tasks (for the same reasons explained above), there is also little chance for doing any meaningful storage-tier-aware task assignments. For

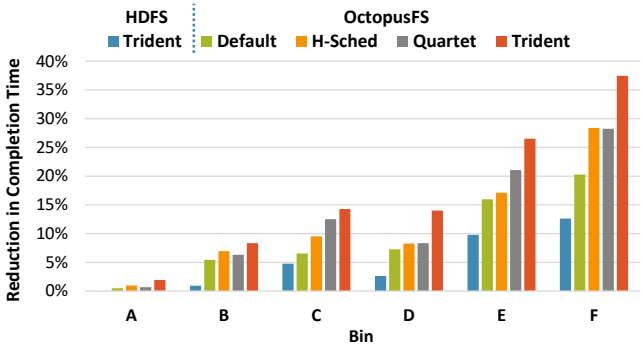

Figure 5: Percent reduction in completion time for Facebook workload, compared to Default Scheduler over HDFS.

bigger jobs, both schedulers are able to make more memory-local assignments, reaching 30-40% of the total tasks and around 50% of the data-local tasks. In addition, SSD-local tasks are typically more compared to HDD-local tasks. With OctopusFS, not only is Trident able to reach almost 100% of data locality for all job sizes, it also obtains over 83% of memory-local tasks. In fact, in 4 out of the 6 bins, *Trident is able to achieve over 99% of memory-local tasks*, showcasing Trident's ability to find *optimal tasks assignments* in terms of both locality and storage tier preferences in a busy cluster.

Figure 5 shows the percent reduction in job completion time compared to using the Default Scheduler over HDFS for each bin (recall Table 2). Using the Trident Scheduler over HDFS improves the overall data-local rates as explained above, which in turn reduces job completion time modestly, up to 13% for large jobs (Bins F). Much better benefits are observed when data is stored in OctopusFS as data is residing in multiple storage tiers, including memory and SSD. Even though the Default Scheduler over OctopusFS does not take into account storage tiers, it still benefits from randomly assigning memory- and SSD-local tasks, and hence, it is able to achieve up to 20% reduction in completion time. The storage-tier-aware schedulers are able to increase the benefits further, depending on the job size. Small jobs (Bins A, B) experience only a small improvement (<8%) in completion time for all schedulers. This is not surprising since time spent on I/O is only a small fraction compared to CPU

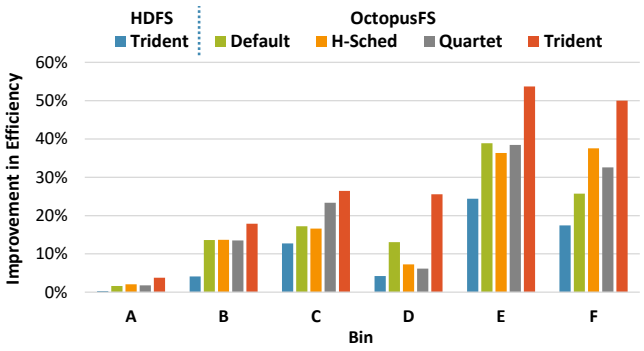

Figure 6: Percent improvement in cluster efficiency for Facebook workload, compared to Default Scheduler over HDFS.

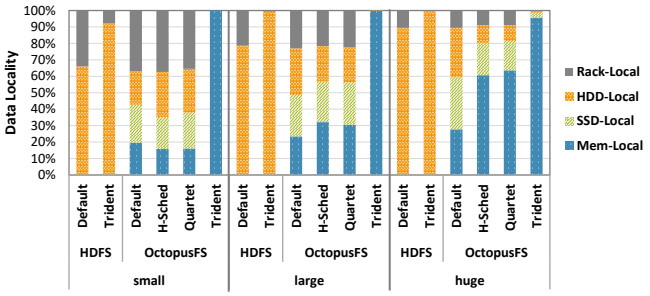

Figure 7: Data locality rates for all schedulers over the two file systems for the HiBench MapReduce workload.

and network overheads. The gains in job completion time increase as the input size increases, while we start observing different trends across the schedulers. In particular, H-Scheduler is able to provide an additional 2%-8% gains over the Default Scheduler, resulting in up to 28% gains for large jobs (Bin F). Quartet offers similar performance, with only 3% higher gains for jobs belonging in bins C and E. Finally, Trident is able to consistently provide the *highest reduction in completion time across all job bins*, with 14%-37% gains for large jobs, almost double compared to the Default Scheduler.

With each memory- and SSD-local access, the cluster efficiency improves as there is more I/O and network bandwidth available for others tasks and jobs. Figure 6 shows how this improvement relates to the different job bins. Larger jobs have a higher contribution in efficiency improvement compared to smaller jobs since they are responsible for performing a larger amount of I/O (recall Table 2). Across different schedulers, the trends for efficiency improvement are similar to the trends for completion time reduction shown in Figure 5 and discussed above: benefits improve with larger jobs and Trident always offers the highest gains. Hence, improvements in efficiency are often accompanied by lower job completion times, doubling the benefits. For example, Trident is able to reduce completion time of large jobs by 37%, while consuming 50% less resources. Another interesting observation (not shown due to lack of space) is that the shuffling time in Hadoop also decreases by up to 37% for large jobs due to the high memory-locality rates achieved by Trident, which reduce local disk I/O and network congestion.

6.2 Hadoop Evaluation with HiBench

In order to further investigate the impact of task scheduling on a variety of workloads exhibiting different characteristics, we used the popular HiBench benchmark v7.1 [24], which provides implementations for various applications on both Hadoop MapReduce and Spark. In total, eight applications were used spanning four categories: micro benchmarks (TeraSort, WordCount), OLAP queries (Aggregation, Join), machine learning (Bayesian Classification, K-means Clustering), and web search (PageRank, NutchIndex) [22]. In addition, all workloads were executed using three *data scale profiles*, namely *small*, *large*, and *huge*, which resulted in about 200MB, 1.5GB, and 10GB of input data per application, respectively.

Since the individual workload characteristics do not affect data locality rates, we present the aggregate rates for each scale profile in Figure 7. The overall trend of data locality rates is similar to the one observed for the Facebook workload: larger jobs exhibit more data-local tasks. However, in these experiments, the rates are much higher since the cluster is lightly loaded, and thus there are more scheduling opportunities (note that HiBench runs one application at a time). Hence, the Default Scheduler is able to achieve 66%–89% of data locality instead of 31%–81% in the case of Facebook. Compared to the Default Scheduler, the H-Scheduler and Quartet offer no to little improvement in terms of memory-locality for *small* and *large* jobs. The two schedulers are able to achieve good results only when scheduling *huge* jobs, for which there are a lot of available resources in the cluster, resulting in about 62% memory-local tasks, followed by 19% SSD-local tasks, and 10% HDD-local tasks. Finally, the Trident Scheduler over OctopusFS is *able to achieve 100% data locality with over 96% memory-locality across all three data scales*, demonstrating once again its superior scheduling abilities.

Figure 8(A) shows the percent reduction in completion time (compared to the Default Scheduler over HDFS) of the eight HiBench applications run using the *large* data scale. As expected, I/O intensive applications (i.e., TeraSort, Aggregation, K-means) display the highest benefits across all schedulers, since scheduling more memory-local tasks has a direct impact in reducing both the generated I/O and the overall job execution time. Simply using the Default Scheduler over OctopusFS results in 23%–31% higher performance for these applications, while H-Scheduler increases the benefits to 25%–35%. Interestingly, Quartet offers almost no benefits over the Default Scheduler, mainly because it falls back to delay scheduling when it cannot make any data-local assignments [14]; a strategy that does not increase data locality rates in this setting, and thus, only causes overhead. Finally, *Trident is able to significantly boost performance up to 44% (i.e., almost 2× speedup)* due to its 100% memory-locality rates.

The CPU-intensive jobs (i.e., WordCount, Join, Bayes, PageRank) exhibit more modest benefits since the I/O gains from improved scheduling are overshadowed by the CPU processing needs. The benefits from Trident over OctopusFS range between 23% and 29%, while they are much lower for the other three schedulers at 8%-23%. Finally, Trident is the only scheduler able to offer any meaningful benefits (28% compared to ~6% for the other schedulers) to the shuffle-intensive NutchIndex job because running 100% data-local tasks frees up the network for the demanding shuffle process.

Figure 8(B) shows the corresponding improvement in cluster efficiency for the *large* HiBench applications. The efficiency results

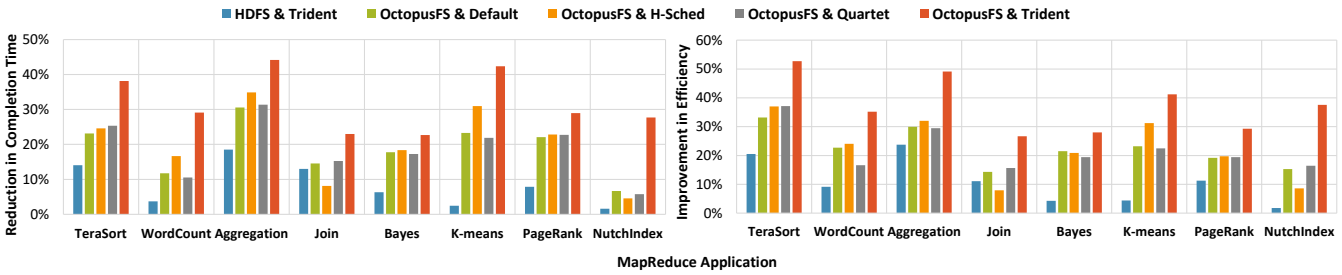

Figure 8: (A) Percent reduction in completion time and (B) percent improvement in cluster efficiency compared to Default Scheduler over HDFS for the HiBench MapReduce applications run using the *large* data scale.

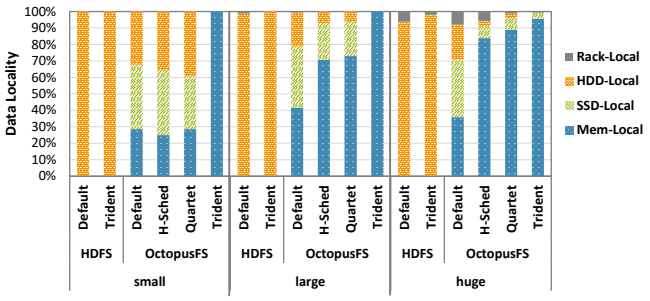

Figure 9: Data locality rates for all schedulers over the two file systems for the HiBench Spark workload.

have the same trends with the reductions in completion times discussed above, but interestingly the magnitude of the gain is higher. The reason is twofold. First, the jobs are executed as a set of parallel tasks. Even if a large fraction of the tasks consume less resources via avoiding disk I/O, the remaining tasks may delay the overall job completion. Second, the job completion time also accounts for CPU processing as well as the output data generation, both of which are independent of the input I/O [21].

The results for the *small* and *huge* data scale are similar in trend and omitted due to space constraints. The main difference is the magnitude in gains, which is typically lower for the *small* scale and higher for the *huge* scale (compared to the *large* scale) for all schedulers. The highest reduction in completion time was recorded for the *huge* Aggregation job using the Trident Scheduler at 57%.

6.3 Spark Evaluation with HiBench

The evaluation with the HiBench workloads was repeated on Spark in the same manner as on Hadoop (described in Section 6.2), with the exception of NutchIndex, which is not implemented for Spark. We used Spark's *Standalone Cluster Manager* for allocating resources across applications, which is widely used in practice [14]. Each application received one Executor process on each worker node, while the Driver process was executed on the Master node.

The overall data locality rates for all schedulers for the HiBench Spark workload are shown in Figure 9. Our first key observation is that, unlike Hadoop, the Spark Default Scheduler is able to achieve over 94% data locality across all data scales. There are two reasons explaining this behavior. First, each application has available resources on all nodes, and hence, can selectively choose which ones to use for the task assignments (especially for smaller jobs), unlike MapReduce that gets resources on some nodes based on containers allocated from YARN. Second, the Default Scheduler has a built-in load balancing feature that iterates the available resources on each node one slot at a time, which increases the opportunities for data-local assignments (or memory-local in the case of H-Scheduler and Quartet). This is also evident by how both the H-Scheduler and Quartet are able to achieve high memory locality rates of over 71% and 84% for *large* and *huge* applications, respectively. Trident, however, is able to achieve 100% memory locality for both *small* and *large* applications, as well as 96% memory locality for *huge* applications, *significantly outperforming all other schedulers*. Finally, note that process-local tasks are all assigned in a separate process, before the other tasks are assigned; hence, the percent of process-local tasks is the same for all schedulers over both file systems.

In terms of reduction in completion time, the overall trends are similar as in the case of running the applications on Hadoop, and are shown in Figure 10 for the *large* and *huge* data scales. In particular, the H-Scheduler and Quartet are able to offer good performance improvements over the Default Scheduler because they are able to exploit the storage tier information, but are still outperformed by Trident in all cases. The magnitude of gains for the *large* iterative applications (i.e., Bayes, K-means, and PageRank) are lower for Spark compared to Hadoop because Spark will cache the output data from the first iteration in memory and then use process-local tasks for the following iterations. Hence, the gains from memory-local task assignments only impact the first iteration. Even then, in the case of *huge* Bayes and K-means, Trident is able to speedup their first iteration by 4×, leading to an overall application speedup of over 2×. Spark's PageRank does not enjoy such benefits because its first iteration is very CPU intensive, thus limiting the I/O gains from memory-locality. Another interesting observation is that, unlike with Hadoop, Quartet is able to outperform H-Scheduler in Spark by 4% on average in most cases, because its delay scheduling approach is actually able to improve memory-locality rates by 2%-5%. Finally, in the case of the *huge* workload, *Trident is able to significantly improve performance for most applications, reaching up to 66% reduction in completion time, i.e.,* 3× *speedup.*

6.4 Time Needed for Scheduling

Finally, we evaluate Trident's scheduling time as we vary the number of tasks n ready for execution and the number of cluster nodes r with available resources. For this purpose, we instantiate a Spark

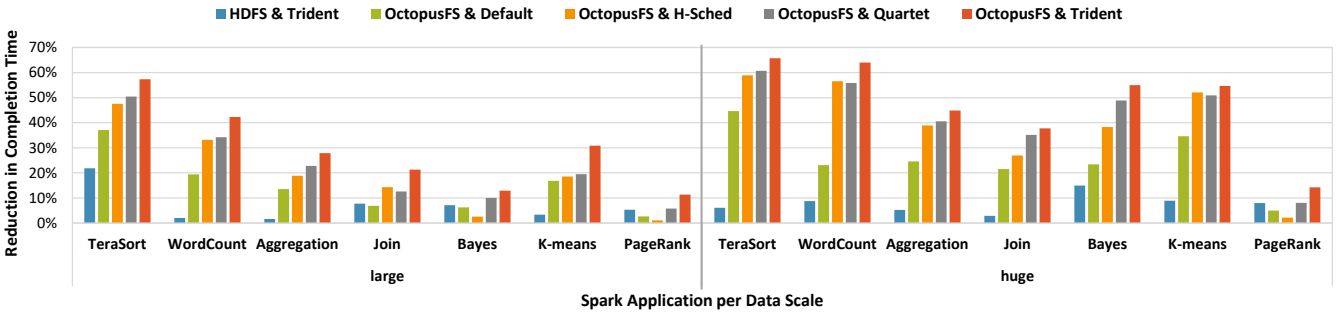

Figure 10: Percent reduction in completion time compared to Default Scheduler over HDFS for the HiBench Spark applications run using the *large* and *huge* data scales.

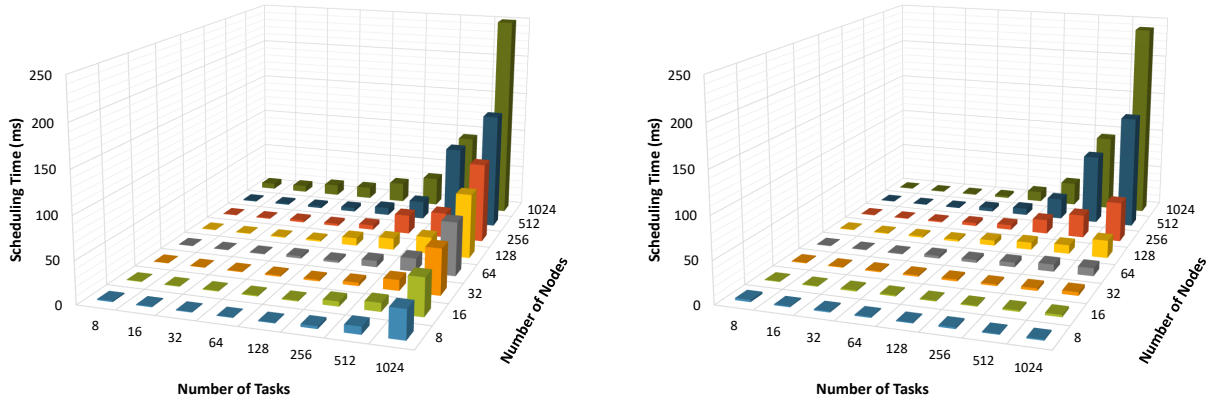

Figure 11: Trident's scheduling time in Spark Driver when the two pruning algorithms are (A) disabled or (B) enabled.

Manager and register r virtual nodes, each with one Executor. Next, we submit a Spark application with one stage of n tasks. Each task has a list of 3 preferred locations (i.e., ⟨node, tier⟩ pairs) in random nodes and tiers across the cluster. Finally, we measure the actual time needed by Trident to make all possible task assignments. This time also includes updating all relevant internal data structures maintained by the Spark Driver.

Figure 11 shows the scheduling times as we vary both n and r between 8 and 1024 (note the logarithmic scale of both axes), when our two pruning algorithms are either disabled or enabled. With pruning, as long as one of the two dimensions (i.e., tasks or nodes) is small, the scheduling time is very low and grows linearly. For example, with up to 64 tasks, the scheduling time is below 2ms regardless the cluster size, whereas it can reach 20ms without pruning for 1024 nodes (i.e., there is an order of magnitude reduction). Similarly, large jobs ($n \geq 256$) get scheduled quickly in under 3ms in small clusters ($r \leq 32$), whereas scheduling time can reach 54ms without pruning. The scheduling time increases non-linearly only when both dimensions are high, since pruning cannot help. However, even in the extreme case of scheduling 1024 tasks on a 1024-node cluster, the scheduling time is only 240ms. More importantly, this overhead is incurred by the Spark Driver (or the MapReduce Application Master in Hadoop) and not the cluster, and is minuscule compared to both the total execution time of such a large job and the potential performance gains from Trident's scheduling abilities.

We repeated this experiment in Hadoop and the scheduling times in MapReduce AM are very similar to the ones observed for Spark. However, Trident's scheduling times in YARN's Resource Manager are much lower, as they are governed by Algorithm 3. Specifically, in the case of 1024 tasks × 1024 nodes, Trident's scheduling time is 61ms as opposed to 60ms for FIFO and 162ms for Capacity (extra time due to updating queue statistics after each assignment), highlighting the negligible overheads induced by our approach.

7 CONCLUSION

The advent of tiered storage systems has introduced a new dimension in the task scheduling problem in cluster computing. Specifically, it is important for task schedulers to consider both the locality and the storage tier of the accessed data when making decisions, in order to improve application performance and cluster utilization. In this paper, we propose Trident, a new scheduling approach that casts the task scheduling problem into a minimum cost maximum matching problem in a bipartite graph, which enables Trident to efficiently find the optimal solution. We have implemented Trident in both Hadoop and Spark, showcasing the generality of the approach in scheduling tasks for two very different platforms. Our experimental evaluation with real-world workloads and industry-validated benchmarks demonstrated that Trident, compared to state-of-the-art schedulers, can maximize the benefits induced by tiered storage and significantly reduce application execution time.

REFERENCES

[1] Cristina L Abad, Yi Lu, and Roy H Campbell. 2011. DARE: Adaptive Data Replication for Efficient Cluster Scheduling. In *Proc. of the 2011 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*. IEEE, 159–168.

[2] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. 2012. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 61–74.

[3] Alluxio 2021. *Alluxio: Data Orchestration for the Cloud.* Retrieved May 5, 2021 from http://www.alluxio.org/

[4] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. 2011. Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters. In *Proc. of the 6th European Conf. on Computer Systems (EuroSys)*. ACM, 287–300.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2011. Disk-locality in Datacenter Computing Considered Irrelevant. In *Proc. of the 13th Workshop on Hot Topics in Operating Systems (HotOS)*. USENIX, 12–17.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX, 267–280.

[7] Apache Hadoop 2021. *Apache Hadoop.* Retrieved May 5, 2021 from https://hadoop.apache.org

[8] Apache Spark 2021. *Apache Spark.* Retrieved May 5, 2021 from https://spark.apache.org

[9] Quan Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. 2010. SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment. In *Proc. of the 10th IEEE Intl. Conf. on Computer and Information Technology (ICCIT)*. IEEE, 2736–2743.

[10] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *PVLDB* 5, 12 (2012), 1802–1813.

[11] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The Case for Evaluating MapReduce Performance using Workload Suites. In *Proc. of the 2011 IEEE Intl. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 390–399.

[12] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce Performance in Heterogeneous Environments with Adaptive Task Tuning. In *Proc. of the 15th IEEE Intl. Conf. on Cluster Computing (CLUSTER)*. ACM, 97–108.

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

[14] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. 2016. Quartet: Harmonizing Task Scheduling and Caching for Cluster Computing. In *Proc. of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, 1–5.

[15] Ran Duan and Seth Pettie. 2014. Linear-time Approximation for Maximum Weight Matching. *Journal of the ACM (JACM)* 61, 1 (2014), 1–23.

[16] Avrilia Floratou, Nimrod Megiddo, Navneet Potti, Fatma Özcan, Uday Kale, and Jan Schmitz-Hermes. 2016. Adaptive Caching in Big SQL using the HDFS Cache. In *Proc. of the 7th ACM Symp. on Cloud Computing (SoCC)*. ACM, 321–333.

[17] Rohan Gandhi, Di Xie, and Y Charlie Hu. 2013. PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters. In *Proc. of the 2013 USENIX Annual Technical Conference (ATC)*. USENIX, 61–66.

[18] Kannan Govindarajan, Supun Kamburugamuve, Pulasthi Wickramasinghe, Vibhatha Abeykoon, and Geoffrey Fox. 2017. Task Scheduling in Big Data-Review, Research Challenges, and Prospects. In *Proc. of the 9th Intl. Conf. on Advanced Computing (ICoAC)*. IEEE, 165–173.

[19] GridGain 2021. *GridGain In-Memory Computing Platform.* Retrieved May 5, 2021 from http://www.gridgain.com/

[20] HDFS 2020. *HDFS Archival Storage, SSD & Memory.* Retrieved May 5, 2021 from https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html

[21] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *PVLDB* 13, 1 (2019), 43–56.

[22] HiBench 2020. *HiBench Suite.* Retrieved May 5, 2021 from https://github.com/intel-hadoop/HiBench

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX, 295–308.

[24] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2011. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *New Frontiers in Information and Software as Services*. Springer, 209–228.

[25] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. of the 22nd ACM Symp. on Operating Systems Principles (SOSP)*. ACM, 261–276.

[26] Jingjie Jiang, Shiyao Ma, Bo Li, and Baochun Li. 2016. Symbiosis: Network-aware Task Scheduling in Data-parallel Frameworks. In *Proc. of the 35th IEEE Intl. Conf. on Computer Communications (INFOCOM)*. IEEE, 1–9.

[27] Elena Kakoulli and Herodotos Herodotou. 2017. OctopusFS: A Distributed File System with Tiered Storage Management. In *Proc. of the 2017 ACM Intl. Conf. on Management of Data (SIGMOD)*. ACM, 65–78.

[28] KR Krish, Ali Anwar, and Ali R Butt. 2014. hatS: A Heterogeneity-aware Tiered Storage for Hadoop. In *Proc. of the 14th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 502–511.

[29] Sparsh Mittal and Jeffrey S Vetter. 2015. A Survey of Software Techniques for using Non-volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 5 (2015), 1537–1550.

[30] Seyed Reza Pakize. 2014. A Comprehensive View of Hadoop MapReduce Scheduling Algorithms. *International Journal of Computer Networks & Communications Security* 2, 9 (2014), 308–317.

[31] Fengfeng Pan, Jin Xiong, Yijie Shen, Tianshi Wang, and Dejun Jiang. 2018. H-scheduler: Storage-aware task scheduling for heterogeneous-storage spark clusters. In *Proc. of the 24th IEEE Intl. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE, 1–9.

[32] Mario Pastorelli, Damiano Carra, Matteo Dell'Amico, and Pietro Michiardi. 2015. HFSP: Bringing Size-based Scheduling to Hadoop. *IEEE Transactions on Cloud Computing* 5, 1 (2015), 43–56.

[33] Aparna Raj, Kamaldeep Kaur, Uddipan Dutta, V Venkat Sandeep, and Shrisha Rao. 2012. Enhancement of Hadoop Clusters with Virtualization Using the Capacity Scheduler. In *Proc. of the Third Intl. Conf. on Services in Emerging Markets*. IEEE, 50–57.

[34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proc. of the 26th Intl. Conf. on Massive Storage Systems and Technology (MSST)*. IEEE, 1–10.

[35] Mbarka Soualhia, Foutse Khomh, and Sofiène Tahar. 2017. Task Scheduling in Big Data Platforms: A Systematic Literature Review. *Journal of Systems and Software* 134 (2017), 170–189.

[36] Xiaoyu Sun, C. He, and Ying Lu. 2012. ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm. In *Proc. of the 18th IEEE Intl. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE, 148–155.

[37] SWIM 2016. *SWIM: Statistical Workload Injector for MapReduce.* Retrieved May 5, 2021 from https://github.com/SWIMProjectUCB/SWIM/wiki

[38] Jian Tan, Xiaoqiao Meng, and Li Zhang. 2013. Coupling Task Progress for MapReduce Resource-aware Scheduling. In *Proc. of the 32nd IEEE Intl. Conf. on Computer Communications (INFOCOM)*. IEEE, 1618–1626.

[39] Zhuo Tang, Min Liu, Almoalmi Ammar, Kenli Li, and Keqin Li. 2016. An Optimized MapReduce Workflow Scheduling Algorithm for Heterogeneous Computing. *The Journal of Supercomputing* 72, 6 (2016), 2059–2079.

[40] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 4th ACM Symp. on Cloud Computing (SoCC)*. ACM, 1–16.

[41] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. 2014. Fresh: Fair and Efficient Slot Configuration and Scheduling for Hadoop Clusters. In *Proc. of the 7th IEEE Intl. Conf. on Cloud Computing (CLOUD)*. IEEE, 761–768.

[42] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. 2014. Map Task Scheduling in MapReduce with Data Locality: Throughput and Heavy-traffic Optimality. *IEEE/ACM Transactions On Networking* 24, 1 (2014), 190–203.

[43] Luna Xu, A. Butt, Seung-Hwan Lim, and R. Kannan. 2018. A Heterogeneity-Aware Task Scheduler for Spark. In *Proc. of the 2018 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*. IEEE, 245–256.

[44] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2009. *Job Scheduling for Multi-User MapReduce Clusters.* Technical Report UCB/EECS-2009-55. EECS Department, University of California, Berkeley. Retrieved May 5, 2021 from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html

[45] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. of the 5th European Conf. on Computer Systems (EuroSys)*. ACM, 265–278.

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. USENIX, 15–28.

[47] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. USENIX, 29––42.