Cost-based Data Prefetching and Scheduling in Big Data Platforms over Tiered Storage Systems

HERODOTOS HERODOTOU, Cyprus University of Technology, Cyprus
ELENA KAKOULLI, Neapolis University Pafos, Cyprus and Cyprus University of Technology, Cyprus

The use of storage tiering is becoming popular in data-intensive compute clusters due to the recent advancements in storage technologies. The Hadoop Distributed File System, for example, now supports storing data in memory, SSDs, and HDDs, while OctopusFS and hatS offer fine-grained storage tiering solutions. However, current big data platforms (such as Hadoop and Spark) are not exploiting the presence of storage tiers and the opportunities they present for performance optimizations. Specifically, schedulers and prefetchers will make decisions only based on data locality information and completely ignore the fact that local data are now stored on a variety of storage media with different performance characteristics. This article presents Trident, a scheduling and prefetching framework that is designed to make task assignment, resource scheduling, and prefetching 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 utilizes two novel pruning algorithms for bounding the size of the graph, while still guaranteeing optimality. In addition, Trident extends YARN's resource request model and proposes a new storage-tier-aware resource scheduling algorithm. Finally, Trident includes a cost-based data prefetching approach that coordinates with the schedulers for optimizing prefetching operations. 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.

CCS Concepts: • **Information systems** → **Hierarchical storage management**; *Distributed storage* • **Theory of computation** → Scheduling algorithms;

Additional Key Words and Phrases: Distributed file systems, tiered storage, data prefetching, task scheduling

ACM Reference format:
Herodotos Herodotou and Elena Kakoulli. 2023. Cost-based Data Prefetching and Scheduling in Big Data Platforms over Tiered Storage Systems. *ACM Trans. Datab. Syst.* 48, 4, Article 11 (November 2023), 40 pages.
https://doi.org/10.1145/3625389

1 INTRODUCTION

Big data applications, such as web-scale data mining, online analytics, and machine learning, are now routinely processing large amounts of data in distributed clusters of commodity hardware [32]. Distributed processing is managed by big data platforms such as Apache Hadoop MapReduce [8] and Spark [10], while cluster resources are managed by cluster managers such as YARN [59] and Mesos [36], as shown in Figure 1. Data are stored on distributed tiered

Authors' addresses: H. Herodotou, Cyprus University of Technology, 30 Arch. Kyprianos Str., Limassol, Cyprus, 3036; e-mail: herodotos.herodotou@cut.ac.cy; E. Kakoulli, Neapolis University Pafos, Pafos, 2 Danais Avenue, Cyprus, 8042 and Cyprus University of Technology, 30 Arch. Kyprianos Str., Limassol, Cyprus, 3036; e-mails: e.kakoulli@nup.ac.cy, elena.kakoulli@cut.ac.cy.

0362-5915/2023/11-ART11
https://doi.org/10.1145/3625389

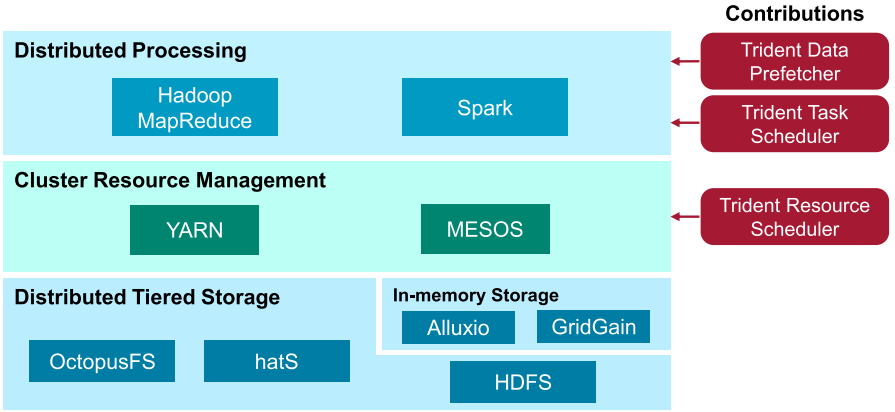

Fig. 1. Ecosystems of big data platforms and contributions.

storage/file systems, where different storage media devices, such as non-volatile random-access memory (NVRAM), solid-state drives (SSDs), and hard disk drives (HDDs), have a variety of capacity and performance capabilities [44]. The **Hadoop Distributed File System (HDFS)** [50] is perhaps the most popular distributed file system used with Hadoop and Spark deployments and now supports storing data in memory and SSD devices, in addition to HDD devices [29]. OctopusFS [40] and hatS [41] 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 [4] and GridGain [24] are used for storing or caching HDFS data in cluster memory.

Tiered storage is advantageous for balancing capacity and performance requirements of data storage and is becoming increasingly important, because the bandwidth of the interconnection network in clusters keeps increasing (and headed toward InfiniBand); hence, the execution time bottleneck of applications is shifting toward the I/O performance of the storage devices [6, 46]. However, current big data platforms are not exploiting the presence of storage tiers and the opportunities they present for optimizing both application performance and cluster utilization. In this work, we identify two such key opportunities: (1) take advantage of the fact that data may be stored on different tiers to optimize task and resource scheduling decisions and (2) proactively prefetch data to higher storage tiers to overlap data transfers with computation.

One of the most important components of big data platforms is their *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 of tasks and resources utilization [23, 51]. A plethora of scheduling algorithms have been proposed in recent years for achieving a variety of complementary objectives such as better resource utilization [39, 48, 57], fairness [38, 60], workload balancing [17, 58], and data management [1, 5, 66]. One of the key strategies toward 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). In the past, scheduling such 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 that each task is reading from.

Data *prefetching* has been used in the past as an effective way to improve data locality by proactively transferring input data to cluster nodes where non-local tasks would be executed [52,

64]. Data prefetching is carried out concurrently with data processing, thereby hiding the latency resulting from network communication or disk operations and improving application performance [25]. When tiering is involved, prefetching can also improve the performance of data-local tasks by proactively loading input data in memory before the task starts execution. Despite its advantages, prefetching poses serious challenges to system designers on where and when to prefetch. First, it is challenging to generate accurate prefetching requests for non-local tasks as that requires predicting where a non-local task will be launched, which is hard to do, as scheduling decisions depend on the current status of the running applications as well as the available cluster resources. Second, it is crucial to ensure the timeliness of prefetching, i.e., that the prefetched data blocks are ready before they are accessed [64]. Otherwise, there is a risk of wasting time and resources by prefetching data that will not be used.

Very few works address (fully or partly) the scheduling and prefetching problems over tiered storage. In the presence of an in-memory caching tier, some systems like PACMan [7], BigSQL [21], and Quartet [19] will simply prioritize scheduling memory-local tasks over data-local tasks (as they assume that only two tiers exist). H-Scheduler [46] is 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. With regards to prefetching, all existing approaches focus only on prefetching input data for non-local tasks. FlexFetch [64] pre-executes the Hadoop MapReduce scheduler ahead of time to predict where and when future non-local tasks will need to read input remotely, while HPSO [52], SADP [13], and CHCDLOS [42] try to predict the most appropriate nodes to which future tasks should be assigned by estimating the completion time of tasks. The existing approaches, unlike ours, do not generalize data prefetching to also optimize data-local tasks and do not coordinate with the scheduler to ensure that prefetched data will be accessed by the scheduled tasks.

In this article, we introduce *a scheduling and prefetching framework that can exploit the storage type information provided by the underlying tiered storage system for making optimal scheduling and prefetching decisions in both a locality-aware and a storage-tier-aware manner*. The framework is modularized in three components as visualized in Figure 1, namely the *Trident Task Scheduler*, *Resource Scheduler*, and *Data Prefetcher*, which together can significantly boost application performance and cluster utilization. More concretely, our *contributions* are as follows:

(1) We formulate the problem of task scheduling over tiered storage as a minimum cost maximum matching problem in a bipartite graph and introduce two pruning algorithms for reducing the scheduling time by up to an order of magnitude without affecting the optimality of the solution.
(2) We extend YARN's cluster resource request model with a general notion of locality preferences to account for storage tiers and propose a new storage-tier-aware resource scheduling algorithm.
(3) We introduce a new storage-tier-aware data prefetching approach that utilizes cost modeling and coordinates with the task and resource schedulers to ensure the timeliness and accuracy of the prefetching operations.
(4) We implemented the Trident scheduling and prefetching components in both Apache Spark and Apache Hadoop, showing the generality and practicality of our approach.
(5) We performed an extensive evaluation using a realistic workload derived from Facebook traces as well as an industry-validated benchmark, showcasing significant benefit for both application performance and cluster efficiency.

This article builds upon our previous work [34] that formally defined the problem of task scheduling over tiered storage and proposed the storage-tier-aware task scheduling approach presented

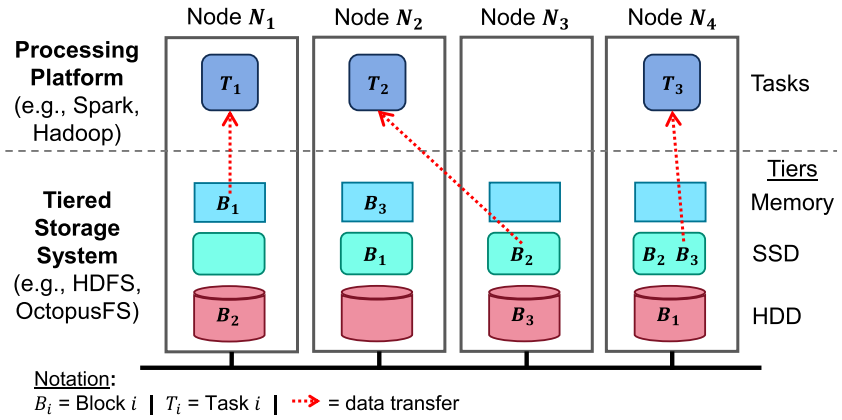

Fig. 2. Example of (i) block replication across cluster nodes and storage media in a tiered storage system and (ii) parallel task execution in a processing platform over tiered storage. Each task T_i processes the corresponding block B_i. Task T_1 is memory-local, T_2 is rack-local, and T_3 is SSD-local.

in Section 2. This article extends to resource scheduling over tiered storage in Section 3 (partly discussed in Reference [34]) and adds a new data prefetching component presented in Section 4, thereby presenting a comprehensive solution to exploiting storage tiers in big data platforms.

Outline. The rest of this article is organized as follows. Sections 2, 3, and 4 present our solutions to the task scheduling, resource scheduling, and data prefetching problems over tiered storage, respectively. Section 5 discusses the implementation details for Spark and Hadoop, while Section 6 reviews prior related work. Section 7 describes the evaluation methodology and results, and Section 8 concludes the article.

2 TASK SCHEDULING OVER TIERED STORAGE

Distributed file systems, such as HDFS [50] and OctopusFS [40], store data as files that are split into large blocks (128 MB 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 storage media of different type. For example, Figure 2 shows how three blocks (B_1–B_3) are stored across four nodes (N_1–N_4), with some replicas residing in memory, SSDs, or HDDs. HDFS and OctopusFS are actively maintaining all block locations at the level of nodes and storage tiers.

Data processing platforms, such as Hadoop and Spark, are responsible for allocating resources to applications and scheduling tasks for execution. In the example of Figure 2, assuming task T_i wants to process the corresponding data block B_i, the task 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 task schedulers only take into account data locality during their decision-making process, we argue (and show) that considering the storage tiers is crucial for taking full advantage of the benefits offered by tiered storage.

2.1 Problem Definition

A typical compute cluster consists of a set of *nodes* $N = \{N_1, \ldots, N_r\}$ arranged in a rack network topology. The cluster offers a set of *resources* $R = \{R_1, \ldots, R_m\}$ for executing tasks. A resource represents a logical bundle of physical resources (e.g., ⟨1 CPU, 2 GB RAM⟩), such as a Container in Apache YARN, an Executor slot in Spark, or a Resource offer in Apache Mesos, and it is bound to a

particular node. Resources are dynamically created based on the availability of physical resources and the requirements of the tasks. 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* $T = \{T_1, \ldots, T_n\}$ 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, then its list of preferred locations contains these three nodes. However, when the data are 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 pre-defined 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 3,200, 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:

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

According to Equation (1), if the location of resource R_j is one of the nodes N_k in T_i's preferred locations, then 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, then 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 in our current cost function. For modern clusters that use Remote Direct Memory Access (RDMA) over commodity Ethernet (RoCEv2) for intra data center communication [26], the preference score c_2 can be set much closer to c_1 and the cost function can be adjusted to also add the tier-based preference score p_k^i. In general, the cost function can be easily adjusted to represent the task-to-resource assignment cost for a particular environment without affecting the task scheduling problem formulation or solution approach.

Using the above definitions, the task scheduling problem can be formulated as a constrained optimization problem:

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

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

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

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

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 constraint guarantees that each task will only be assigned to one resource, while the last constraint

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
$P(T_i)$	List with 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_1	Network transfer cost within a rack
c_2	Network transfer cost across racks
$C(T_i, R_j)$	Cost of scheduling task T_i on resource R_j
d	Default replication factor of the underlying file system

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 $n < m$, then the third constraint 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 constraint 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.

2.2 Minimum Cost Maximum 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,

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

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 constraints 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 3 illustrates an example with three tasks (T_1–T_3) and four available resources (R_1–R_4) located on three distinct nodes (N_1, N_2, N_4). Each task T_i has a list of three preferred locations (i.e., ⟨node, tier⟩ pairs) according to the storage location of the corresponding data block B_i shown in Figure 2. For ease of reference, each tier preference score is indicated using the constants M, S, and H, corresponding to the memory, SSD, and HDD tiers of the underlying storage system, with $M < S < H$. Note that the definition of the preference score in Section 2.1 allows for variable values in case the cluster contains heterogeneous storage devices (e.g., SSDs with different I/O bandwidth on different nodes). The created bipartite graph with seven vertices (three for tasks and four for resources) is also visualized in Figure 3. 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

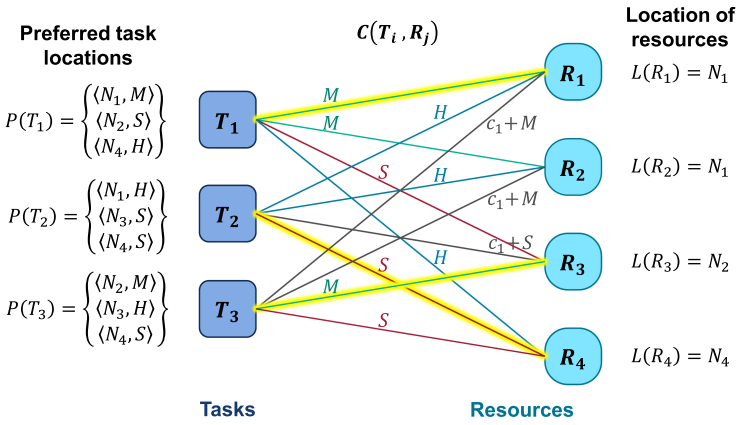

Fig. 3. Example of three tasks with preferred locations, four available resources with their location, and the corresponding bipartite graph. M, S, and H represent the tier preference scores (or costs) for memory-, SSD-, and HDD-local assignments, respectively, while c_1 represents the network transfer cost within the rack. The yellow highlighted edges represent the optimal task assignments.

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 3) 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 [18]. 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 runtime 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 [20]. However, the scheduling time of the overall approach is so low (as evaluated in Section 7.5) that we opted for finding the optimal solution with the Hungarian Algorithm.

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.*

2.3 Excess Resource Pruning Algorithm

Algorithm 1 shows the process of computing and pruning the available resources in a cluster, which will then be used for building the bipartite graph and making the task assignments. 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., containers or slots) for running the

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 then
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) then
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 then
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
```

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 rationale 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, then the list is returned and the process completes (lines 14 and 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. Algorithm 1 is very efficient with a linear complexity of $O(n + m)$, where n is the number of tasks and m is the total number of available resources.

Suppose 3 tasks are ready for execution and their preferred locations are as shown in Figure 4. 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 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

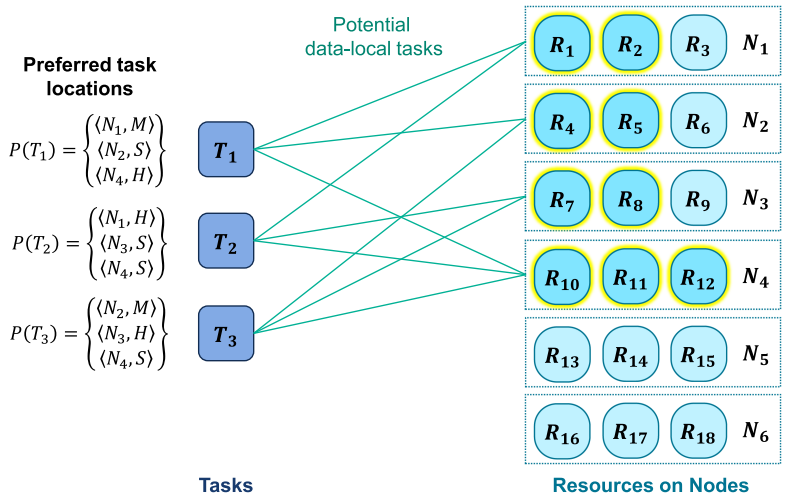

Fig. 4. Example of the excess resource pruning Algorithm 1 with 3 tasks and 18 available resources. Lines connecting tasks with nodes indicate potential data-local tasks. The darker and yellow-highlighted resources represent the selected resources; the others are pruned.

and nm edges without pruning, but only $(1 + d)n$ vertices and dn^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 three tasks (with the same preferred locations) are present but only two slots are available on each of the nodes N_1–N_4. If pruning were to take place (lines 3–15), then all eight slots would be added in the result list, rendering the pruning process pointless. This behavior is expected, since each task will typically have three preferred locations (since each file block has three 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.

2.4 Excess Task Pruning Algorithm

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 rationale 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, then the result list is returned and the process completes (lines 10 and 11). In the opposite case, or when pruning is not enabled, the list with all tasks is returned (line 12). Similarly to Algorithm 1, Algorithm 2 is also very efficient with a linear complexity of $O(n + m)$.

Continuing with the example with the three tasks (recall Figure 3), suppose that only one resource slot is available on node N_2 as shown in Figure 5. In this case, only tasks T_1 and T_3 will

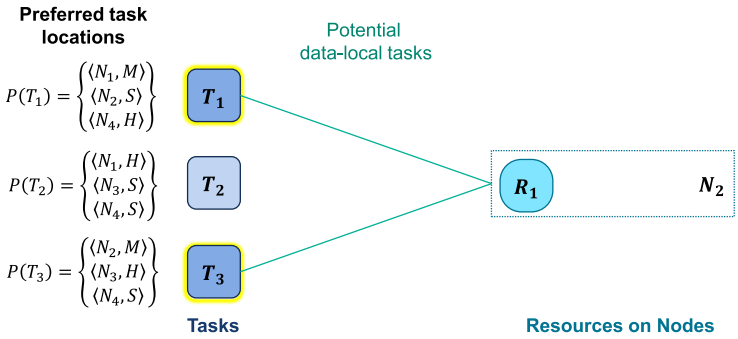

Fig. 5. Example of the excess task pruning Algorithm 2 with three tasks and one available resource. Lines connecting tasks with nodes indicate potential data-local tasks. The darker and yellow-highlighted tasks (T_1, T_3) represent the selected tasks; the other task (T_2) is pruned.

ALGORITHM 2: Prune tasks available for execution

```
1:  procedure PruneTasks(tasks[], resources[])
2:      if tasks.length ≥ d × resources.length then
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)) then
9:                  selectedTasks.add(T_i)
10:         if selectedTasks.length ≥ resources.length then
11:             return selectedTasks                   ▷ Found enough tasks
12:     return tasks                                   ▷ Return all tasks
```

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.

3 RESOURCE SCHEDULING OVER TIERED STORAGE

Distributed processing applications running on Hadoop MapReduce or Spark will typically share cluster resources via negotiating resources with a cluster resource management framework such as Apache Hadoop YARN [59] or Mesos [36]. In this work, we focus on YARN, as it is widely used by both MapReduce and Spark applications. YARN follows a master/worker architectural design, where a *Resource Manager* is responsible for allocating resources to applications and *Node Managers* running on each cluster node are responsible for managing the user processes on those nodes. Allocated resources have the form of *Containers*, which represent a specification of node resources (e.g., CPU cores, memory).

3.1 Current Resource Scheduling

Figure 6 shows the overall application flow over YARN. When a MapReduce or Spark application submits a job for execution ①, an **Application Master (AM)** is launched ② for managing the job's lifecycle, negotiating resources from the **Resource Manager (RM)**, and working with the **Node Managers (NMs)** to execute and monitor the tasks. First, the AM creates a set of

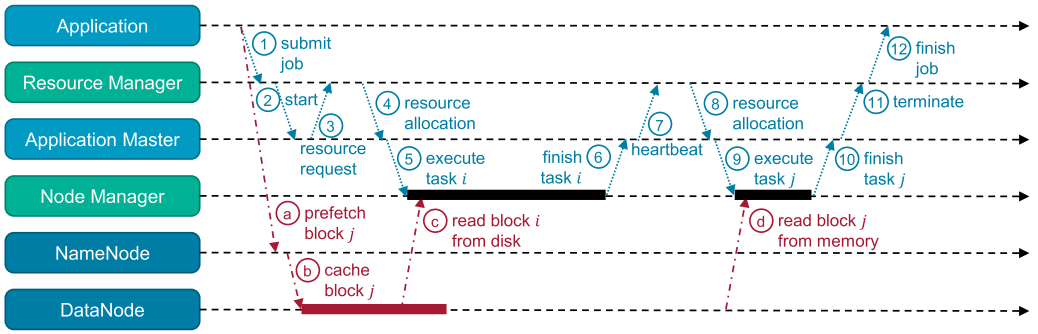

Fig. 6. Overall application flow over time when cluster resources are managed by Hadoop YARN (for either MapReduce or Spark applications). Prefetching flow is also depicted over a tiered-enabled HDFS or OctopusFS.

resource requests ③ based on the input data to process and the tasks to execute. A resource request contains the following [59]:

(1) number of containers;
(2) resources per container (e.g., 1 CPU, 2 GB RAM);
(3) a locality preference (either a host name, a rack name, or '∗'); and
(4) a priority within the application (e.g., map or reduce).

The AM uses heartbeats (every 1 second by default) to send the requests to the RM and to receive the allocated containers. The RM also receives periodic heartbeats from NMs containing their resource availability. Upon a node heartbeat, the RM uses a pluggable *Resource 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., **First In First Out (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 100 ms) based on all available resources in the cluster but do so in the same manner as described above.

Upon a heartbeat from an AM, the RM returns the allocated containers ④. 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. Next, the tasks are sent to the NMs for execution ⑤. When the number of tasks to execute is greater than the number of available containers, multiple waves of parallel tasks will get executed. In this case, whenever some tasks finish execution, the AM is notified ⑥, and upon the next heartbeat with the RM ⑦, it will receive new resource allocations ⑧. The AM will then assign tasks to the allocated containers as before and send them for execution at the NMs ⑨. When all tasks finish execution ⑩, the AM will terminate ⑪, and the application will be notified about the job completion ⑫.

3.2 YARN's Resource Request Model Extension

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.

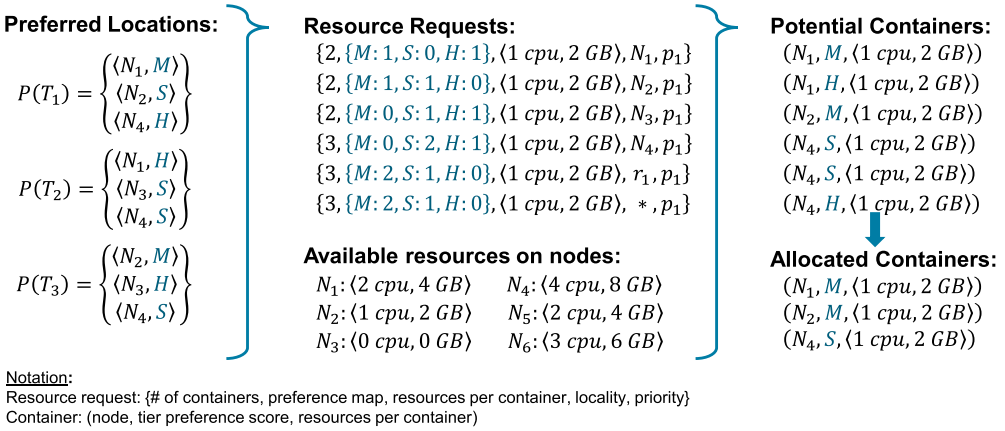

Fig. 7. Example of resource requests created based on preferred locations as well as allocated containers after executing Algorithm 3.

The current Resource Schedulers are oblivious to the underlying tiered storage, since YARN's resource requests do not include any storage tier preferences. To address this issue, we extend the resource request model to include a new *preference map* in the resource request, which maps each tier preference score (recall Section 2.1) to the number of containers requested for that score (i.e, tier). Consider the example in Figure 7 containing three 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 two containers on N_1. The preference map will contain the entries $\{M:1, S:0, H:1\}$, because N_1 is paired with the score M for T_1 and with H for T_2. 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 contains two containers, with preference map $\{M:1, S:1, H:0\}$. This information can then be used by the scheduler for making better decisions. For example, instead of allocating two containers on N_1 (which would lead to two data-local containers, one memory-local, and one HDD-local), it would be better to allocate one container on N_1 and 1 on N_2, which would lead to two memory-local containers.

When computing the preference map for a rack-local resource request, we only count the lowest score (i.e., highest tier) for each task that is paired with nodes belonging to that rack. The rationale 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 in Figure 7, 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 $\{M:2, S:1, H:0\}$. The same process is performed for computing the preference map for the '$*$' (i.e., anywhere) request.

It is important to note that the current resource request model forms a "lossy compression of the application preferences" [59], which makes the communication and storage of requests more efficient, while allowing applications to express their needs clearly [59]. However, the exact preferred locations of the tasks cannot be mapped from the resource requests back to the individual tasks. Hence, the Trident Resource Scheduler will follow a different scheduling approach rather than using the Hungarian Algorithm, described next.

3.3 Storage-tier-aware Resource Scheduling

The Resource Scheduler 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, and (3) which available resources (i.e., containers) to allocate. The first two decisions are subject to various notions or constraints of capacities, queues, fairness, and so on [51]. For example, the Capacity Scheduler will select the first application of the most under-served queue and assign to it resources subject to that queue's limit. Fair Scheduler, however, will select the application that is farthest below its fair share and assign resources up to its share limit. Locality is only taken into consideration during the third decision, when a specific amount of available resources are allocated to a particular application. Hence, the Trident's resource allocation approach, 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 preferences.

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, and so on. 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 node-local resource request S_i^n that references a particular cluster node N_k (lines 4 and 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) as done in Apache YARN. 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.

If the number of potential containers so far is less than the number of needed containers (line 9), then 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 and 16). If the number of collected containers is still below the needed ones, then the remaining potential containers from random nodes are added in the list with assignment cost equal to c_2 (lines 17 and 18).

Due to the aggregate form of the resource requests (recall Section 3.2), 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 for considering all storage tier preferences of the requests. Hence, the last step is to select the *totalConts* containers with the smallest assignment cost from the list of potential containers (lines 19 and 20). The *sort* on line 19 dominates the complexity of Algorithm 3 as $O(klg(k))$, where k is the number of potential containers, which typically equals 3 times the number of requested containers.

The *AddContainers* procedure (also shown 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 particular resource request. The key intuition of the algorithm (lines 22–30) is to assign the lowest cost first as many times as it is 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 three containers are needed. The three corresponding assignment costs to containers will equal M, M, and S.

ALGORITHM 3: Allocate containers in a storage-tier-aware manner

```
1:  procedure AllocateContainers(requests[], maxContainers)
2:      containers = ∅                                          ▷ List of potential containers
3:      totalConts = min{getTotalContainers(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 then
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 then
16:                     break double for loop
17:     if containers.length < totalConts then
18:         add remaining containers to containers with cost c_2
19:     sort(containers)                                        ▷ Sort containers on assignment 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 then
26:             currEntry = iter.next
27:             currCost = currEntry.getKey
28:             currCount = currEntry.getValue
29:         containers.add(Container(node, currCost + rackCost))
30:         currCount = currCount − 1
```

Figure 7 shows a complete example with the resource requests generated based on the tasks preferred locations and the current available resources in a cluster with six nodes. The resource request on N_1 contains two 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 (per the request's preference map). The request on N_2 also asks for two containers but only one can fit there; thus, one container is created with cost M. There are no available resources on N_3 so no containers are created there. Finally, even though four containers can fit on N_4, the corresponding request asks for only three, which leads to the creation of three potential containers with costs S, S, and H. At this point, the list of potential containers contains six entries, which are more than the three total requested containers. Finally, this list is sorted based on increasing assignment cost and the first three containers are allocated to the application, leading to the best resource allocation based on preferred (node and storage tier) locality.

4 COST-BASED DATA PREFETCHING

Data prefetching involves instructing the underlying storage system to proactively move or copy data into memory (or a higher storage tier) so that future application tasks can readily read their data from memory and speedup their execution. In this work, we take advantage of existing APIs

that many distributed file systems offer, including HDFS and OctopusFS, which allow users or applications to cache data into memory [28]. The overall prefetching flow is depicted in Figure 6. During job submission, an application can also submit prefetching requests ⓐ to the file system, and specifically to the *NameNode*, which is the master node of HDFS (and OctopusFS) and responsible for all metadata operations of the file system. The NameNode will then contact the *DataNode(s)* ⓑ that host the block replicas and instruct them to cache the requested block(s). When a task is scheduled for execution, it will read its input data directly from the DataNode. If the input data are not stored in the cache, then it will be read from the storage media storing the data ⓒ; otherwise, it will be read from the cache ⓓ. As observed in Figure 6, data prefetching is carried out concurrently with task execution and other scheduling activities, leading to better resource utilization. In addition, by fetching input data into memory in advance, the execution time of a job can be reduced effectively.

The key challenge in data prefetching is in deciding which data to prefetch to ensure that the prefetched data will indeed be read by the future application tasks. This decision depends on various factors, including the location (both node and tier) of the input data, the current cluster resources availability, as well as the decisions of the schedulers. Trident addresses this challenge by employing a three-pronged approach. First, Trident utilizes cost models for simulating the prefetching operations as well as the parallel execution of tasks on the cluster available resources. These models enable Trident to investigate the impact of prefetching specific data to the job execution and are described in Section 4.1. Second, Trident employs a novel algorithm that (i) identifies several candidate sets of files for prefetching and (ii) utilizes the models to find the set of files to prefetch that will minimize the execution time of the job (presented in Section 4.2). Finally, all relevant prefetching information is relayed to the task and resource schedulers so that they can make informed decisions when scheduling current and future tasks, thereby maximizing the use of prefetched data. This aspect is discussed in both Sections 4.1 and 4.2 when appropriate.

4.1 Modeling of Data Prefetching and Task Execution

The models presented in this section can be used to determine the performance of a job when a specific set of tasks is planned for execution over a specific set of available cluster resources and a specific set of input data is planned to be prefetched. For this purpose, two models are developed: one that simulates the prefetching of input data on the underlying file system and one that simulates the parallel execution of tasks on available cluster resources. The models' novelty, efficiency, and accuracy come from how they use a mix of simulation and cost-based estimation at the level of individual input data blocks and tasks.

Algorithm 4 presents the prefetching model. The input to the model is a list of blocks to prefetch into memory from specific source storage devices. This model will set the time when each block will become available in the cache. Given that prefetching happens using concurrent I/O threads that start about the same time, prefetching more than one block per source device will lead to disk contention. Hence, unlike past modeling works (e.g., References [52, 64]), our model takes disk contention into consideration. The first step is to group the blocks based on the source device (line 2). Then, for each device, the blocks to be prefetched are retrieved and sorted on increasing block length (lines 3 and 4). Suppose that n blocks (*numBlocks* on Algorithm 4) will be prefetched from a specific device (line 5). At first, the n blocks will share the I/O bandwidth of the device equally. The block with the smallest length, b_1, is expected to complete prefetching first. The time needed for b_1's prefetching will equal its length divided by its share of the bandwidth, i.e., total bandwidth BW divided by n (line 10). During this time, the other blocks have made equal progress toward completion. Thus, the block with the second smallest length, b_2, is expected to complete next, but its remaining length to be prefetched (line 9) will only compete with $(n - 1)$ I/O threads

ALGORITHM 4: Model the prefetching of input data blocks

```
1:  procedure ModelPrefecthing(blocksToPref[])
2:      blocksPerDevice = groupBySourceLoc(blocksToPref)          ▷ Map device to blocks
3:      for each blocksList in blocksPerDevice.values() do
4:          sort(blocksList)                                      ▷ Sort on increasing block length
5:          numBlocks = blocksList.size()
6:          prevLength = currLength = 0
7:          prevCacheTime = currCacheTime = 0
8:          for each block in blocksList do                       ▷ Compute cache time for each block
9:              currLength = block.length − prevLength
10:             currCacheTime = prevCacheTime + (numBlocks × currLength) / BW[block.cacheSourceLoc.type]
11:             block.cacheTime = TimeToInitCache + currCacheTime
12:             prevLength = block.length
13:             prevCacheTime = currCacheTime
14:             numBlocks = numBlocks − 1
```

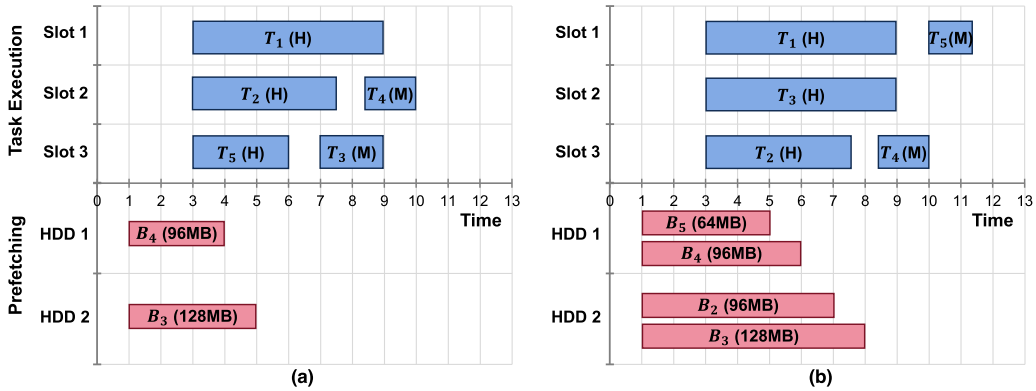

Fig. 8. Two examples of modeling prefetching (Algorithm 4) and task execution (Algorithms 5) when using maximum caching degree of parallelism equal to (a) 1 and (b) 2. Each task T_i is planned to process the corresponding block B_i. All blocks are stored on HDDs with bandwidth equal to 32 MB/s. Block sizes: B_1 = 128 MB, B_2 = 96 MB, B_3 = 128 MB, B_4 = 96 MB, B_5 = 64 MB.

(line 14). Hence, the total time needed for b_2's prefetching will equal the time to prefetch b_1 plus the time to prefetch its remaining length (with the bandwidth divided by $n - 1$). The algorithm continues in a similar fashion for all remaining blocks (lines 8–14) until it sets the cache times for all prefetched blocks. Note that in practice, the device bandwidth might not be shared equally all the time across concurrent readers due to operating system scheduling. However, the data blocks read are typically very large (up to 128 MB), smoothing out any variations in I/O times across the readers. While linearly dividing the bandwidth among concurrent I/O threads is a reasonable simplifying assumption in our work, this part can be easily extended with a more expressive storage model that captures modern devices, such as the Parametric I/O Model [47]. Finally, the model also accounts for a constant initialization time (*TimeToInitCache*) that represents the time needed for scheduling prefetching requests and depends on the heartbeat intervals used between the file system components (defaults to 1 second).

The bottom half of Figure 8 shows two examples of modeling prefetching. In Figure 8(a), only one block is prefetched per storage device. For instance, suppose block B_3 has length 128 MB

and the device's bandwidth is 32 MB/s. Hence, it will take 4 seconds for caching to complete. In Figure 8(b), two blocks are prefetched from the second storage device; B_2 with length 96 MB and B_3 with length 128 MB. Thus, the effective bandwidth for prefetching B_2 is 16 MB/s, and B_2 will take 6 seconds to be prefetched. At the same time, the first 96 MB of B_3 will also be prefetched. After that, the remaining 32 MB of B_3 will utilize the full bandwidth and prefetching will complete after an additional 1 second.

Algorithm 5 shows the modeling of parallel task execution on the available cluster resources. The input consists of a set of tasks submitted for execution (each to process an input data block), information about the available cluster resources, and a delay time parameter that will be explained at the end of the section. This method will set the expected start time and end time of each task based on the available resources and the storage type to read from. The number of tasks for execution is often larger than the number of available resource slots, causing the tasks to be executed in multiple waves. To account for this fact, a min priority queue of slots is created to keep track of when each slot is becoming available for hosting the next task for execution (line 2). In addition, the tasks are also placed in a min priority queue and the priority key is set to the tier preference score (recall Section 2.1) to simulate the presence of a tier-storage-aware task scheduler. It is important to note that a task planned to process a data block that is scheduled for prefetching (but prefetching has not completed yet) will get a tier preference score higher than the slowest tier but lower than the rack-local cost (for example, a score between the HDD score H and the network transfer cost c_1). Once prefetching completes, the preference score will be adjusted to reflect the memory tier. In this manner, a task scheduler will prioritize scheduling other data-local tasks first and leave the tasks with prefetched data to get scheduled later, after prefetching has completed.

For each task (taken in order of the tier preference score) and for each available slot (lines 4 and 5), Algorithm 5 will first compute the start time of the task (lines 6–9). The start time depends on the end time of the previous task executed on the current slot (if any) plus some constant times that represent the job initialization time (*TimeToInit*) and scheduling time (*TimeToSchedule*). Both constants are set based on the heartbeat intervals of the platform's components and are set by default to 2 and 1 seconds, respectively. Next, the algorithm computes the task's expected I/O duration (lines 10–13). If the task will start execution after the caching of its block is completed, then the task will read the block from memory; otherwise, it will read the block from its base storage device. For calculating the CPU processing time of a task, a simple linear regression model is utilized as a function of the input data size, which is built based on a sample of previous task execution logs (line 14). More advanced models for estimating the task processing time, such as the ones proposed in References [14, 31, 54], can be easily plugged in. The current slot is then updated with the tasks' end time and placed back in the priority queue of slots to be used in future iterations of the loop (lines 15 and 16). Finally, the end time of the last task to complete is returned as a proxy to the job execution time.

Figure 8(a) shows an example of modeling the execution of five tasks (T_1–T_5) on a node with three available slots, while two blocks (B_3, B_4) are being prefetched. In this example, each task T_i is planned to process the corresponding block B_i, and all five blocks are stored on HDDs. Initially, the tasks T_1, T_2, and T_5 are scheduled for execution on the three available slots, because their tier preference score is lower than the score of T_3 and T_4 (as there are pending prefetching operations for the latter two tasks). In this manner, the scheduling of T_3 and T_4 is correctly deferred until slots become available, allowing prefetching to complete. Once T_5 ends and the third slot becomes available, T_3 is scheduled and processes its data from memory. Similarly, T_4 is finally scheduled after T_2 completes execution. Algorithm 5 will return the time 10, as that is the end time of the last task to complete (T_4).

ALGORITHM 5: Model parallel execution of tasks on a cluster

```
1:  procedure ModelExecution(tasks[], clusterInfo, delayTime)
2:      slotsQueue = PriorityQueue(clusterInfo.availSlots, 0)        ▷ Priority: task end time
3:      tasksQueue = PriorityQueue(tasks, 0)                         ▷ Priority: tier preference score
4:      for each task in tasksQueue.poll() do
5:          slot = slotsQueue.poll()
6:          if slot.prevTaskEndTime == 0 then
7:              task.startTime = delayTime + TimeToInit + TimeToSchedule
8:          else
9:              task.startTime = slot.prevTaskEndTime + TimeToSchedule
10:         if task.startTime > task.block.cacheTime then
11:             ioTime = task.block.length / BW[Memory]
12:         else
13:             ioTime = task.block.length / BW[task.block.baseType]
14:         task.endTime = task.startTime + ioTime + cpuTime(task)
15:         slot.prevTaskEndTime = task.endTime
16:         slotsQueue.push(slot)
17:     return getLastTaskEndTime(tasks)                             ▷ Return end time of last task
```

The final parameter in Algorithm 5 is a delay time parameter, which can be added for delaying the submission of the job, and thus delaying the execution of the first wave of tasks, to give the file system more time to cache the data before the tasks start execution. This new (optional) feature can be very useful in certain scenarios as it can allow some tasks to process prefetched data in time. For example, suppose a fourth slot was available in the example of Figure 8(a). In this case, task T_4 would get scheduled to start at time 3, before the corresponding block completes prefetching. Thus, prefetching B_4 in memory is wasted. However, if the execution of T_4 is delayed by 1 second, then it can start as soon as B_4 completes prefetching and read the data from memory. Whether (and how much) a delay is beneficial for the overall job execution is explored during the data prefetching algorithm presented next.

4.2 Data Prefetching Algorithm

In this work, data prefetching is formulated as a cost-based optimization problem: *Given a job to be run on some input data and cluster resources, find the set of input data blocks to prefetch that will minimize the execution time of the job*. The key idea of our approach is to enumerate different sets of input data blocks to prefetch and then simulate the execution of tasks without and with prefetching those blocks, using the models presented in the previous section. During the enumeration, the algorithm keeps track of the prefetching set that led to the lowest job execution time. However, the total number of possible subsets of input data blocks is exponential and thus very inefficient to enumerate. Instead, the proposed enumeration process is based on two important observations. First, disk contention can significantly prolong the time needed for prefetching data into memory, which in turn increases the probability of scheduling tasks before prefetching completes (and thus wasting prefetching). At the same time, it is important to take advantage of the presence of multiple disks across nodes and prefetch multiple blocks in parallel to maximize the number of tasks that can process prefetched data. Therefore, the enumeration tries to generate the largest set of data blocks to prefetch, while each time respecting a limit on the maximum number of blocks to prefetch from each device, which we call the caching **degree of parallelism (DoP)**.

Algorithm 6 shows the approach for selecting a set of input data blocks to prefetch, while respecting a specified maximum caching DoP. The input consists of a list of input data to consider,

ALGORITHM 6: Select input data blocks to prefetch that respect a caching degree of parallelism

```
1:  procedure SelectBlocksForPrefetching(inputData[], clusterInfo, maxCachingDoP)
2:      numblocksPerDevice = ∅                                  ▷ Map storage device to count
3:      for each input in inputData do
4:          for each block in input.blocks do
5:              bestLoc = null
6:              bestCacheRemain = 0
7:              for each loc in block.locations do
8:                  cacheRemain = computeCacheRemain(clusterInfo, loc.node)
9:                  if bestLoc == null or loc.type < bestLoc.type or
                      (loc.type == bestLoc.type and cacheRemain > bestCacheRemain) then
10:                     bestLoc = loc
11:                     bestCacheRemain = cacheRemain
12:             currCachingDoP = 1 + numblocksPerDevice.get(bestLoc)
13:             if currCachingDoP ≤ maxCachingDoP then
14:                 block.cacheSourceLoc = bestLoc
15:                 numblocksPerDevice.put(bestLoc, currCachingDoP)
16:                 blocksToPref.add(block)
17:     return blocksToPref
```

information about the available cluster resources, and the maximum caching DoP. This algorithm returns the input blocks to prefetch as well as sets the best location for caching each block. For each input data file and for each input block, the algorithm finds the best candidate cache location, defined the location with the lowest storage type and the highest cache remaining space among the locations hosting the block replicas (lines 5–11). The rationale for this decision, which is also used by HDFS and OctopusFS when caching data, is twofold: (1) to maximize the benefits from prefetching (by caching data located in low tiers) and (2) to balance the use of the cache across the cluster nodes. If the candidate cache location respects the maximum caching DoP, then it is set as a source location for caching the block, while the block is added to the list of blocks to prefetch (lines 12–16).

Algorithm 7 outlines the overall data prefetching algorithm. The input consists of a list of input data to process, information about the available cluster resources, and the job configuration. The output is a list of the best blocks to prefetch and a potential delay time for optimizing the job execution time. The first step is to generate the list of tasks for execution based on the input data and the job configuration (line 2). Typically, each task will process one input data block, where each block is replicated across multiple nodes and storage tiers. The execution of tasks is then modeled using Algorithm 5 without prefetching any data to establish a baseline of execution (line 3). Next, the algorithm enters a loop, where each time the maximum caching DoP increments by 1 (to try all relevant maximum caching DoP starting from 1), until a termination condition is met (lines 5 and 6). The loop will terminate when either all input blocks are prefetched or when prefetching any more blocks will not lead to a reduced job execution time. In each iteration, the algorithm selects a set of blocks to prefetch that respects the maximum caching DoP using Algorithm 6, models the prefetching of those blocks using Algorithm 4, and then models the task execution with prefetching using Algorithm 5 (lines 7–9). In addition, if the optional delay time feature is enabled, then the algorithm computes the execution delay time so that all tasks can start execution after prefetching has completed, and then models the task execution with that delay (lines 10–12). Finally, if prefetching with or without delay leads to a lower job execution time, then that particular set of input blocks to prefetch as well as the computed delay time are maintained (lines 13–16) and returned at the end of the algorithm (line 17).

ALGORITHM 7: Data prefetching algorithm

```
1:  procedure PrefetchData(inputData[], clusterInfo, conf)
2:      tasks = GenerateTasks(inputData, conf)
3:      bestJobTime = ModelExecution(tasks, clusterInfo, 0)          ▷ Without prefetching
4:      bestBlocksToPref = ∅
5:      while (checkTerminationCondition() == false) do
6:          maxCachingDoP = maxCachingDoP + 1
7:          blocksToPref = SelectBlocksForPrefetching(inputData, clusterInfo, maxCachingDoP)
8:          ModelPrefetching(blocksToPref)
9:          jobTimePref = ModelExecution(tasks, clusterInfo, 0)      ▷ With prefetching
10:         if conf.isDelayTimeEnabled then
11:             delayTime = max{0, task.block.cacheTime − task.startTime | ∀ task ∈ tasks}
12:             jobTimePrefDelay = ModelExecution(tasks, clusterInfo, delayTime)
13:         if jobTimePref < bestJobTime or jobTimePrefDelay < bestJobTime then
14:             bestBlocksToPref = blocksToPref
15:             bestJobTime = min(jobTimePref, jobTimePrefDelay)
16:             bestCacheDelay = (jobTimePrefDelay < jobTimePref) ? cacheDelay : 0
17:     return ⟨bestBlocksToPref, bestCacheDelay⟩
```

Figure 8 shows an example with two iterations of the main loop of Algorithm 7, when max caching DoP is set to 1 and 2. In the first case, only one block from each HDD device is prefetched (B_3 and B_4) and the two corresponding tasks (T_3 and T_4) are able to read the data from memory, speeding up their execution, as well as the overall execution of the job. In the second case, four blocks are prefetched (B_2–B_5), two per disk. Since there are only three slots available to run the five tasks, two tasks (namely T_2 and T_3) will start execution before the corresponding prefetching completes and thus not benefit from it. The other two tasks (T_4 and T_5) will process prefetched data from memory, but the overall job execution suffers. Prefetching the fifth block when max caching DoP is set to 3 will also be wasteful and will not improve the job execution (not shown in the figure). Note that without prefetching, all five tasks read the data from the HDDs and the total job execution time is 13 seconds. Hence, in this example, prefetching blocks B_3 and B_4 will lead to the lowest job execution time of 10 seconds.

Overall, the Trident Data Prefetcher is used to decide which input data blocks to prefetch into memory for optimizing job execution time. Once the prefetching requests are submitted, the underlying storage system keeps track of both pending and completed prefetch operations. This information is exposed to higher-level platforms, including the Trident Task Scheduler, which will take it into consideration during scheduling, by adjusting the tier preference scores as discussed in Section 4.1. The Trident Resource Scheduler is also informed through the preference map of the extended resource request model presented in Section 3.2. Therefore, instead of trying to predict how a scheduler will behave for making prefetching decisions (like previous works), our approach uses cost modeling for identifying the best input data blocks to prefetch and then uses the schedulers for making appropriate scheduling decisions.

5 TRIDENT IMPLEMENTATION

This section provides the implementation details of how Trident was implemented in both Apache Hadoop and Spark, while emphasizing a few noteworthy points.

5.1 Trident Implementation in Hadoop

As discussed in Section 3, 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 Resource Scheduler* is implemented as a pluggable component overriding the scheduling interface provided by Hadoop YARN and is running in the RM for making storage-tier-aware allocation decisions using Algorithm 3. YARN's resource request model is also extended to include the preference map, as presented in Section 3.2.

Once the MapReduce AM receives a set of allocated containers, it needs to assign tasks to them. The *Trident Task Scheduler* replaces the default task scheduler in the AM for making optimal storage-tier-aware assignments. In particular, Trident builds a 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 2.2. 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 the containers are allocated. Next, Trident employs 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 task scheduler.

Finally, the data prefetching algorithm discussed in Section 4.2 is implemented by the *Trident Data Prefetcher*, which runs in the MapReduce Client and is invoked as soon as a MapReduce job is ready to be submitted for execution, for deciding which input data blocks to prefetch into memory. Two more changes were required for applying the Trident methodology. First, Hadoop was modified to propagate the storage tier information from the input file readers to the schedulers, in the same way node locations are propagated. Second, HDFS and OctopusFS were modified to also expose the locations of the pending cache requests, in addition to the locations storing the physical block replicas. Overall, we added 3,425 lines of Java code to Hadoop.

5.2 Trident Implementation in Spark

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 [10], Hadoop YARN [59], or Mesos [36]) 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 the tasks' output data over the entire duration of the application.

Internally, an application is divided into *jobs*, where each job is a directed acyclic graph of *stages*, and each stage consists of a set of parallel *tasks*. Whenever a stage S is ready for execution (i.e., its input data are available), the *Spark 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.

The *Trident Task Scheduler* is proposed to replace the current task scheduler in the Spark Driver 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 the

pruning Algorithms 1 and 2 to select which resources and tasks to use, builds the bipartite graph as described in Section 2.2, and uses the Hungarian Algorithm for making the optimal task assignments. Similarly to Hadoop, Spark was modified to propagate the storage tier information from the input file readers to the task scheduler. Overall, we added 944 lines of Scala code to Spark.

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.*

6 RELATED WORK

Multiple scheduling algorithms have been proposed in the past and are presented in various comprehensive surveys [23, 51]. In this section, we discuss the most relevant ones. Hadoop offers three schedulers out-of-the-box: (1) FIFO, which assigns tasks to resources in order of job submission [45]; (2) Capacity, which allocates resources to jobs under constraints of allocated capacities per job queue [49]; and (3) Fair, which assigns resources to jobs such that they get, on average, an equal share of resources over time [27]. 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 [65] 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. [62] focus on striking the right balance between data locality and load balancing using stochastic network theory to simultaneously maximize throughput and minimize delay. Scarlett [5] 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 to correctly identify slow tasks (on less capable nodes) and re-execute them. LATE [66] adopts a static method to compute the progress of tasks, SAMR [14] calculates progress of tasks dynamically using historical information, and ESAMR [54] extends SAMR to employ k-means clustering for generating more accurate estimations. Tarazu [3] and PIKACHU [22] use dynamic load rebalancing to schedule tasks after identifying the fast and slow nodes at runtime. C3 [55] is an adaptive replica selection mechanism implemented for Cloud data stores that utilizes a replica ranking function to prefer faster servers and compensate for slower service times to reduce tail latency. More recently, RUPAM [63] 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 [7], Big SQL [21], and AutoCache [30] 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 [19] 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, whenever it was searching for data-local tasks.

H-Scheduler [46] 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 [46]. 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 7. However, the principled scheduling approach of Trident guarantees that the optimal task assignments (as formalized in Section 2.1) will always be achieved.

While our work focuses on distributed replication-based file systems, other studies have addressed data placement for erasure-coded systems. In particular, one study [43] introduces encoding-aware replication in clustered file systems, which optimizes the placement of replicas to minimize cross-rack downloads during the encoding operation, as well as to preserve availability without data relocation after the encoding operation. WPS [61] is a workload-aware placement scheme designed for erasure-coded in-memory stores. This scheme takes into account the characteristics of the workload, such as access patterns and data popularity, to optimize data placement and migration, with the goal of improving load balancing in the presence of workload skew. Last, EC-Store [2] incorporates dynamic strategies for data access and movement based on workload access patterns within erasure-coded storage systems, with the goal of reducing data retrieval times. These studies highlight the significance of data placement policies tailored for erasure-coded data and are orthogonal to our proposed task and resource scheduling approaches.

Data prefetching mechanisms have been proposed to improve data locality and accelerate non-local tasks. FlexFetch [64] pre-executes the MapReduce scheduler ahead of time to predict the starting time and the node location for future non-local tasks, and then allocates network resources for prefetching data to those nodes. HPSO [52, 53] predicts the remaining execution time of map tasks on Hadoop, which is then used to estimate which resource slot will become idle, and preloads input data to memory on those nodes. Similarly, CHCDLOS [42] estimates map task remaining execution time and then uses predefined rules for prefetching data to target compute nodes ahead of future task executions. Finally, SADP [12, 13] retrieves scheduling information from the computing layer and then uses this information to estimate task completion time and to prefetch and evict data to/from memory. All aforementioned approaches focus explicitly on making predictions for prefetching data that will benefit future non-local task executions. Our approach, however, generalizes data prefetching to optimize both data-local and non-local tasks, and coordinates with the schedulers to ensure that prefetched data will be accessed by the scheduled tasks.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness and efficiency of the Trident schedulers and data prefetcher in exploiting tiered storage 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 7.1).

(2) We investigate the impact of input data size and application characteristics on scheduling using an industry-validated benchmark on both Hadoop (Section 7.2) and Spark (Section 7.3).
(3) We study the impact of data prefetching using both the real-world MapReduce workload from Facebook and the industry-validated benchmark (Section 7.4).
(4) We evaluate the scheduling and prefetching overheads induced by Trident (Section 7.5).

Experimental Setup. The 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, 3.2 GHz CPU, 64 GB RAM, and a 2.1 TB RAID 5 storage configuration. Each Worker node has a 64-bit, 8-core, 2.4 GHz CPU, 24 GB RAM, one 120 GB SATA SSD, and three 500 GB 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 [40], a tiered file system that extends and is backwards compatible to HDFS. OctopusFS was configured to use three storage tiers with 4 GB of memory, 64 GB of SSD, and 3×320 GB of HDD space on each Worker node. The default replication factor is 3 and the default block size is 128 MB for both file systems. Unless otherwise stated, OctopusFS utilizes its default data placement policy, which creates one replica on each of the three storage tiers. The storage tier preference score for memory, SSD, and HDD is set to 1, 8, and 20, respectively, as the measured bandwidth of the three storage media in the cluster is 3,200, 400, and 160 MB/s, respectively. The network transfer cost within a rack (c_1) is set to 40 and across racks (c_2) to 100.

Schedulers. In addition to our *Trident Scheduler*, we implemented two more task schedulers from recent literature, namely *H-Scheduler* [46] and *Quartet* [19] (as described in Section 6), within the MapReduce Application Master and the Spark Driver. When running the Hadoop experiments, the Trident Scheduler consists of its two components, the Trident Task Scheduler and the Trident Resource Scheduler, while H-Scheduler and Quartet were paired with YARN's Capacity Scheduler, as was done in Reference [19]. 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 following three performance metrics are used for the evaluation: (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 job(s) by using less resources compared to the baseline [7, 33]. Specifically, the improvement in cluster efficiency is computed as the percentage difference between the sums of the individual runtime durations of all tasks in a job (or a set of jobs) across two compared scenarios. All results shown are averaged over three repetitions.

7.1 Evaluation of Storage-tier-aware Scheduling 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 [16]. With the traces, we used the SWIM tool [56] 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 [7]. The workload comprises 1,000 jobs scheduled for execution sporadically over 6 hours and processing 92 GB of input data. Hence, the workload exhibits variability in terms of cluster resource usage over time. When the workload starts execution, 380 files already exist on the file system with a total size of 32 GB. The popularity of these files is skewed, as typically observed in data-intensive workloads [7, 15], with a small fraction of the files accessed very frequently, while the rest are accessed less frequently. In particular, these 380 files are accessed by

Table 2. Facebook Job Size Distributions Binned by Data Sizes

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

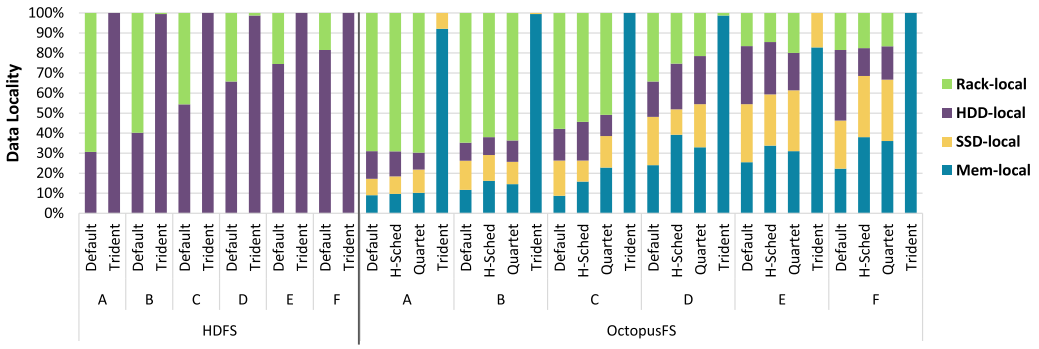

Fig. 9. Data locality rates for all schedulers over the two file systems (HDFS and OctopusFS), broken down into the six Facebook workload bins (A–F).

660 jobs, with 76.8% of the files accessed once and 4.5% accessed more than 5 times. The remaining 340 jobs are accessing 63 input files generated by other prior jobs that executed on the cluster, with a lower popularity skew: 52.4% of the files are accessed once while 25.4% are accessed more than 5 times. The remaining 947 files generated by the workload have a size of 47 GB and are not accessed by the workload. When using OctopusFS, we enabled its Least Recently Used eviction policy [33] so that later jobs in the workload can take advantage of the memory tier (since the aggregate capacity of the memory tier is 40 GB).

To differentiate the effect of 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 [7, 15], the jobs exhibit a heavy-tailed distribution of input sizes. Even though small jobs that process <128 MB 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. However, jobs processing over 1 GB of data account for over 54% of resources and over 68% of I/O. More in-depth workload statistics can be found in Reference [33].

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 (Default, H-Scheduler, Quartet, and Trident). Data prefetching is disabled and will be investigated later in Section 7.4. Figure 9 shows the data locality rates for all schedulers over the two file systems, broken down according to the job bins. With HDFS and the Default Scheduler, there is a clear increasing trend in the percentage of data-local tasks (note that all data are stored on HDDs) as the job size increases. The achieved data locality is low at 30–40% for small jobs (Bins A and 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. In particular, when a new job is submitted in the cluster, there are on average 3.4 jobs and 8.2 tasks already running and processing (i.e., reading

and writing) 1.5 GB of data, while in the worst case, there are 17 jobs and 50 tasks processing 21.0 GB of data. 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, however, considers all available resources together when building the bipartite graph of tasks and resources, 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 9). 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 three reasons explained above), there is also little chance for doing any meaningful storage-tier-aware task assignments. For 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, because the likelihood of finding a task that can be memory-local on a particular node is increased with more 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 four of the six 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 10(a) shows the percentage 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 are stored in OctopusFS as data are 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 for large jobs. The storage-tier-aware schedulers are able to increase the benefits further, depending on the job size. Small jobs (Bins A and 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 and network overheads. The gains in job completion time increase as the input size increases, while different trends across the schedulers are also observed. 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 over OctopusFS, due to the significantly higher memory locality rates it is able to achieve as observed in Figure 9.

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 10(b) 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 10(a) 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*.

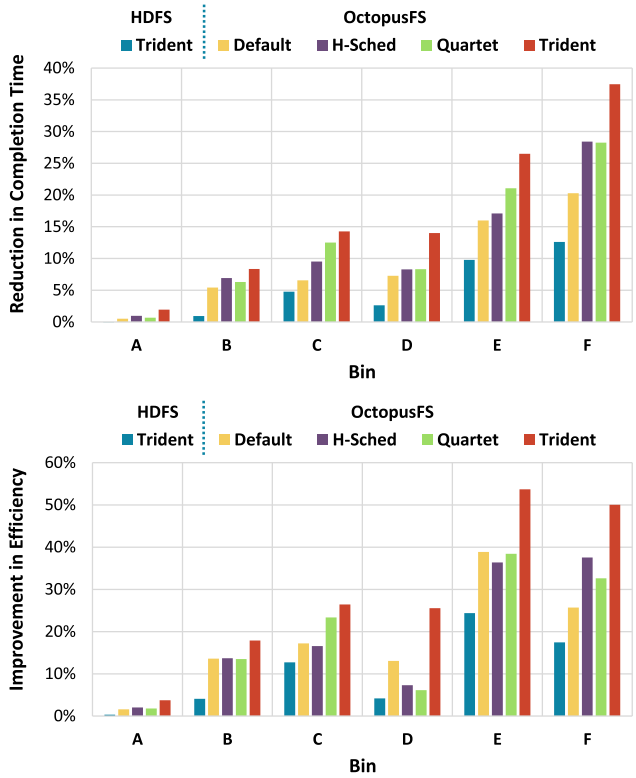

Fig. 10. (a) Percentage reduction in completion time and (b) percentage improvement in cluster efficiency for the Facebook workload compared to Default Scheduler over HDFS.

Even though Trident achieves on average 91% memory-local tasks for the large jobs compared to 35% of the other two storage-tier-aware schedulers, it only yields about 10% additional reduction in completion time and 16% additional improvement in cluster efficiency over them. The reasons behind this discrepancy are as follows. Job completion time depends on the execution time of all the map tasks, the shuffle stage (which in Hadoop happens as part of the reduce task execution), and the reduce task processing. Increased memory locality will primarily reduce the input I/O time of the map tasks, while it will not impact the CPU processing time of the map and reduce tasks. An interesting observation (not shown in the figures) is that the average execution time of reduce tasks in Hadoop also decreases by up to 37% for large jobs due to the improved cluster efficiency, even though their scheduling is not affected by Trident. The high memory-locality rates achieved by Trident reduce local disk I/O and network congestion, which in turn reduce the time needed for data shuffling between map and reduce tasks. Overall, even though the improvements in completion time and cluster efficiency are not proportional to the improvement in memory locality, *Trident is still able to offer the highest improvements compared to the other state-of-the-art approaches.*

Key takeaways. Storage-tier-aware scheduling with Trident leads to almost 100% data locality rates and very high memory-locality rates (>83%) across all job sizes, while the second best scheduler is able to achieve only up to 86% data-locality and 39% memory-locality rates. In addition, Trident consistently provides the highest benefits in terms of application completion time and cluster efficiency across all scenarios studied.

Table 3. HiBench Applications with Primary Characteristics That Dominate Their Execution Time

Category	Application	CPU	I/O	Network
Micro	TeraSort		✓	✓
	WordCount	✓		
OLAP	Aggregation		✓	
	Join	✓		
ML	Bayes	✓		✓
	K-means	✓	✓	
Web Search	PageRank	✓		✓
	NutchIndex			✓

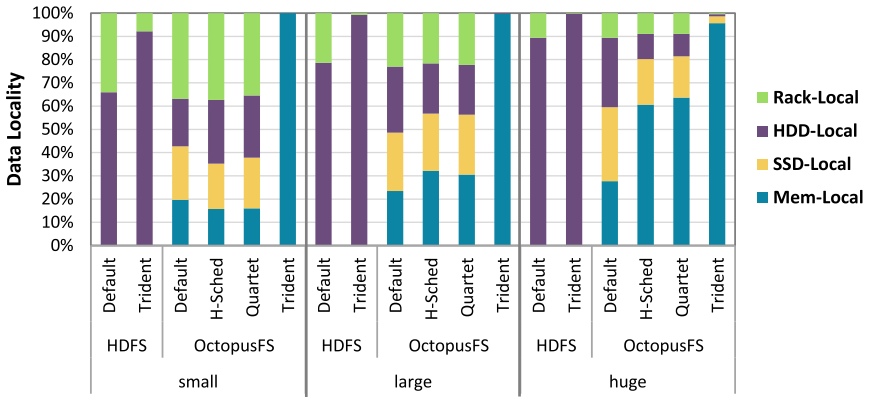

Fig. 11. Data locality rates for all schedulers over the two file systems (HDFS and OctopusFS) for the HiBench MapReduce workload for data scales *small*, *large*, and *huge*.

7.2 Evaluation of Storage-tier-aware Scheduling in Hadoop with HiBench

To further investigate the impact of task scheduling on a variety of workloads exhibiting different characteristics, we used the popular HiBench benchmark v7.1 [37], 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) [35]. Table 3 lists the applications along with the physical resource(s) that dominate their execution time. In addition, all workloads were executed using three *data scale profiles* defined by the benchmark, namely *small*, *large*, and *huge*, which resulted in about 200 MB, 1.5 GB, and 10 GB of input data per application, respectively.

Since the individual workload characteristics (i.e., whether the application is CPU-bound or I/O-bound or network-bound) do not affect data locality rates, we present the aggregate rates for each scale profile in Figure 11. The individual data locality rates per application are very similar to the aggregated ones. 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 opportunities for data-local scheduling (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

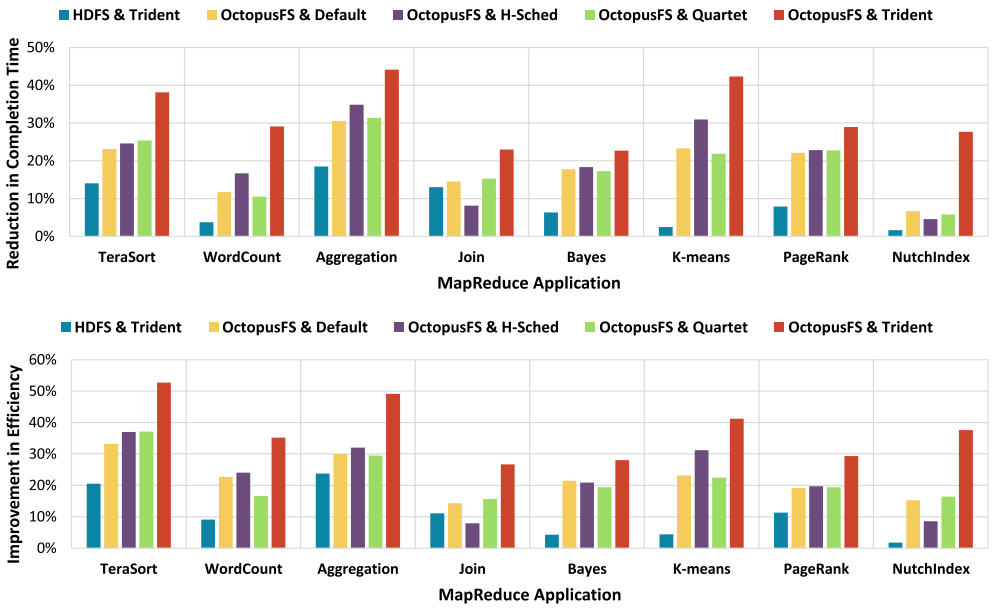

Fig. 12. (a) Percentage reduction in completion time and (b) percentage improvement in cluster efficiency compared to Default Scheduler over HDFS for the HiBench MapReduce applications running with the *large* data scale.

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. Having a lot of available resources in a cluster can improve data locality for huge jobs, because it increases the likelihood of scheduling several data-local (and in the case of H-Scheduler and Quartet, memory-local) tasks on the available resources of any cluster node. This does not necessarily apply to the smaller jobs, because the likelihood of finding a task that can be memory-local on a particular node is reduced. Finally, the Trident Scheduler over OctopusFS is *able to achieve 100% data locality with over 96% memory-locality across all three data scales* due to the optimality guarantees of the minimum cost maximum matching formulation, demonstrating once again its superior scheduling abilities irrespective of workload size or characteristics.

Figure 12(a) shows the percentage 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 by extension 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 [19]; 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 high CPU processing needs. The benefits from Trident over OctopusFS range between 23% and 29%, while they are

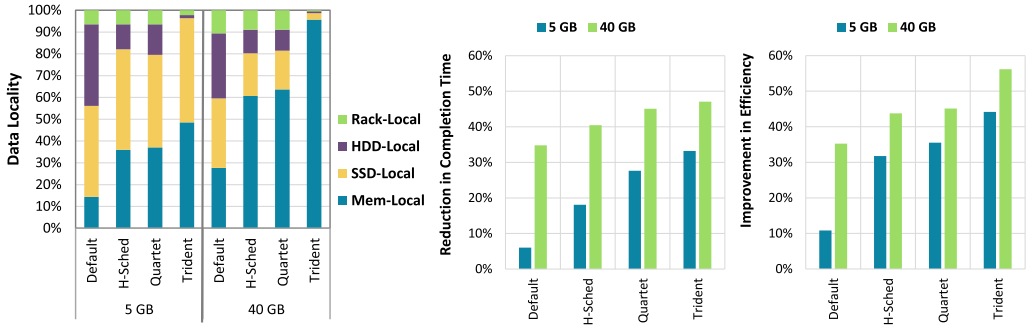

Fig. 13. (a) Data locality rates for all schedulers, (b) percentage reduction in completion time, and (c) percentage improvement in cluster efficiency compared to Default Scheduler over HDFS when allocating 5 GB vs. 40 GB for the OctopusFS's memory tier for the *huge* HiBench workload.

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 \raise.17ex~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 12(b) shows the corresponding improvement in cluster efficiency for the *large* HiBench applications. The efficiency results 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 [33].

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% (i.e., 2.3× speedup).

In the above experiments, the memory tier of OctopusFS is configured to use 40 GB (i.e., 4 GB per node), which is sufficient for storing 1 replica of the input dataset of each application of the *huge* HiBench workload. To evaluate the impact of the memory configuration on the benefits provided by Trident, we repeated the experiment with the *huge* HiBench MapReduce workload but constrained the capacity of the memory tier to 5 GB (i.e., 512 MB per node). With this setup, only about half of the input data can have a replica in the memory tier. Figure 13(a) compares the aggregate data locality rates achieved by all schedulers over OctopusFS for the two memory sizes for the *huge* HiBench workload. When the memory is constrained to 5 GB, the Trident Scheduler is able to achieve almost 50% memory locality, because it is able to optimally schedule the tasks that are planning to process a block with a replica in memory. Another 48% of tasks are SSD-local, while the remaining tasks are split between HDD-local and rack-local assignments. The other two tier-aware schedulers achieve around 36% of memory locality, which is close to half compared to the 63% of memory locality achieved when all input data reside in memory. The same trend is observed for the Default Scheduler, whose memory locality reduces from 28% to 15%. Hence, *the achieved memory locality rate is proportional to the number of input blocks that reside in memory.*

Figure 13(b) and (c) respectively show the average percentage reduction in completion time and percentage improvement in cluster efficiency compared to the baseline for the *huge* HiBench workload for the two memory settings. As expected, the benefits achieved by all schedulers is lower for

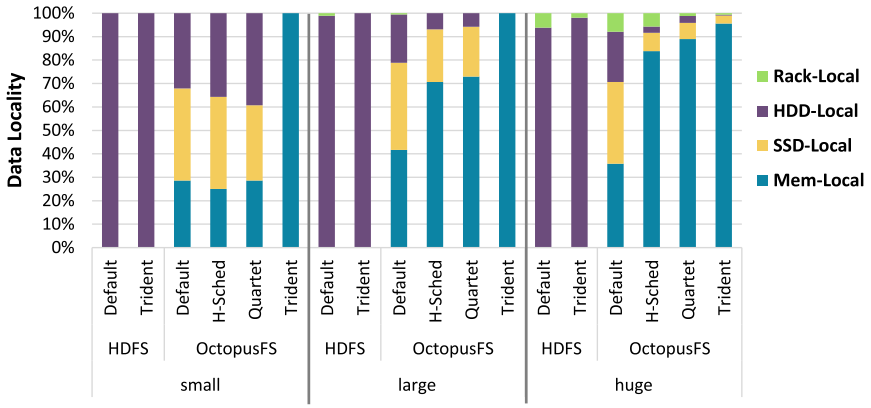

Fig. 14. Data locality rates for all schedulers over the two file systems (HDFS and OctopusFS) for the HiBench Spark workload for data scales *small*, *large*, and *huge*.

both metrics due to the reduced memory locality rates, but not by the same amount. In particular, the reduction in completion time of H-Scheduler and Quartet is reduced to half in the memory-constraint scenario, whereas for Trident it decreases from 47% to 33%. When no more tasks can be scheduled in a memory-local way, Trident shifts the assignment of tasks in an SSD-local way (more than the other schedulers), leading to good benefits for both application performance and cluster efficiency.

Key takeaways. These performance results clearly demonstrate that Trident can achieve very high memory-locality rates (>95%) irrespective of application size or characteristics. The benefits in terms of completion time and cluster efficiency are typically higher for I/O- and network-intensive applications as well as for larger jobs.

7.3 Evaluation of Storage-tier-aware Scheduling in Spark with HiBench

The evaluation with the HiBench workloads was repeated on Spark in the same manner as on Hadoop (described in Section 7.2), 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 [19]. 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 14. The 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 Executor slot at a time, which increases the opportunities for data-local assignments (or memory-local in the case of H-Scheduler and Quartet). These features are shared by the H-Scheduler and Quartet as well and are thus 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 percentage of process-local tasks is the same for all schedulers over both file systems.

Fig. 15. Percentage reduction in completion time compared to Default Scheduler over HDFS for the HiBench Spark applications running with (a) the *large* and (b) the *huge* data scale.

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 15 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.*

Key takeaways. The benefits in terms of improved locality, reduction in completion time, and reduction in cluster efficiency provided by Trident to Spark workloads and clusters are very similar to the ones provided for Hadoop.

7.4 Impact of Data Prefetching over Tiered Storage

In this section, we investigate the impact of data prefetching in addition to scheduling over tiered storage. We experiment with two flavors of the Trident Data Prefetcher, one that does not allow any delays in the execution of tasks (denoted as *Trident-D*) and one that does (denoted as *Trident+D*), as discussed in Section 4.2. For comparison purposes, we implemented a prefetcher (called *PrefetchAll*)

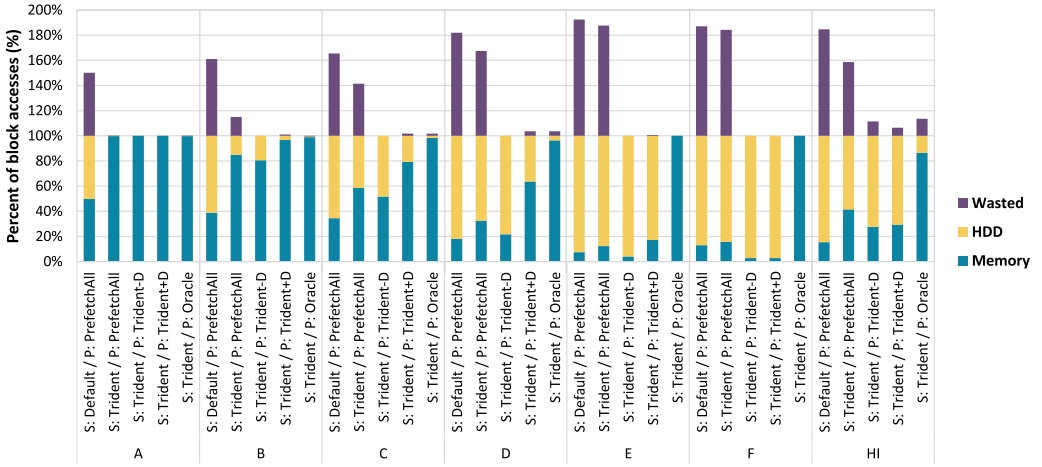

Fig. 16. Percentage of block accesses from memory and HDDs for the various scheduling (S) and prefetching (P) combinations for the Facebook workload (Bins A–F) and the HiBench MapReduce workload (HI) with the *huge* data scale. The bars appearing over 100% represent the percentage of blocks that were prefetched but not accessed from memory (i.e., wasted).

that will issue prefetching requests for all data needed by each job upon its submission. Finally, we also implemented an *Oracle* prefetcher for prefetching the data needed before each job submission, so that one input block replica is always available in memory when needed. The Oracle prefetcher cannot exist in practice but we implemented one to establish an upper bound on the potential benefits provided by any prefetcher.

For these experiments, we used both the Facebook workload and the HiBench MapReduce workload with the *huge* data scale. The input data are initially stored using three replicas on HDDs, while prefetching requests cache one replica into memory. We repeated the experiments using HDFS and OctopusFS as the underlying storage system but the results were very similar. Thus, we only show the results for OctopusFS. Finally, to demonstrate the impact (and necessity) of a storage-tier-aware scheduler when prefetching is enabled, we also run the PrefetchAll prefetcher with the Default Scheduler. In all other cases, we use the Trident Scheduler.

Figure 16 shows the percentage of input blocks accessed from memory and HDDs for each scheduler-prefetcher combination. For the Facebook workload, we break down the results to the six job sizes for Bins A–F (recall Table 2), while for the HiBench workload we present the aggregated results from the eight applications (see Table 3). Since the Trident Scheduler is able to achieve near 100% data-locality in all scenarios, and to keep the figure readable, we do not differentiate between node-local and rack-local block accesses. The figure also shows the percentage of blocks that were prefetched but not read from memory by the tasks, i.e., prefetching was *wasted*. This figure enables direct comparisons across multiple dimensions, showcasing the impact of scheduler, prefetcher, and their combination on both memory locality and wasted prefetch operations.

Our first observation from Figure 16 confirms that prefetching data without a proper task scheduler in place will result in many wasted prefetching requests. For example, in the case of small Facebook jobs (Bin A), prefetching all input data and using the Default Scheduler, which does not take storage tiers into account, leads to 50% of memory accesses, thereby wasting the prefetching of the other 50%. However, *using the Trident Scheduler leads to* 100% *of memory accesses when prefetching all data*, revealing that prefetching completes before the tasks begin execution (due to the small input size). As the job size increases, prefetching all input data leads to I/O contention when caching

Table 4. A Comparison of Trident Data Prefetcher with Enabled Delay (Trident+D) against Trident Data Prefetcher without Delay (Trident-D) for the Facebook Workload

Bin	% of Jobs with delay	Mean delay (sec)	Max delay (sec)	% Improvement in memory locality	% Reduction in mean job latency	% Reduction in 99% tail job latency	% Improvement in cluster efficiency
A	0.27	0.62	0.90	0.00	−0.20	−0.14	−0.59
B	9.26	2.98	4.48	16.11	1.64	8.83	3.71
C	20.00	3.53	6.17	27.59	2.77	9.11	4.44
D	26.67	3.83	6.10	41.99	3.71	12.98	8.54
E	12.50	3.50	3.50	13.01	2.73	7.32	0.79
F	0.00	0.00	0.00	0.00	0.22	0.36	0.76

data from the HDDs into memory. This causes delays in prefetching and many tasks are scheduled for execution before prefetching completes. As a result, the percentage of *wasted* block prefetches increases with job size when the PrefetchAll is used. For the larger jobs (Bins E–F), over 84% of prefetches are wasted with either scheduler. Therefore, prefetching all input data is not a viable strategy for improving cluster performance.

In the case of small jobs, the Trident Data Prefetcher (with and without delay) correctly prefetched all data and achieved 100% memory accesses. However, as the job size increases, the Trident Data Prefetcher becomes increasingly selective and only prefetches a small portion of the overall data; the portion that avoids disk contention and ensures that the prefetched data will be read by the tasks. As a result, the percentageof memory accesses decreases with increasing job size but only a very small fraction, if any, of the prefetch requests are wasted (less than 4%). When the delay feature is enabled, the Trident+D Prefetcher is able to achieve higher percentages of memory accesses for the mid-sized jobs (Bin B-E) compared to Trident-D, as also shown in Table 4. For instance, 27.6% of the jobs belonging to Bin D are delayed on average by 3.8 seconds with a max delay of 6.1 seconds. As a result, the percentage of memory accesses increases from 22% with Trident-D to 64% with Trident+D, leading to an overall 12% decrease in the average job completion time (including the delay). For smaller jobs, Trident+D will delay a smaller percentage of jobs for a smaller duration, because prefetching completes quickly, making delays unnecessary. For larger jobs, Trident+D will also delay a small percentage of jobs, because delaying the job to wait for prefetching to complete will typically eliminate the benefits of reading data from memory. In fact, for Bin F, Trident+D will choose not to delay any job and behaves identically to Trident-D. In addition to reducing the mean job completion time, Trident+D is also able to reduce the 99% tail job completion time by up to 13% for mid-sized jobs (Bin D) compared to Trident-D because of the large percentage of jobs (27.6%) that are able to benefit from prefetching with delay. Finally, as expected, the Oracle Prefetcher leads to very high memory access percentages of over 96% for the Facebook workload and 87% for the HiBench workload, with small corresponding waste.

Figure 17(a) and (b) show the percentage reduction in completion time and percentage improvement in cluster efficiency, respectively, for the various scheduling and prefetching combinations compared to the Default Scheduler without prefetching over OctopusFS. Prefetching all data when using the Default Scheduler yields almost no benefits with regards to application performance (despite the increased memory access rates), while it will hurt cluster efficiency due to the large amount of wasted prefetches. Prefetching all data when using the Trident Scheduler leads to some modest reduction in completion time (4–11%) due to the increased memory-local tasks, but these benefits come in the expense of reduced cluster efficiency due to the wasted prefetches. The

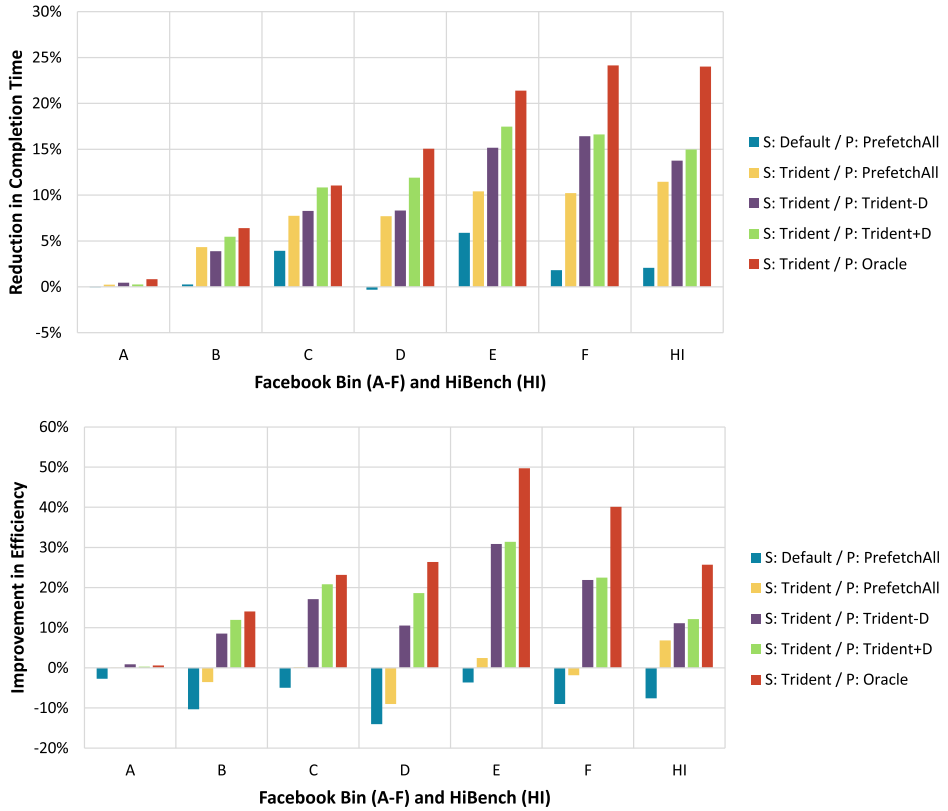

Fig. 17. (a) Percent reduction in completion time and (b) percentage improvement in cluster efficiency for the various scheduling (S) and prefetching (P) combinations compared to Default Scheduler without prefetching over HDFS for the Facebook workload (Bins A–F) and the HiBench MapReduce workload (HI).

Trident Data Prefetcher, with its good memory access rates and minimal waste, always leads to higher reductions in completion times (up to 17%), while at the same time achieving high cluster efficiency gains (up to 31%). Trident+D, with its targeted use of delaying some task executions, it is able to provide some additional benefits compared to Trident-D, up to 3.7% reduction in mean completion time and 8.5% improvement in cluster efficiency.

Finally, we compare the scenario of the Oracle Prefetcher against the Trident Data Prefetcher. Recall that in the former case, jobs begin execution with their input data already cached in memory, while in the later case, no data are initially in memory. Hence, the Oracle case represents the theoretically best benefits achievable by any prefetcher. For large jobs, it is not possible to prefetch all data in memory while the job is running and achieve good memory access rates. Hence, there is some observed difference between the Oracle and the Trident experiments, up to 8% in terms of job completion time. For smaller jobs, however, *the achieved benefits of Trident are very close to the ones of the Oracle Prefetcher* (less than 3% difference), showcasing the near-optimal decisions made by the Trident Data Prefetcher.

Key takeaways. From the results, it becomes clear that the Trident Data Prefetcher, with the algorithms and modeling presented in Section 4, is able to navigate the various tradeoffs effectively and balance the amount of data to prefetch as well as specify an appropriate delay (if any), to maximize the benefits from prefetching data over tiered storage.

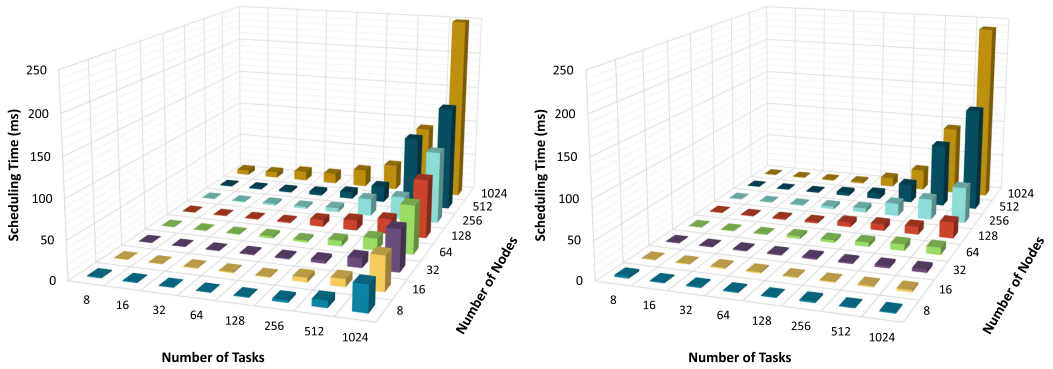

Fig. 18. Trident's scheduling time in Spark Driver when the two pruning algorithms are (a) disabled or (b) enabled.

7.5 Evaluation of Trident's Overheads

We begin the investigation of overheads by evaluating 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 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 18 shows the scheduling times as we vary both n and r between 8 and 1,024 (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 with pruning is below 2 ms regardless the cluster size, whereas it can reach 20 ms without pruning for 1,024 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 and the main algorithm's complexity is $O(\min(n, r)^3)$. However, even in the extreme case of scheduling 1,024 tasks on a 1,024-node cluster, the scheduling time is only 240 ms. 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 with complexity $O(klg(k))$, where k is the number of requested containers. Specifically, in the case of 1,024 tasks × 1,024 nodes, Trident's scheduling time is 61 ms as opposed to 60 ms for FIFO and 162 ms for Capacity (extra time due to updating queue statistics after each assignment), highlighting the negligible overheads induced by our scheduling approach.

Finally, we evaluated the time needed for the Trident Data Prefetcher to select which blocks to prefetch, per Algorithm 7. For this purpose, we varied the number of nodes in the cluster ($4 - 1{,}024$), the number of input data blocks ($4 - 1,024$), as well as the number of storage devices per node ($3 - 12$), since our prefetching modeling works at the level of individual storage devices

(recall Algorithm 4). In the majority of the tested scenarios, the prefetching algorithm run in less that 2 ms, while in the worst case scenario of 1,024 input data blocks × 1024 nodes × 12 storage devices per node, it run in 20 ms, showcasing the high efficiency of the Trident Data Prefetcher.

Key takeaways. The very low overheads observed for task scheduling (average: 13.6 ms; max: 240 ms), resource scheduling (average: 9.5 ms; max: 61 ms), and data prefetching (average: 3.2 ms; max: 20 ms) reveal the practicality of the Trident methodology, even when scheduling very large jobs to very large clusters.

8 CONCLUSION AND FUTURE WORK

The advent of tiered storage systems has introduced a new dimension in the scheduling and prefetching problems in cluster computing. Specifically, it is important for task schedulers, resource schedulers, and data prefetchers to consider both the locality and the storage tier of the accessed data when making decisions, to improve application performance and cluster utilization. In this article, we propose Trident, a comprehensive approach with three key components: (i) a task scheduler that casts the task scheduling problem into a minimum cost maximum matching problem in a bipartite graph and uses two pruning algorithms that enable Trident to efficiently find the optimal solution; (ii) a resource scheduler that makes decisions based on a newly introduced notion of preferences in cluster resource requests that can account for storage tiers; and (iii) a data prefetcher that employs a cost modeling optimization approach for deciding which data and when to prefetch, and coordinates with the schedulers for maximizing the impact of prefetching. We have implemented Trident in both Hadoop and Spark, showcasing the generality of the approach in scheduling tasks for two very different platforms. The 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.

In this work, the cluster is considered to have homogeneous nodes in terms of CPU capabilities and memory sizes. Adding support for clusters with heterogeneous nodes requires adjusting the models used by the proposed data prefetching methodology (i.e., Algorithms 4 and 5), while we do not expect any significant changes for the proposed task and resource scheduling approaches. We leave this investigation as future work. Another potential enhancement to the proposed task and resource scheduling approaches is to incorporate additional task characteristics to improve scheduling decisions, such as the task's data input size or estimated runtime. The cost function presented in Equation (1) can be enhanced or replaced to account for such characteristics in addition to the storage tier preference score, to prioritize the assignment of some tasks over others. Going a step further, a learning mechanism can be incorporated into Trident for learning the various preference scores based on real-time observations of data accesses, alleviating the need for an administrator to set the scores manually. By analyzing the patterns and characteristics of data accesses, the system can measure the costs associated with accessing data from local caches, different storage tiers, as well as other cluster nodes, and set up the scores accordingly.

Regarding the data prefetching models, there is the possibility of making inaccurate time estimations in certain scenarios or conditions due to the interaction with other components of the system such as operating system buffers or I/O schedulers. An interesting idea for future work would be to monitor the time taken for prefetching data during job execution and compare it against the time estimated by the model to identify any cases where the model makes incorrect estimates and correct them. Finally, our experimental evaluation on prefetching has revealed some interesting tradeoffs between prefetching all (or more) data and achieving higher memory locality rates versus improving application performance and cluster utilization. The wasted prefetching would be reduced or justified if the prefetched data were to be used by future applications. This

observation gives rises to interesting future research directions for predicting the future use of input data before prefetching and/or implementing our proposed methodology in higher-level workflow management systems such as Apache Oozie [9] and Azkaban [11].

The aforementioned workflow management systems typically express analytical workloads as directed acyclic graphs of jobs. In such workloads, the output data from one job becomes the input to the following job(s), and hence, smart (intermediate) data placement combined with task scheduling can have great benefits to the overall workload performance. For example, a Trident-based workload scheduler can place the intermediate data in local memory or SSDs to speed up the overall processing. In addition, it can use its own knowledge and understanding of the workload to increase or decrease the replication factor per tier in the file system to improve locality rates and performance. Overall, the co-optimization of data placement and task scheduling is an intriguing direction for future research.

REFERENCES

[1] Cristina L. Abad, Yi Lu, and Roy H. Campbell. 2011. DARE: Adaptive data replication for efficient cluster scheduling. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 159–168.

[2] Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, and Yuanfeng Tian. 2018. EC-store: Bridging the gap between storage and latency in distributed erasure coded systems. In *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE, 255–266. https://doi.org/10.1109/ICDCS.2018.00034

[3] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. 2012. Tarazu: Optimizing MapReduce on heterogeneous clusters. *ACM SIGARCH Comput. Arch. News* 40, 1 (2012), 61–74.

[4] Alluxio 2023. *Alluxio: Data Orchestration for the Cloud*. Retrieved September 18, 2023 from http://www.alluxio.org/

[5] 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 *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*. ACM, 287–300.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2011. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS'11)*. USENIX, 12–17.

[7] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX, 267–280.

[8] Apache Hadoop 2023. Apache Hadoop. Retrieved September 18, 2023 from https://hadoop.apache.org

[9] Apache Oozie 2023. Apache Oozie Workflow Scheduler for Hadoop. Retrieved September 18, 2023 from https://oozie.apache.org/

[10] Apache Spark 2023. Apache Spark. Retrieved September 18, 2023 from https://spark.apache.org

[11] Azkaban 2023. Azkaban: Open-source Workflow Manager. Retrieved September 18, 2023 from https://azkaban.github.io/

[12] Chien-Hung Chen, Ting-Yuan Hsia, Yennun Huang, and Sy-Yen Kuo. 2017. Scheduling-aware data prefetching for data processing services in cloud. In *Proceedings of the 31st International Conference on Advanced Information Networking and Applications (AINA'17)*. IEEE, 835–842.

[13] Chien-Hung Chen, Ting-Yuan Hsia, Yennun Huang, and Sy-Yen Kuo. 2019. Data prefetching and eviction mechanisms of in-memory storage systems based on scheduling for big data processing. *IEEE Trans. Parallel Distrib. Syst.* 30, 8 (2019), 1738–1752.

[14] Quan Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo. 2010. SAMR: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology (ICCIT'10)*. IEEE, 2736–2743.

[15] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB* 5, 12 (2012), 1802–1813.

[16] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The case for evaluating MapReduce performance using workload suites. In *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'11)*. IEEE, 390–399.

[17] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. 2014. Improving MapReduce performance in heterogeneous environments with adaptive task tuning. In *Proceedings of the 15th IEEE International Conference on Cluster Computing (CLUSTER'14)*. ACM, 97–108.

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

[19] Francis Deslauriers, Peter McCormick, George Amvrosiadis, Ashvin Goel, and Angela Demke Brown. 2016. Quartet: Harmonizing task scheduling and caching for cluster computing. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*. USENIX, 1–5.

[20] Ran Duan and Seth Pettie. 2014. Linear-time approximation for maximum weight matching. *J. ACM* 61, 1 (2014), 1–23.

[21] 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 *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. ACM, 321–333.

[22] Rohan Gandhi, Di Xie, and Y. Charlie Hu. 2013. PIKACHU: How to rebalance load in optimizing MapReduce on heterogeneous clusters. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*. USENIX, 61–66.

[23] Kannan Govindarajan, Supun Kamburugamuve, Pulasthi Wickramasinghe, Vibhatha Abeykoon, and Geoffrey Fox. 2017. Task scheduling in big data-review, research challenges, and prospects. In *Proceedings of the 9th International Conference on Advanced Computing (ICoAC'17)*. IEEE, 165–173.

[24] GridGain 2023. *GridGain In-Memory Data Platform for High-Performance Applications*. Retrieved September 18, 2023 from http://www.gridgain.com/

[25] Tao Gu, Chuang Zuo, Qun Liao, Yulu Yang, and Tao Li. 2013. Improving MapReduce performance by data prefetching in heterogeneous or shared environments. *Int. J. Grid Distrib. Comput.* 6, 5 (2013), 71–82.

[26] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'16)*. ACM, New York, NY, 202–215. https://doi.org/10.1145/2934872.2934908

[27] Hadoop: Fair Scheduler 2023. Hadoop: Fair Scheduler. Retrieved September 18, 2023 from https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[28] HDFS 2023. Centralized Cache Management in HDFS. Retrieved from https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html

[29] HDFS 2023. HDFS Archival Storage, SSD & Memory. Retrieved September 18, 2023 from https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html

[30] Herodotos Herodotou. 2019. AutoCache: Employing machine learning to automate caching in distributed file systems. In *Proceedings of the IEEE 35th International Conference on Data Engineering Workshops (ICDEW'19)*. IEEE, 133–139.

[31] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proc. VLDB* 4, 11 (Aug. 2011), 1111–1122.

[32] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv.* 53, 2 (2020), 1–37.

[33] Herodotos Herodotou and Elena Kakoulli. 2019. Automating distributed tiered storage management in cluster computing. *Proc. VLDB* 13, 1 (2019), 43–56.

[34] Herodotos Herodotou and Elena Kakoulli. 2021. Trident: Task scheduling over tiered storage systems in big data platforms. *Proc. VLDB Endow.* 14, 9 (2021), 1570–1582.

[35] HiBench 2023. HiBench Suite. Retrieved September 18, 2023 from https://github.com/intel-hadoop/HiBench

[36] 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 *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*. USENIX, 295–308.

[37] 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.

[38] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. ACM, 261–276.

[39] Jingjie Jiang, Shiyao Ma, Bo Li, and Baochun Li. 2016. Symbiosis: Network-aware task scheduling in data-parallel frameworks. In *Proceedings of the 35th IEEE International Conference on Computer Communications (INFOCOM'16)*. IEEE, 1–9.

[40] Elena Kakoulli and Herodotos Herodotou. 2017. OctopusFS: A distributed file system with tiered storage management. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, 65–78.

[41] K. R. Krish, Ali Anwar, and Ali R. Butt. 2014. hatS: A heterogeneity-aware tiered storage for hadoop. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*. IEEE, 502–511.

[42] Chunlin Li, Jing Zhang, Yi Chen, and Youlong Luo. 2019. Data prefetching and file synchronizing for performance optimization in hadoop-based hybrid cloud. *J. Syst. Softw.* 151, 5 (2019), 133–149.

[43] Runhui Li, Yuchong Hu, and Patrick P. C. Lee. 2017. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2500–2513. https://doi.org/10.1109/TPDS.2017.2678505

[44] Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (2015), 1537–1550.

[45] Seyed Reza Pakize. 2014. A comprehensive view of hadoop mapreduce scheduling algorithms. *Int. J. Comput. Netw. Commun. Secur.* 2, 9 (2014), 308–317.

[46] Fengfeng Pan, Jin Xiong, Yijie Shen, Tianshi Wang, and Dejun Jiang. 2018. H-scheduler: Storage-aware task scheduling for heterogeneous-storage spark clusters. In *Proceedings of the 24th IEEE International Conference on Parallel and Distributed Systems (ICPADS'18)*. IEEE, 1–9.

[47] Tarikul Islam Papon and Manos Athanassoulis. 2021. A parametric I/O model for modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DAMON'21)*. ACM, New York, NY. https://doi.org/10.1145/3465998.3466003

[48] Mario Pastorelli, Damiano Carra, Matteo Dell'Amico, and Pietro Michiardi. 2015. HFSP: Bringing size-based scheduling to hadoop. *IEEE Trans. Cloud Comput.* 5, 1 (2015), 43–56.

[49] Aparna Raj, Kamaldeep Kaur, Uddipan Dutta, V. Venkat Sandeep, and Shrisha Rao. 2012. Enhancement of hadoop clusters with virtualization using the capacity scheduler. In *Proceedings of the Third International Conference on Services in Emerging Markets*. IEEE, 50–57.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the 26th International Conference on Massive Storage Systems and Technology (MSST'10)*. IEEE, 1–10.

[51] Mbarka Soualhia, Foutse Khomh, and Sofiène Tahar. 2017. Task scheduling in big data platforms: A systematic literature review. *J. Syst. Softw.* 134, 8 (2017), 170–189.

[52] Mingming Sun, Hang Zhuang, Changlong Li, Kun Lu, and Xuehai Zhou. 2016. Scheduling algorithm based on prefetching in MapReduce clusters. *Appl. Soft Comput.* 38, 82 (2016), 1109–1118.

[53] Mingming Sun, Hang Zhuang, Xuehai Zhou, Kun Lu, and Changlong Li. 2014. HPSO: Prefetching based scheduling to improve data locality for MapReduce clusters. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 82–95.

[54] Xiaoyu Sun, C. He, and Ying Lu. 2012. ESAMR: An enhanced self-adaptive MapReduce scheduling algorithm. In *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS'12)*. IEEE, 148–155.

[55] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Oakland, CA, 513–527.

[56] SWIM 2016. *SWIM: Statistical Workload Injector for MapReduce*. Retrieved September 18, 2023 from https://github.com/SWIMProjectUCB/SWIM/wiki

[57] Jian Tan, Xiaoqiao Meng, and Li Zhang. 2013. Coupling task progress for MapReduce resource-aware scheduling. In *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM'13)*. IEEE, 1618–1626.

[58] Zhuo Tang, Min Liu, Almoalmi Ammar, Kenli Li, and Keqin Li. 2016. An optimized mapreduce workflow scheduling algorithm for heterogeneous computing. *J. Supercomput.* 72, 6 (2016), 2059–2079.

[59] 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 *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC'13)*. ACM, 1–16.

[60] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. 2014. Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters. In *Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD'14)*. IEEE, 761–768.

[61] Shuang Wang, Jianzhong Huang, Xiao Qin, Qiang Cao, and Changsheng Xie. 2017. WPS: A workload-aware placement scheme for erasure-coded in-memory stores. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS'17)*. IEEE, 1–10. https://doi.org/10.1109/NAS.2017.8026881

[62] 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 Trans. Netw.* 24, 1 (2014), 190–203.

[63] Luna Xu, A. Butt, Seung-Hwan Lim, and R. Kannan. 2018. A heterogeneity-aware task scheduler for spark. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'18)*. IEEE, 245–256.

[64] Ze Yu, Min Li, Xin Yang, Han Zhao, and Xiaolin Li. 2015. Taming non-local stragglers using efficient prefetching in MapReduce. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'15)*. IEEE, 52–61.

[65] 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 *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, 265–278.

[66] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX, 29–42.

Received 1 August 2022; revised 26 June 2023; accepted 14 September 2023