Custody: Towards Data-Aware Resource Sharing in Cloud-Based Big Data Processing

Shiyao Ma*, Jingjie Jiang*, Bo Li*, and Baochun Li†
Department of Computer Science and Engineering, Hong Kong University of Science and Technology*
Department of Electrical and Computer Engineering, University of Toronto†

***Abstract*—With the advent of big data processing frameworks, the performance of data-parallel applications is heavily affected by the time it takes to read input data, making it important to improve data locality. Existing methods in achieving data locality have primarily focused on selecting machines to place tasks of applications. Nevertheless, the set of machines that an application can choose from is determined by a cluster manager, which is oblivious to the location of data in existing resource sharing frameworks. In this paper, we design, implement and evaluate *Custody*, a new cluster management framework that helps to maximize data locality by allocating the executor processes with local access to data to those applications in need. *Custody* achieves this objective by dynamically collecting runtime information of an application's input data and by effectively allocating executors among and within applications through theoretic analyses of the data-aware resource sharing problem. With significantly better data locality, *Custody* avoids unnecessary network transfers and thus expedites job completion times. Our experimental results on a 100-node cluster demonstrate that *Custody* can improve the data locality for input tasks by 36.9% in comparison with Spark's default cluster manager. Meanwhile, it reduces the job completion times by 14.9% due to fewer network transfers.**

I. Introduction

In recent years, we have witnessed the exponential growth of applications that perform big data processing, based on a variety of data-parallel computing frameworks (*e.g.*, MapReduce [1], Pregel [2], and Spark [3]) running in cloud datacenters. Despite new topologies [4], [5] and better bandwidth allocation schemes [6], [7], [8] proposed in the literature, reading input data into these applications are still time-consuming, especially with the ever-increasing volume of data. As reported in [9], MapReduce jobs spend more than 59% of their lifetimes in map stages due to slow network transfers for input data. It is thus critical to co-locate a task with its input by distributing the task to a machine (called a *worker node*) storing or caching its input data.

Since application tasks are given a limited amount of resources in a cluster, it is impossible for them to directly select whichever worker nodes they wish for. A cluster manager is needed to share worker nodes and datasets across applications by launching multiple *executor* processes on a worker node to run tasks from different applications concurrently. To achieve data locality, an application must first be allowed to launch executors on those worker nodes that store the inputs for its tasks, and then distribute its tasks to executors using data-aware scheduling policies [10].

Nevertheless, existing cluster managers (*e.g.*, Mesos [11], YARN [12], and Spark standalone [13]) randomly allocate available resources to applications when launching executors. For static resource sharing [13], an application only has access to a subset of executors throughout its lifetime. Therefore, the best possible locality that can be achieved is restricted by the set of executors it has access to. In contrast, although a dynamic resource sharing scheme [11], [14] continuously offers available resources to different applications, these applications cannot explicitly express their data demands to acquire certain worker nodes. Such an offer-and-accept mechanism incurs additional overhead in launching each task, since it may need a long time before the cluster manager offers the *correct* executors with data locality to an application. The tasks are thus unnecessarily delayed and ultimately slowed down.

Due to the limitations of current cluster managers, we propose to improve data locality using data-aware resource sharing. To achieve this objective, we will need to know the data demands at runtime, before distributing tasks to executors. Furthermore, since a job's completion time is dictated by the slowest task (called the *straggler*), achieving data locality for a subset of tasks would not actually improve job performance. In other words, minimizing a job's completion time requires perfect locality for all its constituent tasks. However, due to limited resources, it is impossible to satisfy the locality requirements for all the jobs during peak hours. We should carefully allocate resources to maximize the number of jobs with perfect locality while maintaining fairness across concurrent applications.

In this paper, we design, implement and evaluate *Custody*, a new system framework that helps maximize data locality by allocating the executors with local access to data to those applications who need them. Given the limited computation resources in a cluster, *Custody* coordinates across multiple concurrent applications to determine the set of executors that should be allocated to each application. Such coordination is based on our theoretical analyses on the data-aware resource sharing problem. With the objective of achieving max-min fairness among applications, we translate the resource sharing problem to the *maximum concurrent flow problem* with integral constraints, which proves to be NP-hard [15]. We circumvent such difficulty by decoupling the resource sharing problem into a two-level allocation procedure. At the top level, we decide which application should first select from the current available executors based on the percentage of local jobs it has already achieved. At the next level, we decide the locality requirements from which subset of jobs inside an application should be satisfied through an effective 2-approximation algorithm, which gives the highest priority to the job with the *fewest remaining tasks*. Through such priority-based strategies, we avoid the situation where each job in the application only gets a fraction of the desired executors, and cannot be accelerated due to stragglers that lack locality.

The highlight of *Custody* lies in its capability of allocating executors to applications who really need them without modifying or delaying task submission. The key to enabling such request-driven resource sharing is to acquire the demand for executors with different data blocks at runtime. To acquire such information without incurring extra delay, we put off the allocation process till users submit analytic jobs but before the jobs are compiled into parallel tasks. Therefore, we ensure that *Custody* never keeps jobs waiting for executors and will continuously run as if the executors are already allocated to them at the very beginning.

Last but not the least, *Custody* is designed to be practical: it does not require applications to explicitly express their locality demands. Therefore, it can be easily deployed in production clusters, improving modern data-parallel frameworks and can run existing applications with no modification. We have implemented *Custody* within Spark 1.4 [16], and thoroughly evaluated its performance with various workloads. Our experiments on a Linode [17] cluster with 100 worker nodes show that *Custody* can improve the data locality for input tasks by 36.9% on average, when compared to Spark with the default cluster manager [13]. The improved data locality by *Custody* effectively shortens the input tasks and thereby reduces the average job completion time by 14.9%.

II. Custody: Background and Motivation

Data-parallel computing frameworks support a wide range of applications, such as machine learning algorithms for recommendation systems, web search and various SQL queries [18]. These frameworks usually run in a computing cluster where a distributed file system [19], [20] stores and manages the data to be processed. Each data file is divided into fixed-size blocks and stored on worker nodes across the cluster. To ensure fault tolerance, each data block typically has three replicas randomly distributed in the cluster. Recent popularity-based strategies [9] store different numbers of replicas for each of the data blocks based on its access frequency, such that applications will not all compete for the computing slots on worker nodes storing hot data.

To be more general, we consider a cluster that simultaneously runs multiple applications with different demands for data blocks. Each worker node in the cluster can launch multiple executor processes, and runs tasks on these executors with multithreading. Leveraging modern container techniques, such as *Docker containers* [21], co-located executors could share the datasets on the same worker nodes while achieving performance isolation at the same time. Therefore, multiple applications can concurrently launch executors on the same worker node to run their own tasks without interference.

A. Achieving Data Locality in a Shared Cluster

Achieving data locality consists of three sequential steps: 1) properly placing data blocks onto different worker nodes across the cluster; 2) allocating resources on worker nodes (*i.e.*, executors) to each application; and 3) distributing a task within the application to an executor process on the worker node that stores its input data.

Existing works to achieve data locality focused only on how to replicate data blocks [9], [19], [20] and how to schedule tasks to worker nodes with their input data [10], [22], [23]. On one hand, distributed file systems in production clusters [19], [20] store three replicas for each data block to ensure fault tolerance and data availability. Recent academic research (*e.g.,* [9]) further proposed a popularity-based replica policy to eliminate hot spots and improve data accessibility. On the other hand, task schedulers designed by both the industry [22] and academia [10], [23] went to great lengths to impose data locality as constraints when scheduling tasks.

Nevertheless, the indispensable step of allocating executors to applications in consideration of data locality is still missing. With a static resource sharing model, each application has access to a fixed set of executors throughout its lifetime [13]. Since such static partitioning of the cluster only accounts for the number of executors in each set, it is highly possible that the executors allocated to an application do not store its input data. As a result, many tasks in the application cannot achieve data locality regardless of the task scheduling strategies used.

With a dynamic sharing strategy [11], [12], once an application no longer needs to launch tasks onto an executor, the corresponding resources would be released. The cluster manager then offers idle resources to another application and launches a new executor if the offer is accepted. Despite their improved efficiency, these dynamic resource managers still ignore data locality and pass the buck to the task scheduler within each application. A data-aware task scheduler would reject offers from worker nodes that cannot satisfy locality requirements, and would wait until it receives an *appropriate* offer. Thus, the resource manager has to resend an offer to multiple applications before any of them accepts it. Such frequent rejections negatively affect both the efficiency of the allocation strategy and the utilization of the cluster. To make things worse, the applications may still not achieve data locality after waiting for a long time, since the resource sharing strategies are not aware of such data locality, called *data awareness* henceforth in this paper.

B. Data-Aware Resource Sharing: A Motivating Example

To better understand the benefits of data-aware resource sharing, we consider the following situation as a motivating example.

Suppose there are four worker nodes, each of which stores one data block and has one CPU core. Two applications have just been initiated and will both submit one job consisting of two input tasks to the cluster. The input data block for each task is shown in Fig. 1. Without considering the data input information, the cluster manager would randomly allocate two

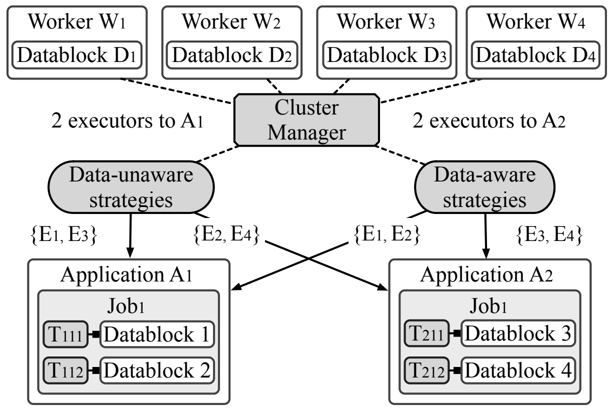


Fig. 1. In this example, each worker node can launch one executor. Existing cluster managers usually allocate executors in a round-robin fashion. For A_1 and A_2, only one task of a job can achieve data locality. However, data-aware cluster managers launch executors on nodes that store the task inputs of applications and achieve 100% data locality.

executors with equal computation power to each application. It is possible that the cluster manager launches an executor E_1 on W_1 and E_3 on W_3 for A_1. Similarly, E_2 and E_4 would be allocated to A_2. As a result, only one task of the job in each application achieves data locality no matter which task scheduling policy each application uses. But if we know the data input information beforehand, it is natural to allocate the executors E_1 and E_2 to A_1, and E_3 and E_4 to A_2. As a result, the task schedulers of both applications can easily achieve perfect data locality by embracing any data-aware scheduling strategy (*e.g.*, [10], [22]).

We next formally analyze the problem of sharing resources in a cluster with data awareness.

III. Data-aware Resource Sharing

A. System Model

Without the input information of jobs in an application, it is impossible to decide whether executors can satisfy data locality requirements. Nevertheless, traditional cluster managers dispatch executors even before the application runs [13]. We propose to *postpone* the allocation process till users submit analytic requests to an application.

Such a request is associated with an input dataset, which is usually divided into multiple equal-sized data blocks, each of which corresponds to an input task of a job that consists of a DAG (directed acyclic graph) of tasks [3]. For tasks that depend on multiple upstream tasks, it is unlikely for them to achieve data locality since they have to read the outputs from different worker nodes running the upstream tasks. In contrast, an input task reads a data block from a single worker node. Furthermore, the volume of input data is significantly larger than the volume of intermediate results [24]. Therefore, we only care about the locality for input tasks.

An application, A_i, consists of ρ_i different jobs, each of which consists of μ_{ij} input tasks. A_i has τ_i input tasks in total, each of which, denoted as T_{ijk}, needs to process one data block d_{ijk}. A worker node can launch multiple executors concurrently based on its computation resources. Each executor, denoted as E_u, has identical computation capacity, and can run one task at a time [16] for the ease of analysis. Since we only care about the data blocks stored in each executor, an executor E_u can be also defined as $E_u = \{D_x : E_u \text{ stores or caches } D_x\}$. The annotations are summarized in Table I.

TABLE I
Notations and Definitions

Notation	Definition
$E_u = \{D_x\}$	an executor process that stores multiple data blocks
A_i	an application consists of multiple jobs
y_i^u	whether an executor E_u is allocated to A_i
σ_i	the total number of executors A_i can get
ζ_i	the number of executors A_i has already got
ρ_i	the total number of jobs A_i has
τ_i	the total number of input tasks that A_i has
G_i	A_i's allocation graph that maps tasks to executors
Ψ_i	the set of edges in A_i's allocation graph G_i
$\mathcal{T}_i$	the set of tasks in A_i's allocation graph G_i
J_{ij}	A_i's job, consisting of multiple input tasks $\{T_{ijk}\}$
U_{ij}	whether J_{ij} is a local job
μ_{ij}	the number of input tasks J_{ij} has
T_{ijk}	A_i's input task, belonging to J_{ij}
L_{ijk}	whether T_{ijk} is a local task
d_{ijk}	the data block that T_{ijk} requires
x_{ijk}^u	whether E_u has the data block required by T_{ijk}
z_{ijk}^u	whether T_{ijk} should be assigned to E_u

With data-aware resource sharing, we try to allocate executors to applications in order to improve the maximum data locality that can be later achieved through task scheduling. A task satisfies data locality if its input data is stored or cached on the executor it is assigned to. A job is called local if it achieves perfect locality, namely, all its constituent input tasks achieve data locality. We next analyze the problem of achieving task-level and job-level locality in detail.

B. Achieving Task-level Data Locality

Since multiple applications share the limited resources in a cluster, we need to balance the demands from all these applications. Therefore, instead of maximizing the total number of tasks that can achieve data locality, it is more effective to maximize the minimum percentage of local tasks in each application. The problem is formulated as below:

$$\max \min_i \quad \frac{1}{\tau_i}\sum\nolimits_u \sum\nolimits_j \sum\nolimits_k x_{ijk}^u \cdot y_i^u \cdot z_{ijk}^u \tag{1}$$

$$\text{subject to} \quad \sum\nolimits_i y_i^u \leq 1, \qquad \forall\ E_u \tag{2}$$

$$\sum\nolimits_{i,j,k} z_{ijk}^u \leq 1, \qquad \forall\ E_u \tag{3}$$

$$\sum\nolimits_u z_{ijk}^u \leq 1, \qquad \forall\ T_{ijk} \tag{4}$$

$$x_{ijk}^u,\ y_i^u,\ z_{ijk}^u \in \{0,1\} \tag{5}$$

where x_{ijk}^u indicates whether the executor E_u has the data block required by T_{ijk}, which is determined by the underlying file system. y_i^u indicates whether the executor E_u is allocated to the application A_i, which is to be determined by our resource sharing strategy. z_{ijk}^u indicates whether T_{ijk} should be distributed to E_u in order to maximize locality. Constraints (2) (3) ensure that each executor can at most be allocated to one application, and can run one task at a time. Constraints (4) ensure each task can be only placed onto one executor.

From this formulation, it is clear to see that for a task T_{ijk} to achieve data locality, one of the executors storing its input data (namely, $\{E_u : x_{ijk}^u = 1\}$) has to be allocated to A_i. T_{ijk} then has to be assigned to one of the allocated executors with its input. Therefore, allocating executors and scheduling tasks are essentially co-related. On one hand, the executors allocated to an application determine the upper bound performance that can be achieved by task scheduling. On the other hand, the outcome of task scheduling helps evaluate the value of each executor for an application.

Despite the interdependency of allocating executors and scheduling tasks, we convert the problem stated in Eq. (1) - Eq. (5) to a maximum concurrent flow problem by constructing a flow network through the following steps: 1) add a source node for each application; 2) add a common virtual sink; 3) add an intermediate node for each input task and each executor; 4) construct an edge with capacity 1 between an application and each of its input task; 5) construct an edge with capacity 1 between each executor and the sink; 6) add an edge between a task and each of the executors storing its input. The demand for each application equals to its total number of input tasks, namely, the maximum locality it wants to achieve. An example flow network for two applications and three executors are shown in Fig. 2.

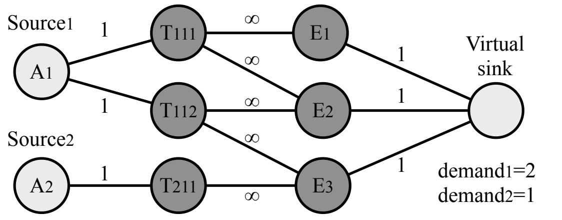


Fig. 2. The resource sharing problem to achieve max-min fairness among active applications can be transformed into a maximum concurrent flow problem, where each application acts as a source with demand equal to its total number of input tasks.

The constraints in Eq. (2) (3) (4) are then converted to capacity constraints, and the objective in Eq. (1) is translated to finding the maximum concurrent flow in the constructed network [15]. Unfortunately, this problem is NP-hard since we only allow for integral flows. Adding job-level semantics further complicates this problem as analyzed next.

C. Achieving Job-level Data Locality

In practice, each application would submit multiple jobs. A job starts when a user submits an analytic request and ends when the final computing result is returned to the user. In other words, job completion times are determined by the slowest task. The input tasks without data locality would become stragglers and lag far behind the other tasks, since network transmission is as much as 20 times slower than local data access [10]. Therefore, meeting the data locality requirement for only a subset of tasks would not optimize job performance. We must take job-level locality into consideration as well.

To examine whether a job is local, we have to inspect each of its constituent tasks. The problem to maximize the minimum percentage of local jobs is formulated below:

$$\max \min_i \quad \frac{1}{\rho_i} \sum\nolimits_j \mathrm{U}_{ij} \tag{6}$$

$$\text{subject to} \quad (2), (3), (4)$$

$$L_{ijk} = x_{ijk}^u \cdot y_i^u \cdot z_{ijk}^u, \tag{7}$$

$$U_{ij} = \prod_k L_{ijk} \tag{8}$$

where $L_{ijk} \in \{0, 1\}$ indicates whether T_{ijk} achieves data locality and U_{ij} indicates whether J_{ij} is local. We briefly prove this problem is also NP-hard by reducing the task-level resource sharing problem to it. For each task-level problem, we can construct an equivalent job-level problem where $\mu_{ij} = 1,\ \forall\, i, j$ and thus $\rho_i = \tau_i, U_{ij} = \sum_u \sum_k x_{ijk}^u \cdot y_i^u \cdot z_{ijk}^u$.

Despite the difficulty of data-aware resource sharing, we try to solve it through a two-level decision making procedure. At the first level, we decide which application should first choose from the set of idle executors to achieve data-aware max-min fairness. At the second level, we select from the set of executors to maximize the number of local jobs within the application.

IV. Custody: Design

In this section, we walk through the design of *Custody* to illustrate how it satisfies the hierarchical locality demands of active applications in a cluster. We further present how *Custody* meets the challenge of acquiring input information of tasks, and how it can be deployed in current production clusters without modifying user applications.

A. Data-Aware Allocation: Inter-Application Strategies

If the resources in a cluster are sufficient, it is trivial to allocate executors to an application that need them once we acquire the input information. Unfortunately, the resources in a cluster, namely, the free computing *slots* on executors with the desired data blocks, may become too scarce to satisfy the locality requirements from all the jobs of the active applications. On one hand, the replications of data blocks may not promptly change with the dynamic user requests. The executors storing popular blocks might be desired by multiple applications, whereas other executors may be left idle since no task needs the data it has. On the other hand, even if the placement distribution of data blocks fits the popularity distribution, the executors might have no free computing slots during busy hours. Under either situation, *Custody* needs to coordinate among different applications to handle conflicted data demands.

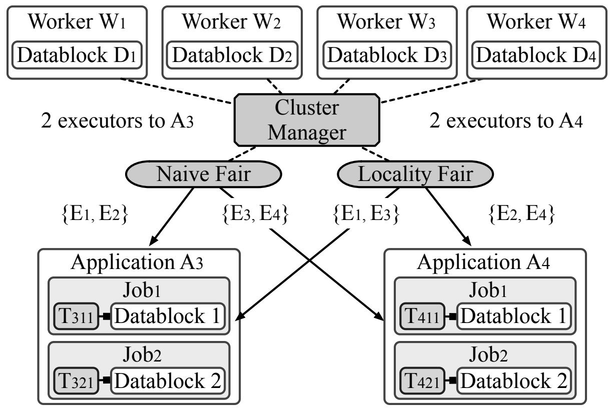


Fig. 3. Each application has two jobs, each of which has one input task. We cannot satisfy the locality for all the four jobs due to improper block placement. Using the naive fairness in existing inter-application strategies, it is possible that A_3 has two local jobs, while A_4 has no locality. Under locality-aware fairness, each application has one local job, and thus users of the two applications experience similar performance.

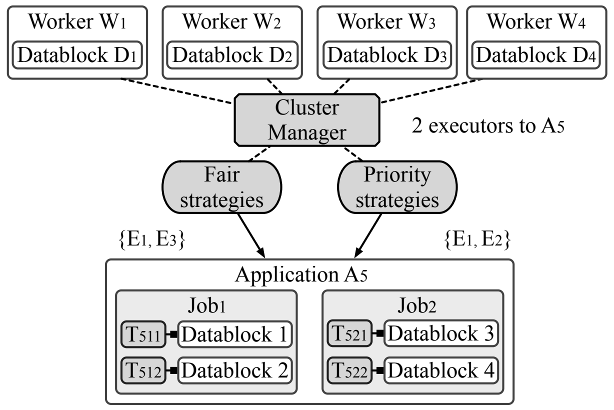


Fig. 4. Each worker node has one core and can launch one task at a time. The application A_5 only has access to two executors. Based on fairness-based strategies, one task within each job should achieve data locality by allocating E_1 and E_3 to A_5; priority strategies satisfy data locality for both tasks of the first job by allocating E_1 and E_2 to A_5.

Nevertheless, the fair sharing strategy used in existing cluster managers only consider the number of executors allocated to each application. As a result, the two allocation plans shown in Fig. 3 will be considered equivalent. If we allocate E_1 and E_2 to A_3, both jobs of A_4 need to fetch input data over the network and will be significantly slowed down. The seemingly fair allocation actually leads to different job completion times, which is directly related to user-experienced performance.

We propose to integrate data awareness into fair sharing by balancing the percentage of *local* jobs in each application to achieve max-min fairness as described in Eq. 9. It is thus natural to allocate an executor to the application with less percentage of local jobs when multiple applications compete for the same executor. The corresponding inter-application algorithm is presented in Algorithm 1.

Algorithm 1 Data-Aware Inter-Application Allocation

```
1:  procedure minLocality(apps)
2:      Sort apps in the increasing order of the percentage
3:        of local jobs
4:                  ▷ Break ties by the percentage of local tasks
5:      return the first app in the sorted list
6:  procedure Inter-App fairness
7:      Initiate: the set of running applications apps
8:      Initiate: the set of idle executors executors
9:      while executors is not empty do
10:         $A_i$ = minLocality(apps)
11:         Intra-App Allocation($A_i$, executors)
12:     ▷ Update executors and re-sort apps during allocation
```

Whenever there are idle executors in the cluster, *Custody* sorts the applications based on the percentage of local jobs they have achieved and try to allocate the set of executors to the least localized application. Under this strategy, after one of the hot executors (*e.g.*, E_1) is allocated to A_3, A_4 would have a higher priority and thus achieves the other hot executors E_2 such that both applications have one local job.

B. Data-Aware Allocation: Intra-Application Strategies

In practice, apart from data-aware fairness, we still need to limit the total number of executors each application can get. As a result, even if all the executors that an application desires are idle, it can only achieve a portion of these executors. It is thus necessary to choose the subset of executors that can maximally improve job-level locality.

Given the set idle of executors, $\mathcal{E}_{\text{idle}}$, and the set of executors desired by all the jobs in an application, $\mathcal{E}_{\text{desire}}$, we have the candidate set of executors ($\mathcal{E}_{\text{candidate}} = \mathcal{E}_{\text{idle}} \cap \mathcal{E}_{\text{desire}}$) to choose from with the objective of maximizing the number of local jobs. We simplify the intra-application resource sharing problem by asserting each local task contributes to $\frac{1}{\mu_{ij}}$ local job. The problem can then be formulated as:

$$\max \quad \sum\nolimits_j \sum\nolimits_k \frac{1}{\mu_{ij}} \mathrm{L}_{ijk} \tag{9}$$

$$\text{subject to} \quad \sum\nolimits_u y_i^u \le \sigma_i, \tag{10}$$
$$(3), (4)$$

The problem above is equivalent to finding a *constrained bipartite matching* [25] with cardinality equal to σ_i in the graph $G_i = (\mathcal{T}_i, \mathcal{E}_{\text{candidate}}, \Psi_i)$. An edge $e = (t_{ijk}, E_u)$ with weight equal to $\frac{1}{\mu_{ij}}$ exists in Ψ_i if and only if E_u stores d_{ijk}.

Despite the existence of polynomial algorithm to the constrained bipartite matching problem [25], we illustrate through the following example that a priority-based algorithm is more beneficial in practice. Consider an application A_5 consisting of two jobs, each of which would launch two tasks to process two data blocks. The data requirements and the allocate budget are shown in Fig. 4. A fairness-based intra-application strategy might allocate E_1 and E_3 to A_5 such that each job achieves data locality for one task. As a result, neither job meets locality requirements for both of its input tasks. The job completion times would be still bottlenecked by the slower task in the job. As shown in Fig. 5, the average completion time of two jobs in Fig. 4 is 2 time units due to slow network trasfers. Alternatively, a priority-based allocator will choose to satisfy perfect locality for the first job by allocating E_1 and E_2 to A_5. The average completion time then reduces to 1.25 time units.

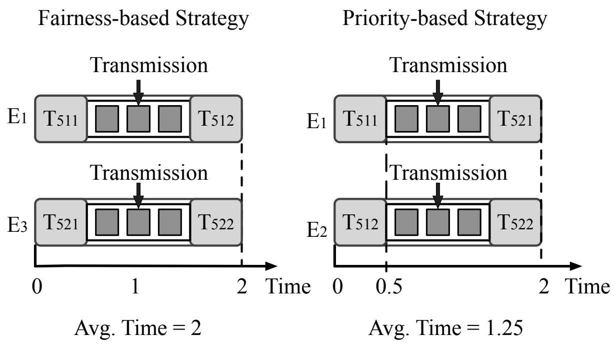


Fig. 5. For jobs within an application, if we try to achieve job-level fairness, each job in Fig. 4 can launch one local task. As a result, their completion times are both 2 time units due to network transfers. If we prioritize Job$_1$, its completion time can decrease to 0.5 time unit without slowing down Job$_2$.

Algorithm 2 Data-Aware Intra-Application Allocation

```
 1: procedure ALLOCATEEXECUTOR($A_i$, $E_u$)
 2:     Assign $E_u$ to $A_i$
 3:     $\zeta_i = \zeta_i + 1$                    ▷ Update application state
 4:     executors = executors - $E_u$              ▷ Update executors
 5:     if $A_i \neq$ MINLOCALITY(apps) then return TRUE
 6:     else return FALSE
 7: procedure INTRA-APP ALLOCATION($A_i$, executors)
 8:     Initiate jobs: the set of jobs of app
 9:     Sort jobs in the increasing order of the number of
10:       unsatisfied input tasks
11:     for $j \in$ jobs do
12:         for $t \in j$ do          ▷ Satisfy all the tasks of $j$ first
13:             if $\zeta_i == \sigma_i$ then return
14:             if $\exists$ $E_u$ in executors that stores t's input then
15:                 flag = ALLOCATEEXECUTOR($A_i$, $E_u$)
16:                 if flag then return
17:     if executors is not empty then
18:         while $\zeta_i < \sigma_i$ do
19:             for $E_u \in$ executors do
20:                 ALLOCATEEXECUTOR($A_i$, $E_u$)
```

The key is how to determine the priorities of jobs. Since every constrained bipartite matching problem has an equivalent perfect matching problem, an algorithm to the perfect matching problem also gives a solution to the original problem with the same approximation ratio. We decide the priorities of jobs based on the 2-approximation greedy algorithm to the perfect matching problem, which iteratively selects the edge with the largest weight. Equivalently, this algorithm implies that a job with fewer input tasks should be assigned with higher priority since it is easier to satisfy all its locality demands. We randomly assign priorities when two jobs have the same number of input tasks. In accordance with our strict priority-based strategy, we apply for all the desired executors of a job before moving to the next job (line 12 of Algorithm 2). For tasks in low-priority jobs that cannot achieve data locality, we offer the current idle executors to them and rely on the task scheduler in the application to reject current executors and wait for new offers. We can further utilize existing straggler mitigation schemes (*e.g.*, [26], [27], [10]) to offset such performance degradation.

C. Bridge the Gap: ***Custody*** *in Production Clusters*

In the underlying distributed file system (*i.e.,* HDFS [19]), the unique *NameNode* [28] manages the directory tree of all files in the system, and tracks where the data is stored across the whole cluster. Many *DataNodes* in the system store the actual data blocks and periodically report to the *NameNode* about their states. By inquiring the *NameNode*, *Custody* acquires the list of relevant *DataNodes* that store the input data blocks of jobs in an application.

We could further put off the allocation process until the job scheduler divides the job into parallel tasks and submits the ready tasks to the system. Nevertheless, tasks then have to wait for the cluster manager to allocate qualified executors. To avoid such extra delay, we enforce that executors with sufficient resources must be allocated to the application before task submissions, such that ready tasks can be directly launched to the executors and start processing.

Fortunately, we find that lacking detailed task information has little influence on resource sharing strategies since the input tasks of a job are mostly homogeneous. The requirements for both computation and network resources of the input tasks in a job are similar because they share the same processing logic and read the equal-sized partitions of the same dataset as input. Therefore, when a job can only get a subset of executors with its input data, selecting whichever subset is essentially equivalent in that the resultant job durations would be similar.

After users initiate job requests, *Custody* will extract the data input information of each job in the application, then locates the corresponding worker nodes by inquiring the HDFS *NameNode*. With such information, *Custody* first inquires the cluster manager about the total amount of resources each application can use and tracks the available resources on the worker nodes with input data, such as available cores and free memory space. *Custody* then coordinates the conflicting demands among concurrent applications and jobs using Algorithm 1 and 2. Finally, *Custody* submits the list of desired worker nodes and the required resources on each worker node to the cluster manager to apply for executors. Such executor application would always succeed since *Custody* has already taken the resource restrictions into account.

When application users initiate new requests, *Custody* re-evaluates the demand of all the unfinished jobs. If the current executors allocated to the application cannot satisfy the de-

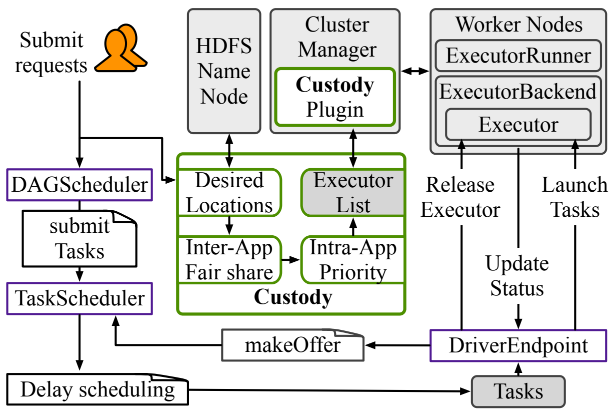


Fig. 6. Implementing *Custody* on top of Spark.

mands of the new jobs, *Custody* tries to dynamically add or remove executors to adapt to the up-to-date locality requirements.

V. System Implementation

We have implemented *Custody* to evaluate its performance on a Linode [17] cluster of 100 nodes. *Custody* is built on top of Spark 1.4.0 [16] in the Scala programming language. We run Spark under the standalone mode in our implementation to use its default cluster manager. In this section, we present the system components of *Custody* and discuss its portability.

As shown in Fig. 6, *Custody* gathers the information of locality requirements from the applications and collects the information of data locations from the *NameNode*. It then makes resource sharing decisions on behalf the cluster manager and passes the allocation results to the original cluster manager to make it allocate the desired executors to applications.

Without *Custody*, after an application registers with the cluster manager, it would be immediately allocated with a set of executors. Since such allocation is done even before users commit job requests, it is impossible to acquire the input information. Therefore, we do not allocate executors until users submit requests. *Custody* acquires the unique URL of the input dataset of a job, and then inquires the *NameNode* of HDFS to achieve the `desired locations` of the job before the `DAGScheduler` in Spark translates each job into a DAG of tasks. Because *Custody* does not require extra information from jobs, we do not need to design new application interfaces. Existing applications built upon Spark can enjoy the benefits of *Custody* without modification.

Custody then invokes the `Inter-App Fair Share` module and the `Intra-App Priority` module to deal with the possible conflicts both among and within applications. Our inter-application allocation module tries to make every application achieve the same percentage of local jobs. For intra-application allocation, *Custody* first satisfies the requests for executors from the job with the highest priority in the application.

Through these two modules, *Custody* can actually both determine the set of executors each application should get, and on which executor each task should run. *Custody* can submit both the list of executors and the scheduling suggestions to the cluster manager. Since Spark's standalone cluster manager does not support proactive requests for specific executors, we implement a *Custody* plugin inside the cluster manager to achieve this functionality. The cluster manager then allocates the executors to applications based on our proposal. Although the locality each application actually achieves depends on the task scheduling policies, we do not impose the applications to follow the instructions included in our allocation results such that each application can adopt an independent scheduling strategy without modification. In our experiments, all the applications use the standard delay scheduling [22] of Spark to accept resource offers and schedule tasks.

Custody adds a new type of messages to make the driver of Spark proactively inform the cluster manager that a specific executor can be *released*. As a result, *Custody* can keep track of all the idle executors and dynamically allocate executors once new jobs are submitted to the system. *Custody* is invoked whenever new jobs are submitted into the system or existing jobs finish and leave the system. Therefore, the application is not restricted to a fixed set of executors, but are dynamically allocated with the set of most suitable executors that can meet the locality requirements of its current active jobs.

Custody can be easily built upon other cluster management schemes (*e.g.*, YARN [12] and Mesos[11]) or acts as an independent cluster manager. The modifications would depend on specific interfaces of different cluster managers, while the key components in *Custody* remain unchanged. Our current implementation is only for evaluating the benefits of *Custody* over data-unaware cluster managers. The specific type of cluster manager has little influence on the performance gains.

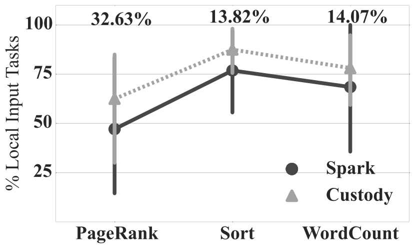
VI. Experimental Evaluation

To validate the effectiveness of *Custody*, we deploy it on a 100-node cluster and evaluate its performance under different types of workloads. We further analyze the performance of *Custody* and compare it with Spark under the default mode.

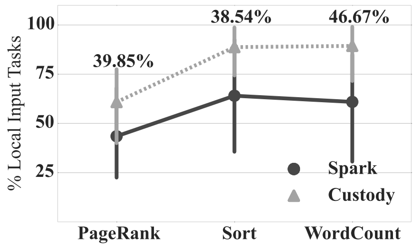
A. Setup

1) Deploy the Cluster: All our experiments are run on a 100-node cluster with each node having 8 cores, 16GB of memory and 384GB SSD storage. The bandwidth limit of a node is 40Gbps downlink and 2Gbps uplink. We write a customized module based on the popular deployment tool, Ansible [29], to launch our Linode cluster [17] and uniformly configure the nodes across the cluster.

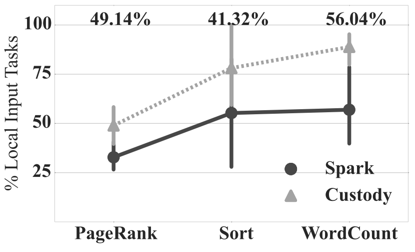
The experiments are separately run on clusters with 25, 50 and 100 nodes. Two executors are launched on each node to run tasks. According to the standard cluster configuration [10], [3], [11], the block size is set to 128MB and the replication level is set to three. For each group of experiments, we compare *Custody* with the Spark's current cluster manager, which performs very well in production datacenters [13].



(a) Cluster size = 25

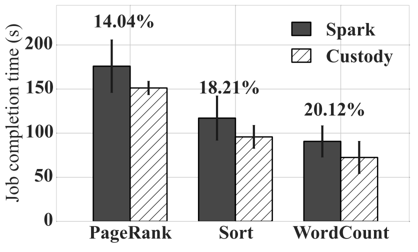


(b) Cluster size = 50

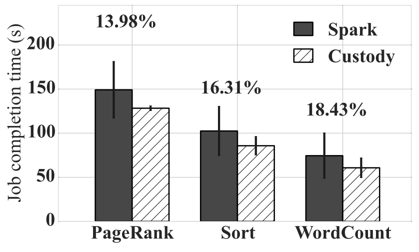


(c) Cluster size = 100

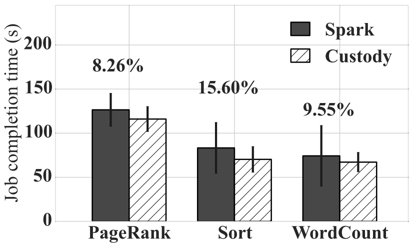
Fig. 7. The data locality of input tasks under different workloads: in all the three clusters, it is clear to see that *Custody* improves the data locality significantly with the performance gains varying between 13.82 % to 56.04%.



(a) Cluster size = 25



(b) Cluster size = 50



(c) Cluster size = 100

Fig. 8. The average job completion times under different workloads: the performance gains brought by *Custody* are over 8% in all the groups of experiments. With more worker nodes in the cluster, the performance gains of all the three workloads tend to slightly decrease.

2) Workloads: We choose three representative workloads with different characteristics.

- **PageRank**: this is a graph-based algorithm [30] that computes the rank of each web page to optimize search engine results. The input is extracted from a 32GB Wiki dump [31] that records the page-to-page Wikipedia links. Each job runs on a subset of this dump. Due to the graph-based nature, `PageRank` jobs usually involve a large amount of network transfers and are thus identified as *network-heavy* jobs. The size of the input data file for a `PageRank` job is 1GB.
- **WordCount**: this application computes the occurrence frequency of each word in the data file. Since the intermediate results of `WordCount` are significantly reduced in comparison with the input, it is usually identified as a representative of *network-light* jobs. The size of the input file for a `WordCount` job ranges between 4GB and 8GB.
- **Sort**: this type of jobs sorts the same Wiki dump based on the content of each line. Such `Sort` jobs not only call for extensive computation resources but also incur a large amount of network transmissions. The size of the input file for a `Sort` job ranges between 1GB and 8GB.

For each workload, we compare the average job completion times and the data locality for input tasks under *Custody* and Spark's default resource manager. We generate a common job submission schedule that is shared by all the experiments to minimize the influence of random factors. The distribution of inter-arrival times is roughly exponential with a mean of 14 seconds in accordance with the Facebook trace [22]. In all the following experiments, we register four applications to the cluster manager and submit 30 jobs with an independent submission schedule to each application.

B. Overall Performance

We summarize the performance gains in terms of data locality and job completion time in Fig. 7 and 8 respectively.

The percentage of input tasks in a job that achieve data locality is shown in Fig. 7. We plot the mean and standard deviation of that percentage in each workload. In comparison with Spark's default cluster manager, *Custody* improves data locality significantly: the best case performance gain is as high as 56.04% whereas the worst case gain is still larger than 10%. Furthermore, we can see from Fig. 7(a) - Fig. 7(c) that the locality achieved by the default cluster manager is very unstable. For instance, although it meets locality requirements for about 65% of input tasks on average, some jobs only have less than 35% of local tasks. In contrast, *Custody* achieves over 50% data locality for all the jobs in the cluster. Since we have not achieved perfect locality for jobs on average, the completion times of input tasks are not minimized. However, improving data locality for a portion of tasks still helps expedite the jobs since the tail completion times of all the input tasks are essentially shorten.

In Fig 8, we evaluate the average job completion times under different settings. Even in the cluster with 100 nodes, *Custody* can reduce job completion times by more than 8% for all the workloads. It is worth noticing that the performance gain in terms of job completion times is not as significant as that of data locality. There are two reasons for this interesting phenomenon. Firstly, improving task locality can only reduce the completion times of tasks in the input stages as shown

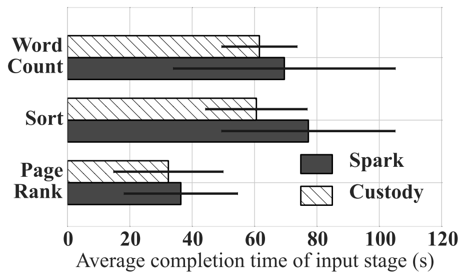


Fig. 9. The average completion time of map (input) stages in the 100-node cluster: compared to Spark's standalone cluster manager, the input tasks running with the help of *Custody* are expedited due to improved data locality.

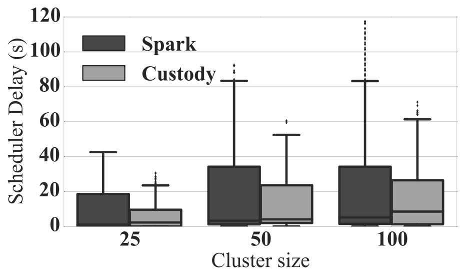


Fig. 10. Compared to Spark, tasks under *Custody* experience shorter delay since they are easier to be offered with executors that store their data.

in Fig. 9, whereas tasks in the downstream stages are not directly influenced. Secondly, the nodes we use for experiments guarantee about 2 Gbps bisection bandwidth for each node, which means transmitting a data block does not need too much time. Therefore, the benefit of data locality is actually under-estimated since most production clusters do not have such a high-speed network [22].

In addition, it is clear to see that the three workloads have various behavior under different experiment settings: `PageRank` jobs have lower performance gains in terms of job completion times. Such difference is mainly caused by the multiple iterations involved in the `PageRank` algorithm. In contrast, `Sort` jobs and `WordCount` jobs only involve one map stage and one very short reduce stage [32]. Expediting input tasks thus has less influence on `PageRank` jobs.

C. Influence of Cluster Size

By comparing the results in different cluster sizes, we find that *Custody* is more beneficial in a larger cluster. For each workload, the improvement of locality over the baseline increases with the sizes of clusters. For instance, the locality improvement of `Sort` jobs increases from 14.07% in Fig. 7(a) to 56.04% in Fig. 7(c). Essentially, the locality level under *Custody* is relatively insensitive to the sizes of clusters. No matter how many executors are available in the cluster, *Custody* can always effectively select the best set of executors that meet the data demands for tasks in an application. However, the locality level under Spark's default cluster manager decreases dramatically in larger clusters. This is due to the fact that the standalone manager randomly selects among all the available resources and allocates whichever set of executors that have sufficient computation resources to an application. With more executors spreading out in a larger cluster, it is less likely that the default manager happens to select the set of executors that store the right data blocks.

The trend of job completion times, however, is inconsistent with the data locality: the performance gain decreases from 17% in the 25-node cluster to 11% in the 100-node cluster on average. Certainly, the applications in the larger cluster can launch tasks on more executors. Even if input tasks cannot achieve data locality, they can be easily launched onto idle executors and receive their inputs without too much delay. Therefore, the benefit of data locality is largely offset.

D. Allocation Overhead of ***Custody***

The scheduler delay of a task is the time period between the task is submitted to the system and the task is actually launched onto an idle executor. We evaluate the overhead of using *Custody* to allocate executors by comparing the scheduler delay with that of Spark standalone. From Fig. 10 we can see that the scheduler delay using *Custody* is less than the standalone cluster manager on average. This extra benefit stems from the better locality *Custody* has achieved. Under the delay scheduling used in Spark, a task waits for executors that store its input for a certain amount of time before it accepts a resource offer from an executor without locality. By allocating the right executors to an application, the input tasks are easier to find the desired executors and thus spend less time waiting in the scheduler.

VII. Related Work

Cluster managers: Spark's current cluster manager [13] allocates a fixed set of executors to an application throughout its lifetime. The resource manager in YARN [12] dynamically [33] partitions the cluster resources among various applications into different resource pools, which only captures computation resources as metrics and still lacks data awareness. Mesos [11] is an offer-based platform for multiple applications to share a cluster. It offers idle computation resources to an application once an executor terminates and releases its resources. The fine-grained resource allocation in Mesos significantly improves cluster utilization. However, without data awareness, it suffers from repetitive offer rejections if data locality is required by the pending tasks.

All the schemes mentioned above as well as the cluster managers used in production clusters (*e.g.*, Omega [34] and Borg [14]) ignore data locality when allocating resources. *Custody* improves existing schemes by considering the input information of jobs and allocate the desired executors to the applications. Through carefully handling the conflicts both inter- and intra- applications, *Custody* is able to meet the locality demands of applications when they share the limited resources in clusters.

Task schedulers: Quincy [35] makes a trade-off between fairness and locality by frequently killing running tasks based on the solution to a minimum-cost flow problem. Instead, the delay scheduling used in HFS [22] waits for running tasks to release executors with desired data blocks at the cost of delayed task placement. Sparrow [23] imposes data locality as hard constraints when scheduling tasks, while lacks discussions about how to access the executors storing the relevant data. KMN [10] improves the possibility of data locality by processing a subset of input data blocks. The best locality these schedulers can achieve depends on the executors that the cluster manager has allocated to an application. *Custody* essentially complements task schedulers by maximizing the upper bound locality that task schedulers can achieve.

Caching and storage management: Distributed file systems in production clusters typically maintain three replicas for each data block [19], [20]. However, different data blocks have different access patterns. The worker nodes that store the popular blocks thus become hot spots. Anantharayanan *et al.* propose to replicate blocks based on their popularity to improve data locality [9]. Adopting these sophisticated techniques will further enhance the performance of *Custody* since they reinforce the foundation of data locality.

VIII. Conclusion

In this paper, we have proposed, designed and implemented a new resource sharing framework, *Custody*, to maximize data locality for parallel-computing applications in cloud datacenters. *Custody*'s core principle is to enable data-aware executor allocation through postponing the allocation process till users submit job requests. Utilizing the input information, *Custody* selects executors that can best meet an application's data locality requirements without delaying task submissions. We have implemented *Custody* in Spark and demonstrate it outperforms Spark's current cluster manager both through theoretic analyses and extensive experiments in a large-scale cluster under various workloads. Although *Custody* is currently built on top of Spark's cluster manager, it can be easily integrated into other resource management systems with little modification.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX OSDI*, 2004.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD*, 2010.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX NSDI*, 2012.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.

[5] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[6] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM SIGCOMM*, 2014, pp. 503–514.

[7] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proc. ACM SIGCOMM*, 2014, pp. 443–454.

[8] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM*, 2015, pp. 393–406.

[9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proc. Conference on Computer Systems (Eurosys)*, 2011, pp. 287–300.

[10] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. USENIX OSDI*, 2014, pp. 301–316.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, 2011.

[12] "Apache Hadoop NextGen MapReduce (YARN)," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[13] "Spark standalone mode," http://spark.apache.org/docs/latest/spark-standalone.html.

[14] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*, 2015, pp. 18:1–18:17.

[15] F. Shahrokhi and D. W. Matula, "The maximum concurrent flow problem," *Journal of the ACM*, vol. 37, no. 2, pp. 318–334, 1990.

[16] "Apache Spark," https://github.com/apache/spark.

[17] "Linode: SSD cloud hosting," http://www.linode.com/.

[18] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 13–24.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 29–43.

[21] "Docker: An open platform for distributed applications for developers and sysadmins," https://www.docker.com/.

[22] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. European conference on Computer systems (Eurosys)*, 2010, pp. 265–278.

[23] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 69–84.

[24] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "WANalytics: Geo-distributed analytics for a data intensive world," in *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 1087–1092.

[25] J. Plesnk, "Constrained weighted matchings and edge coverings in graphs," *Discrete Applied Mathematics*, vol. 92, no. 23, pp. 229 – 241, 1999.

[26] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 289–302.

[27] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 185–198.

[28] "Hadoop NameNode," http://wiki.apache.org/hadoop/NameNode.

[29] "Ansible is simple IT automation," http://www.ansible.com/.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: bringing order to the Web.* Stanford InfoLab, 1999.

[31] "English Wikipedia dump," http://dumps.wikimedia.org/enwiki/.

[32] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with Orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[33] "Cloudera: Dynamic resource pool," http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cm_mc_resource_pools.html.

[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. ACM European Conference on Computer Systems (EuroSys)*, 2013.

[35] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 261–276.