Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems

Andrey Goder Alexey Spiridonov Yin Wang

{agoder, lesha, yinwang}@fb.com
Facebook, Inc

Abstract

Data-intensive batch jobs increasingly compete for resources with customer-facing online workloads in modern data centers. Today, the two classes of workloads run on separate infrastructures using different resource managers that pursue different objectives. Batch processing systems strive for coarse-grained throughput whereas online systems must keep the latency of fine-grained end-user requests low. Better resource management would allow both batch and online workloads to share infrastructure, reducing hardware and eliminating the inefficient and error-prone chore of creating and maintaining copies of data. This paper describes Facebook's Bistro, a scheduler that runs data-intensive batch jobs on live, customer-facing production systems without degrading the end-user experience. Bistro employs a novel hierarchical model of data and computational resources. The model enables Bistro to schedule workloads efficiently and adapt rapidly to changing configurations. At Facebook, Bistro is replacing Hadoop and custom-built batch schedulers, allowing batch jobs to share infrastructure with online workloads without harming the performance of either.

1 Introduction

Facebook stores a considerable amount of data in many different formats, and frequently runs batch jobs that process, transform, or transfer data. Possible examples include re-encoding billions of videos, updating trillions of rows in databases to accommodate application changes, and migrating petabytes of data among various BLOB storage systems [11, 12, 31].

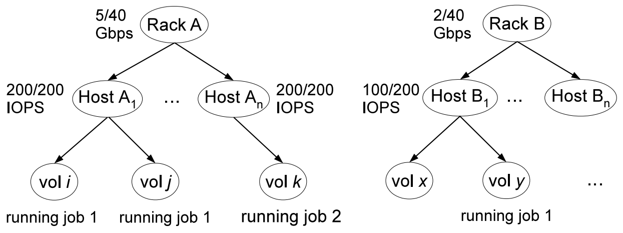
Typical large-scale data processing systems such as Hadoop [41] run against offline copies of data. *Creating* copies of data is awkward, slow, error-prone, and sometimes impossible due to the size of the data; *maintaining* offline copies is even more inefficient. These *troubles* overshadow the benefits if only a small portion of the offline data is ever used, and it is used only once. Furthermore, some batch jobs cannot be run on copies of data; e.g., bulk database updates must mutate the online data. Running batch jobs directly on live customer-facing production systems has the potential to dramatically improve efficiency in modern environments such as Facebook.

Unfortunately, existing batch job schedulers are mostly designed for offline operation [13, 15, 25, 28, 29, 32, 33, 36, 40, 42] and are ill-suited to online systems. First, they do not support hard constraints on the burdens that jobs place upon *data hosts*. The former may overload the latter, which is unacceptable for *online* data hosts serving end users. Second, batch schedulers assume immutable data, whereas online data changes frequently. Finally, a batch scheduler typically assumes a specific offline data ecosystem, whereas online systems access a wide range of data sources and storage systems.

Facebook formerly ran many data-intensive jobs on Hadoop, which illustrates the shortcomings of conventional batch schedulers. Hadoop tasks can draw data from data hosts outside the Hadoop cluster, which can easily overload data hosts serving online workloads. Our engineers formerly resorted to cumbersome distributed locking schemes that manually encoded resource constraints into Hadoop tasks themselves. Dynamically changing data poses still further challenges to Hadoop, because there is no easy way to update a queued Hadoop job; even pausing and resuming a job is difficult. Finally, Hadoop is tightly integrated with HDFS, which hosts only a small portion of our data; moving/copying the data is very inefficient. Over the years, our engineers responded to these limitations by developing many ad hoc schedulers tailored to specific jobs; developing and maintaining per-job schedulers is very painful. On the positive side, the experience of developing many specialized schedulers has given us important insights into the fundamental requirements of typical batch jobs. The most important observation is that many batch jobs are "map-only" or "map-heavy," in the sense that they are (mostly) embarrassingly parallel. This in turn has allowed us to develop a sched-

resources		jobs	
name	capacity	video re-encoding	volume compact
volume	1 lock	1 lock	1 lock
host	200 IOPS	100 IOPS	200 IOPS
rack	40 Gbps	1 Gbps	0 Gbps

(a) Resource capacity and job consumption



(b) Forest resource model

Figure 1: The scheduling problem: *maximum resource utilization subject to hierarchical resource constraints.*

uler that trades some of the generality of existing batch schedulers for benefits heretofore unavailable.

This paper describes Bistro, a scheduler that allows offline batch jobs to share clusters with online customer-facing workloads without harming the performance of either. Bistro treats data hosts and their resources as first-class objects subject to hard constraints and models resources as a hierarchical forest of resource trees. Administrators specify total resource capacity and per job consumption at each level. Bistro schedules as many tasks onto leaves as possible while satisfying their resource requirements along the paths to roots.

The forest model conveniently captures hierarchical resource constraints, and allows the scheduler to flexibly accommodate varying data granularity, resource fluctuations, and job changes. Partitioning the forest model corresponds to partitioning the underlying resource pools, which makes it easy to scale both jobs and clusters relative to one another, and allows concurrent scheduling for better throughput. Bistro reduces infrastructure hardware by eliminating the need for separate online and batch clusters while improving efficiency by eliminating the need to copy data between the two. Since its inception at Facebook two years ago, Bistro has replaced Hadoop and custom-built schedulers in many production systems, and has processed trillions of rows and petabytes of data.

Our main contributions are the following: We define a class of data-parallel jobs with hierarchical resource constraints at online data resources. We describe Bistro, a novel tree-based scheduler that safely runs such batch jobs "in the background" on live customer-facing production systems without harming the "foreground" workloads. We compare Bistro with a brute-force scheduling solution, and describe several production applications of Bistro at Facebook. Finally, Bistro is available as open-source software [2].

Section 2 discusses the scheduling problem, resource model, and the scheduling algorithm of Bistro. Section 3 explains the implementation details of Bistro. Section 4 includes performance experiments and production applications, followed by related work and conclusion in Sections 5 and 6, respectively.

2 Scheduling

Bistro schedules data-parallel jobs against online clusters. This section describes the scheduling problem, Bistro's solution, and extensions.

2.1 The Scheduling Problem

First we define a few terms. A **job** is the overall work to be completed on a set of data shards, e.g., re-encoding all videos in Haystack [12], one of Facebook's proprietary BLOB storage systems. A **task** is the work to be completed on one data shard, e.g., re-encoding all videos on one Haystack volume. So, a job is a set of tasks, one per data shard. A **worker** is the process that performs tasks. A **scheduler** is the process that dispatches tasks to workers. A **worker host**, **data host**, or **scheduler host** is the computer that runs worker processes, stores data shards, or runs scheduler processes, respectively.

Consider the scheduling problem depicted in Figure 1a. We want to perform two jobs on data stored in Haystack: video re-encoding and volume compaction. The former employs advanced compression algorithms for better video quality and space efficiency. The latter runs periodically to recycle the space of deleted videos for Haystack's append-only storage. Tasks of both jobs operate at the granularity of data volumes. To avoid disrupting production traffic, we constrain the resource capacity and job consumption in Figure 1a, which effectively allows at most two video re-encoding tasks or one volume compaction task per host, and twenty video re-encoding tasks per rack.

Volumes, hosts, and racks naturally form a forest by their physical relationships, illustrated in Figure 1b. Job tasks correspond to the leaf nodes of trees in this forest; each job must complete exactly one task per leaf. This implies a one-to-one relationship between tasks and data units (shards, volumes, databases, etc.), which is the common case for data-parallel processing at Facebook. A task

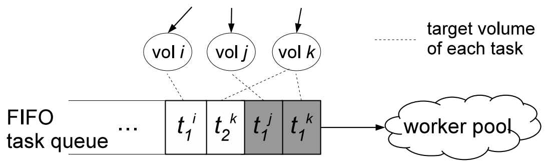


Figure 2: Head-of-line blocking using FIFO queue for our scheduling problem. *Here t_2^k can block the rest of the queue while waiting for the lock on vol k held by t_1^k, and possibly other resources at higher-level nodes.*

requires resources at each node along the path from leaf to root. In this figure, the two jobs have a total of four running tasks. Volumes i, j, k, and hosts A_1, A_n are running at full capacity, while other nodes have extra capacity for more tasks. More formally, our problem setting is the following.

- A resource forest with multiple levels, and a resource capacity defined at each level.
- A set of jobs $\mathcal{J} = \{J_1, ..., J_m\}$, where J_i consists of a set of tasks $\{t_i^a, t_i^b, ...\}$, corresponding to leaf nodes $a, b, ...$, respectively. This encodes the one-to-one relationship between tasks and data units.
- A task requires resources on nodes along its path to the root, and the demand at each level is defined per job.

Subject to resource constraints, *the scheduling objective is to maximize resource utilization.* The scheduler should never leave a task waiting if its required resources are available. High resource utilization often leads to high throughput, which in turn reduces the total *makespan*, i.e., the time required to finish all jobs. We do not directly minimize makespan, because the execution time of each task is unpredictable, and long tails are common [17]. We instead use scheduling policies to prioritize jobs, and mitigate long tails as described in Section 3.3. At large scale, the main challenge is to minimize scheduling overhead, i.e., to quickly find tasks to execute whenever extra resources become available.

In our scheduling problem formulation, each task requires resources along a unique path in our forest model. This is fundamentally different from the typical scheduling problem that considers only interchangeable computational resources on worker hosts. The latter is extensively studied in the literature, and First-In-First-Out (FIFO) queue-based solutions are common [13, 15, 25, 28, 29, 32, 33, 36, 40, 42]. For our problem, FIFO queues can be extremely inefficient. Figure 2 shows an example using the example of Figure 1. Assuming the worker pool has sufficient computational resources to run tasks in parallel, a task can easily block the rest of the FIFO queue if its required resources are held by running tasks. A non-FIFO

Algorithm 1 Brute-force scheduling algorithm (baseline)

```
 1: procedure SCHEDULEONE($M$)
 2:     for job $J_i \in \mathcal{J}$ do
 3:         for node $l \in$ all leaf nodes of $M$ do
 4:             if task $t_i^l$ has not finished and there are enough
                    resources along $l$ to root then
 5:                 $update\_resource(M, t_i^l)$
 6:                 $run(t_i^l)$
 7:             end if
 8:         end for
 9:     end for
10: end procedure

11: procedure BRUTEFORCE( )
12:     while True do
13:         SCHEDULEONE(snapshot of the resource forest)
14:     end while
15: end procedure
```

Algorithm 2 Tree-based scheduling algorithm (Bistro)

```
1: procedure TREESCHEDULE( )
2:     while True do
3:         $t \leftarrow finished\_task\_queue.blockingRead()$
4:         $d \leftarrow$ the highest ancestor of the leaf node corre-
               sponding to $t$ where $t$ consumes resources
5:         SCHEDULEONE(snapshot of the subtree at $d$
               and the path from $d$ to root)
6:     end while
7: end procedure
```

scheduler might look ahead in the queue to find runnable tasks [42], but unless the scheduler examines the entire queue, it might overlook runnable tasks. Unfortunately, for large-scale computations, the overhead of inspecting the entire queue for every scheduling decision is unacceptable. Section 2.2 introduces a more efficient scheduling algorithm and contrasts it to this brute-force approach.

2.2 Our Scheduling Algorithms

Our performance baseline is the aforementioned *brute-force* scheduler that avoids unnecessary head-of-line blocking by searching the entire queue for runnable tasks. The pseudocode is in Algorithm 1. The BRUTEFORCE function executes in an infinite loop. Each iteration examines all tasks of all jobs to find runnable tasks to execute, and updates resources on the forest model accordingly. The complexity of each scheduling iteration is $O(\text{number of jobs} \times \text{number of leaves} \times \text{number of levels})$.

In practice, each iteration of the brute-force scheduler is slow—around ten seconds for a modest million-node problem. This incurs a significant scheduling overhead for short tasks. On the other hand, one iteration can schedule multiple tasks because it runs asynchronously using snapshots of the resource model. So, as tasks finishes more quickly, each iteration reacts to larger batches of changes, amortizing the scheduling complexity.

Algorithm 2 is Bistro's more efficient *tree-based* scheduling algorithm. The algorithm exploits the fact that a task requires only resources in one tree of the resource forest, or just a subtree if it does not consume resources all the way to root. We can therefore consider only the associated tree for scheduling when a task finishes and releases resources. This algorithm again executes the scheduling procedure in an infinite loop, but instead of blindly examining all tasks of all jobs in each iteration, it waits for a finished task, and invokes the scheduling iteration on the corresponding subtree only. Although TREESCHEDULE may not schedule tasks in large batches as BRUTEFORCE does, it enables multi-threaded scheduling since different tasks often correspond to non-overlapping trees or subtrees. Bistro uses reader-writer locks on tree nodes for concurrency control.

2.3 Model Extensions and Limitations

The forest resource model is easy to extend. First, the scheduling problem described in Section 2.1 assigns one resource to each level, but it is straightforward to have multiple resources [24]. This allows heterogeneous jobs to be throttled independently, e.g., I/O bound jobs and computation bound jobs can co-exist on a host. In addition, different jobs can execute at different levels of the trees, not necessarily the bottom level. For example, we may run maintenance jobs on server nodes alongside video encoding jobs on volume nodes.

For periodic jobs like volume compaction in Figure 1, Bistro has a built-in module to create and refresh time-based nodes. We add these nodes to volumes nodes as children, so volume compaction runs at this new level periodically as the nodes refresh.

Partitioning the forest model for distributed scheduling is flexible too. Node names are unique, so we can partition trees by a hash of the names of their root nodes. Partitioning by location proximity is another choice. Some of our applications prefer filtering by leaf nodes, such that after failover, the new host that contains the same set of volumes or databases is still assigned to the same scheduler.

In addition to the forest model, Bistro supports Directed Acyclic Graphs (DAGs). A common case is data replicas, where multiple servers store the same logical volume or database. With DAGs, resources are still organized by levels, and tasks run at the bottom level. For each bottom level node, Bistro examines all paths to root and can schedule a task if any path has all the resources it needs.

Finally, Bistro is designed for embarrassingly parallel jobs, or "map-only" jobs, where each task operates on its own data shard independently. At Facebook, other than data analytic jobs on Hive [37], many batch jobs are map-only or *map-heavy*. Bistro applies to map-heavy jobs by running the reduce phase elsewhere, e.g., as a Thrift service [1]. For jobs with non-trivial "reduce" phases, our engineers came up with a solution that runs "map" and "reduce" phases on separate Bistro setups, buffering the intermediate results in RocksDB [5] or other high-performance data store; see Section 4.2.1 for an example.

2.4 Scheduling Against Live Workloads

Bistro requires manual configuration of resource capacity based on the characteristics of live foreground workloads, and manual configuration of the resource consumption of background jobs. Users can adjust these settings at runtime upon workload changes, which Bistro will enforce in the subsequent scheduling iterations by scheduling more or killing running tasks. Bistro could monitor realtime resource usage at runtime and adjust these settings automatically. However, live workloads are often spiky, such that aggressive dynamic scheduling requires reliable and rapid preemption of background jobs. This is challenging if the data, worker, and scheduler hosts are all distributed, and it complicates the task design by the requirement of handling frequent hard kills. At Facebook, job owners usually prefer static resource allocation for simplicity.

3 Implementation

This section discusses the implementation details of Bistro, including its architecture and various components.

3.1 Architecture

Figure 3 depicts the architecture of Bistro, which consists of six modules. All of these modules work asynchronously, communicating via either snapshots or queues. The *Config Loader* reads and periodically updates the resource and job configuration from some source, such as a file, a URL, or a Zookeeper instance [27]. Based on the current resource configuration, the *Node Fetcher* builds and periodically updates the resource model. For example, if we want to process all files in a directory,

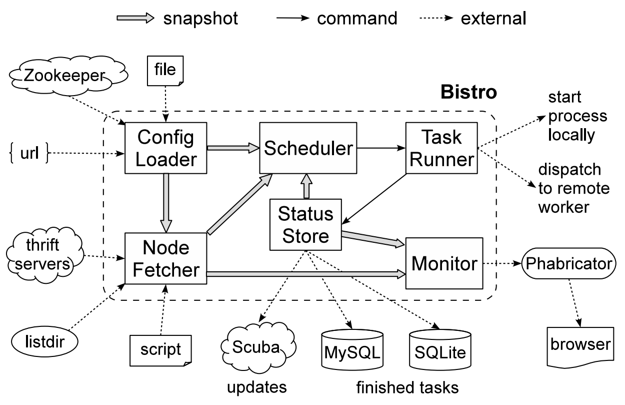


Figure 3: Bistro architecture

the Node Fetcher can periodically scan the directory to maintain an up-to-date view of its contents. Most of our large-scale jobs receive their nodes from a set of Thrift [1] servers for scalability and reliability. These servers obtain the resource model from databases or Zookeeper, typically with an intermediate Memcached layer. The *Scheduler* chooses tasks to execute based on the latest configuration and resource model, as well as the statuses of running and unfinished tasks. These statuses are maintained by the *Status Store*. The *Task Runner* receives a list of tasks to start, and can either run them locally or dispatch them to remote workers depending on the configuration, discussed in the next subsection. The Runner also monitors task execution and sends status updates to the Status Store. Users observe job progress through a Web UI provided by the *Monitor*.

3.2 Scheduling Modes

The forest resource model of Bistro facilitates flexible scheduler/worker configuration. We can have either one Bistro instance for centralized scheduling or multiple instances for distributed scheduling. Workers can reside on either data hosts or separate worker pool. This leads to four scheduling modes to accommodate diverse job requirements in large data centers, described below.

The *single/co-locate* mode has one central Bistro instance, and one worker on each data host, which receives only tasks that access local data. In addition to data locallity, the centralized scheduler can enforce optimal load balancing and fair scheduling. Since the entire forest model is on one Bistro instance, we can add a common root to all trees to enforce global resource contraints too. The scheduler is a single point of failure but Bistro can log task statuses to redundant storage for fail over. We employ this mode whenever the data hosts have sufficient compute resources and the total number of nodes is small.

The *multi/co-locate* mode is for large-scale jobs that a single scheduler cannot handle efficiently due to an excessively large resource model or an excessively high turnover rate. The latter happens if tasks are short or high concurrency. If each data host corresponds to an independent resource tree, we often run one Bistro instance on each data host too, which avoids network communication by connecting to the co-located worker directly. This scheduling mode is very robust and scalable since each Bistro instance works independently. One downside of share-nothing schedulers is poor load balancing. For a view of the global state, Bistro's monitoring tools efficiently aggregate across all schedulers.

If data hosts have limited compute resources, we have to move the workers elsewhere. The *single/remote* mode is similar to single/co-locate with one scheduler and multiple workers on dedicated worker hosts, good for load balancing. The *multi/remote* mode has multiple Bistro instances. In this case, similar to multi/co-locate, we can have one Bistro instance per worker to avoid network communication if we assign a fixed partition to each worker. Alternatively we can run schedulers on dedicated hosts for dynamic load balancing. Bistro's Task Runner module can incorporate probabilistic algorithms such as *two-choice* for scalability [30, 32].

3.3 Scheduling Policies

Bistro implements four scheduling policies to prioritize among jobs. Round robin loops through all jobs repeatedly and tries to find one task from each job to execute, until no more task can be selected. Randomized priority is similar to *weighted round robin*. We repeatedly pick a job with probability proportional to its priority, and schedule one of its tasks. Ranked priority sorts jobs by user defined priorities, and schedules as many tasks as possible for the job with the highest priority before moving to the next job. Long tail scheduling policy tries to bring more jobs to full completion via "ranked priority" with jobs sorted by their remaining task count in increasing order.

Round robin and randomized priority approximate fair scheduling, while ranked priority and long tail are prioritized. The latter two can be risky because one job that fails repeatedly on a node can block other jobs from running on that node. All these policies guarantee locally maximal resource utilization in the sense that no more tasks can run on the subtree. Supporting globally maximal resource utilization or globally optimal scheduling polices would require much more complex computation and incur greater scheduling overhead. We have not encountered any production application that requires optimal scheduling.

3.4 Config Loader and Node Fetcher

For each Bistro deployment, users specify the resource and job configuration in a file, a URL, or a Zookeeper cluster [27]. Most of our production systems use Zookeeper for reliability. Bistro's Config Loader refreshes the configuration regularly at runtime.

The resource configuration specifies the tree model, resource capacity at each level, and the scheduling mode. It also specifies a *Node Fetcher* that Bistro will call to retrieve nodes of the tree and periodically refresh them. Different storage types have different node fetcher plugins so Bistro can model their resource hierarchies.

A Bistro deployment can run multiple jobs simultaneously. Each job defaults to the same number of tasks, corresponding to all bottom nodes in the forest model. The job configuration specifies for each job the resource consumption and the task binary. Each tree node has a unique name, and Bistro passes the leaf node name to the command so it will operate on the corresponding data shard. Users can include `filters` in a job configuration to exclude nodes from executing tasks for the job. Filters are useful for testing, debugging, and non-uniform problems.

3.5 Status Store and Fault Tolerance

A task can be in one of several states, including ready, running, backoff, failure, and done. The Status Store module records status updates it receives from Task Runner. It also provides status snapshots to Scheduler and Monitor. A task is uniquely identified by a (job name, node name) pair, and therefore Status Store does not need to track resource model changes.

Status Store can log task statuses to external storage for failure recovery, such as Scuba [7], remote MySQL databases, and local SQLite databases. Scuba is scalable and reliable but has limited data size and retention. Replicated MySQL is more reliable than SQLite, while the latter is more scalable because it is distributed over scheduler hosts. Users choose the external storage based on job requirements. In practice, most failures are temporary. Therefore, users often choose SQLite for the best performance, and write *idempotent* tasks so Bistro can always start over for unrecoverable failures.

Users monitor tasks and jobs through a Web UI, which is an endpoint on Phabricator [3]. The endpoint handles browser requests and aggregates task status data from one or multiple Bistro instances, depending on the scheduling mode. Bistro logs task outputs to local disk, which can also be queried through Phabricator for debugging.

3.6 Task Runner and Worker Management

After scheduling, the Task Runner module starts scheduled tasks, monitors their executions, and updates the Status Store. Task Runner supports both running tasks locally and dispatching it to remote workers. In the latter case, it considers resource constraints on worker hosts as well as *task placement constraints* [23, 32, 35]. Currently, each Bistro instance manages its own set of workers, but it is straightforward to incorporate probabilistic algorithms such as two-choice for better load balancing [30].

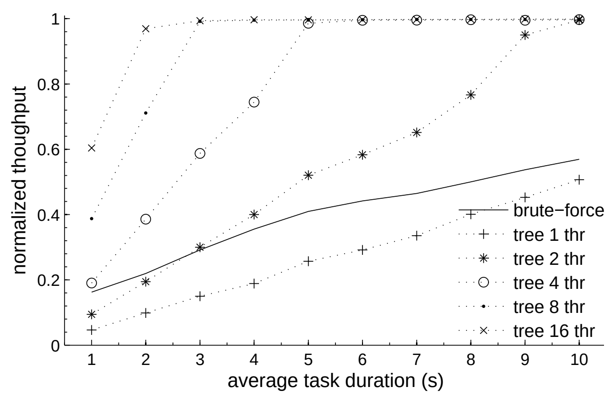
4 Evaluation

We implemented Bistro in C++ with less than 14,000 physical source lines of code (SLOC). As explained in Section 2.1, FIFO-queue-based schedulers lead to almost serial execution for our scheduling problem. Unfortunately most schedulers in the literature as well as commercial schedulers are queue-based, not very interesting for comparison. Therefore, we compare our tree-based scheduling algorithm with the brute-force approach, both explained in Section 2.2. The brute-force approach was widely used in our ad-hoc schedulers before Bistro.

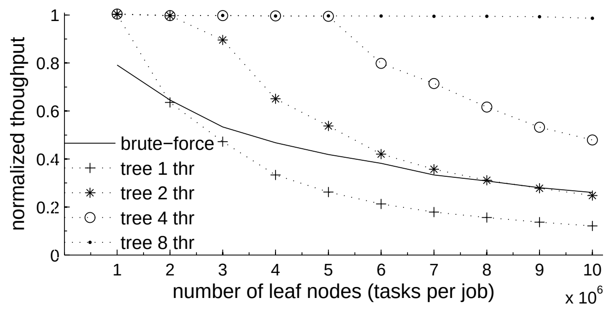
4.1 Microbenchmark

We run our experiments in multi/remote mode with 10 scheduler hosts and 100 worker hosts, each scheduler host has 16 Xeon 2.2 GHz physical cores and 144 GB memory. The resource setup and workload mimic our database scraping production application, which is most sensitive to scheduling overhead because of its short task duration. There are two levels of nodes (or three if adding a root node to enforce global resource constraints), host and database. Each data host has 25 databases; we vary the number of data hosts to evaluate Bistro's scalability. The resource model is partitioned among the 10 schedulers by hashing the data host name. Each task sleeps for a random amount of time that is uniformly distributed over $[0, 2d]$, where d is a job configuration value that we vary.

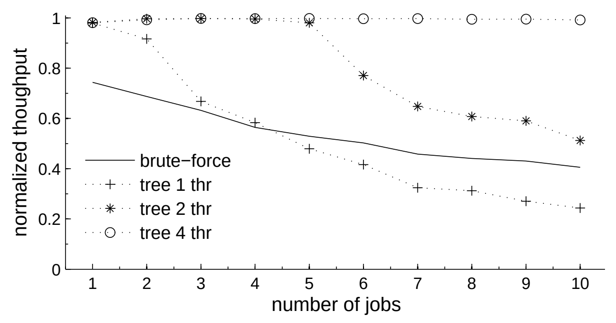
Figure 4 shows the performance results with different resource and job settings. In the legend, "brute-force" refers to brute-force scheduling, and "tree n thr" means tree-based scheduling with n threads, where different threads work on different subtrees, explained in Section 2.2. We show normalized throughput of different scheduling algorithms where 100% means no scheduling overhead, calculated as the maximum number of concurrent tasks allowed by resource constraints divided by the



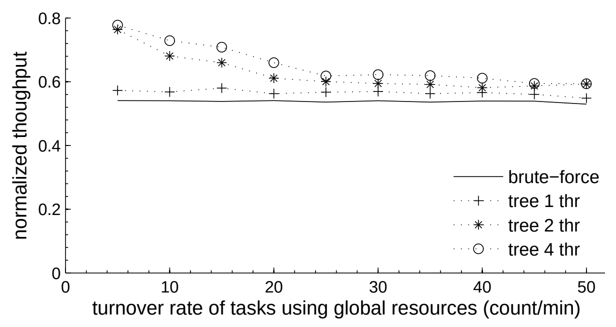
(a) varying task duration (3m nodes, 10 jobs)



(b) varying number of nodes (5 jobs, task duration 5s)



(c) varying number of jobs (3m nodes, task duration 5s)



(d) varying number of tasks using global resources (300k nodes, 5 jobs, task duration 5s)

Figure 4: Scheduling throughput using synthetic workload under different configurations.

average task duration. In all our experiments, the throughput increases slightly for both scheduling algorithms as more tasks finish because there are less tasks to examine in each scheduling iteration. It then drops quickly when there are not enough tasks to occupy all resources. We measure the throughput at the midpoint of each run, and report the average of five runs. The variation is negligible and therefore we do not include error bars in the figure.

Overall, brute-force scheduling shows relatively stable performance because of asynchronous scheduling. If a scheduling iteration takes too long, more tasks finish and release resources, such that the next iteration can schedule more tasks. For a model size of 300k nodes at one scheduler and 5 jobs, each scheduling iteration takes roughly 5 seconds. Tree-based scheduling achieves almost zero overhead when there are enough threads to handle the turnover rate, otherwise the performance drops quickly.

The worst case for the tree-based scheduling is with global resources, shown in Figure 4d. In this case there is only one scheduler, and we add a root node to model a global resource, as explained in Section 2.3. Only one job uses the global resource. We set the job to a high priority and use the *ranked priority* scheduling policy so it runs at the maximum throughput allowed by the global resource constraint. The result shows that the tree-based scheduling performs similarly to brute-force scheduling when the turnover rate of the tasks using global resources is high. This is because when global resources become available for scheduling, Bistro needs to lock the entire tree for scheduling. The overhead of brute-force scheduling is not affected by global resources since it always takes a snapshot of the entire model.

4.2 Production Applications

There are roughly three categories of data store at Facebook, SQL databases for user data, Haystack [12] and F4 [31] for Binary Large OBjects (BLOBs), and Hive [37] and HBase for mostly analytical data. Bistro is currently the only general-purpose scheduler for batch jobs on SQL databases and Haystack/F4. It has also replaced Hadoop for some map-only jobs on HBase, especially on clusters that serve live traffic.

Table 1 shows some of our production applications. The mode column indicates the scheduling mode discussed in Section 3.2. The resource model section shows the characteristics of the resources at each level. The concurrency column shows the resource capacity of that level divided by the default consumption. The change rate column is the average percentage of nodes added or removed per time interval, measured over 30 days in pro-

Application	Mode	Resource Model					Job			
		level	resource type	node count	concur-rency	change rate	concurrent jobs	change rate	task data	avg. task duration
		1	root	1	various	-				
Database	single/	2	host	~10 k	2	2.1%/ hr				
Iterator	remote	3	db	~100 k	1	2.1%/ hr	5	10/ day	300k rows	5 min
Database	single/	1	host	~10 k	1	2.4%/ hr				
Scraping	remote	2	db	~100 k	1	2.3%/ hr	10	5/ hr	1 MB	5 sec
Storage	single/	1	host	~1k	3	<1%/ day				
Migration	co-locate	2	dir	~100 k	1	<1%/ day	1	<1/ day	100 GB	7 hr
		1	host	~1k	1	1.3%/ hr				
Video	multi/	2	volume	~100 k	1	1.2%/ hr				
Re-encoding	remote	3	video	~1 b	1	1.3%/ hr	1	<1/ day	5 MB	20 min
		1	host	~100	3	2.6%/ min				
HBase	multi/	2	region	~1 k	3	3.1%/ min				
Compression	co-locate	3	time	~10 k		8.3%/ min	10	10/hr	3m rows	1 min

Table 1: Summary of some production applications at Facebook. *We show node number in order of magnitude to preserve confidentiality, prefixed by ~. Numbers in the Job section are approximate too for the same reason.*

duction. Node Fetcher refreshes the model every minute by polling the publishing source, the actual change rate can be higher. The job section shows the characteristics of jobs and tasks. The change rate shows the actual number of changes rather than a percentage, because often we just change the configuration of a job instead of adding or removing jobs. Next we discuss these applications.

4.2.1 Database Iterator

Database Iterator is a tool designed to modify rows directly in hundreds of thousands of databases distributed over thousands of hosts. For example, backfilling when new social graph entities are introduced, and data format changes for TAO caching [14]. Because of the lack of batch processing tools for large-scale online databases, this use case motivated the Bistro project.

We use a single Bistro instance since the resource model fits in memory, and some jobs need global concurrency limits. The scheduler sends tasks to a set of workers, i.e., single/remote scheduling mode. We measure 2.1% of hosts and databases change every hour, mostly due to maintenance and newly added databases. Users write their own iterator jobs by extending a base class and implementing `selectRows()`, which picks rows to process, and `processRows()`, which modifies the rows selected. Each task (one per database instance) modifies roughly 300 thousand rows on average. The task duration, while very heterogeneous, averages around five minutes. The total job span is much longer due to long tails.

Before Bistro, Database Iterator ran on a distributed system that executes arbitrary PHP functions asynchronously. Bistro replaced the old system two years ago and it has processed more than 400 iterator jobs and hundreds of trillions of rows. Table 2 compares both systems from the point of view of the production engineers.

Most Database Iterator jobs read and write data on a single database, so our forest resource model is sufficient for protection. Some jobs, however, read from one database and write to many other databases because of different hash keys. Database Iterator supports general "reduce" operations by buffering the intermediate results in RocksDB [5]. One such example was a user data migration. We used two Bistro setups in this case since the source and destination databases are different. The "map" Bistro runs tasks against source databases, writing to RocksDB "grouped by" the hash keys of destination. The "reduce" Bistro runs tasks against destination databases, reading from RocksDB using the corresponding keys. Both sets of databases, totalling hundreds of thousands, were serving live traffic during the migration.

4.2.2 Database Scraping

Database scraping is a common step in ETL (Extract, Transform, Load) pipelines. The resource configuration is similar to database iterator except that we include all replicas because the jobs are read-only; see Section 2.3 on how to model replicas. The scheduling mode is also single/remote but the worker hosts are dedicated to scraping jobs, which allows Bistro to manage their resources and enforce task placement constraints. For example, since scraping jobs run repeatedly, we monitor their resource usage and balance workload among workers.

Before Bistro, we ran scraping on a distributed execution system for time-based jobs, compared in Table 3.

	a proprietary asynchronous execution framework	Bistro
resource throttling	Similar to brute-force scheduling except that tasks have to lock resources themselves by querying a central database, which gets overloaded frequently.	Supports hierarchical resource constraints so tasks need not check resources themselves. Tree-based scheduling achieves better throughput.
model/job updates	Database failovers, new databases, and job changes are frequent, such that queued tasks become outdated quickly, and no job finishes 100% in one run.	Resource models and job configurations are always up to date. All jobs finish completely.
canary testing	No support. Canary testing is crucial because each job runs only once. We make adjustments frequently.	Supports whitelist, blacklist, fraction, and etc. Easy to pause, modify, and resume jobs at runtime.
monitor	Shows various performance counters but not the overall progress, since it does not know the entire set of tasks.	Shows a progress bar per job, with all tasks in different statuses.

Table 2: Feedback from Database Iterator operations team

	an execution framework for time-based jobs	Bistro
LOC	1,150 for an ad-hoc resource manager (RM)	135 for a new node fetcher
resource throttling	The framework applies batch scheduling. Tasks query RM to lock resources. The framework supports only slot based resource allocation on worker hosts.	Supports hierarchical resource constraints for external resources, as well as resource constraints and task placement constraints on workers.
model/job updates	RM takes 45 min to save all resources and tasks to database before scheduling. No update afterwards.	21 sec to get all resources and tasks, and constantly updated.
SPOF	RM, central database, and scheduler all failed multiple times, halting production.	Only the scheduler, which fails over automatically
Priority	No support. Often all jobs get stuck at 99%	Ranked Priority gets jobs to 100% one by one
Job filter	Choices of databases are hard coded in each job.	Automatically tracks database changes.

Table 3: Feedback from Database Scraping operations team

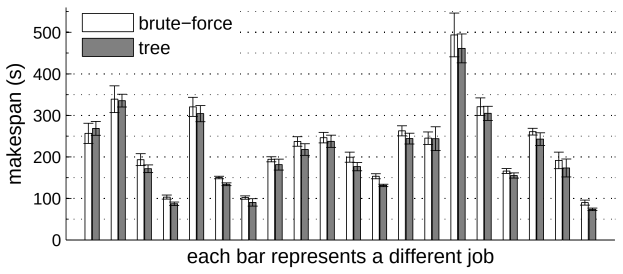
Similar to the asynchronous execution framework used by Database Iterator previously, the system does not consider data resources consumed by tasks, and our engineers had to write an ad-hoc resource manager to lock databases for tasks, which did not scale well. The scheduling algorithm of the distributed execution system is similar to brute-force scheduling, where the scheduler loops through all unfinished tasks repeatedly. Upon start, the resource manager takes 45 minutes to collect all resource and tasks, and log them to a central database. Discounting the 45 minute startup time, we compare the performance of Bistro with brute-force scheduling by replaying a real scraping workload, shown in Figure 5.

There are 20 jobs in the workload, and we are interested in the makespan of the schedule with different job settings. For each experiment, we take 10 runs with different leaf node ordering to average out the makespan variation due to the long tail. The figure shows both the average makespan and the standard deviation. Figure 5a is the makespan of scheduling only one job. Tree scheduling is only slightly better than brute-force scheduling, because long tail tasks often dominate the entire schedule. When there are multiple jobs, on the other hand, tree-based scheduling can be as much as three times faster than brute-force scheduling, shown in Figure 5b. Bistro took over scraping jobs a year ago, which significantly reduced the scraping time, and eased our daily operation.

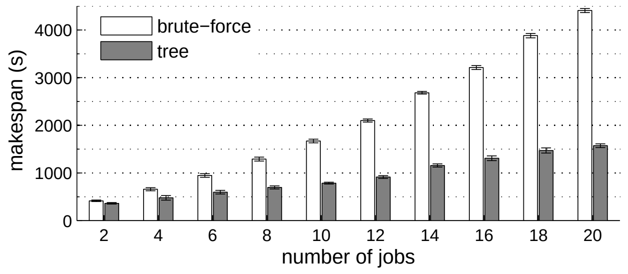
4.2.3 Haystack/F4 Applications

Storage Migration in Table 1 refers to a one-time job that moved sixteen petabytes of videos from a legacy file system to Haystack. Bistro scheduled tasks on each server of the proprietary file system, which sent video files from local directories to Haystack. The destination Haystack servers were not serving production traffic until the migration completed, so we did not throttle tasks for destination resources. At the time of migration, newly uploaded videos already went to Haystack and the old file system was locked. Therefore, the model change rate was low. The overall job took about three months to finish.

Video re-encoding is a project to recompress old user videos with more advanced algorithms to save space without detectable quality degradation. We use fast compression algorithms for live video uploads in order to serve them quickly. Recompressing "cold" videos saves storage space substantially. Compressing each user video using the advanced algorithm takes roughly twenty minutes. Compressing a whole volume can take several days even with multithreading, during which time many videos may

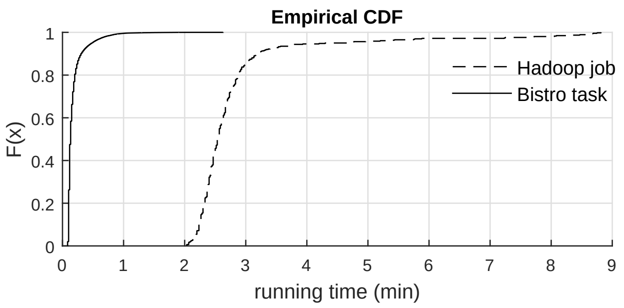


(a) Makespan of scheduling one job

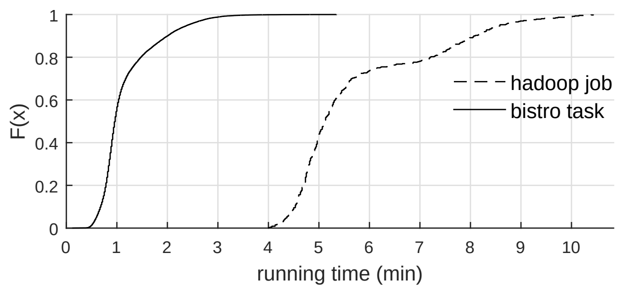


(b) Makespan of scheduling multiple jobs concurrently

Figure 5: Makespan of our Database Scraping jobs



(a) HBase table A



(b) HBase table B

Figure 6: Running time of Bistro tasks vs. Hadoop jobs for HBase Compression

be deleted or updated by users, leading to corrupted metadata. Therefore we process tasks at video level rather than volume level, and let Bistro keeps all videos updated. Each Bistro instance can store a hundred million nodes in memory so we only need a dozen or so scheduler hosts. Compressing a billion videos in a reasonable amount of time, however, requires hundreds of thousands of workers. We are working on a project that harnesses underutilized web servers for the computation.

4.2.4 Hbase Compression

HBase [21] is convenient for long-term data storage with schema. We often need to compress old data to save space. Our performance counter data is one such example. There are three million rows stored every second from all performance counters. We want to preserve old data at coarse granularity. This is done by a compression job that runs every 15 minutes.

We set up Bistro in multi/co-locate mode so each instance processes only local HBase regions. This essentially enforces *data locality*, which benefits our I/O heavy job. In addition to the node fetcher that retrieves HBase hosts and regions, we generate time-based nodes to run the compression job periodically; see Section 2.3 for detail. During our measurements, a few HBase servers did not return region information reliably, so we see a significant percentage of node changes.

Before Bistro, we ran the compression job on Hadoop, compared in Table 4. The HBase cluster is heavily used internally for performance counter queries, and Hadoop often disturbed the query workload. In addition, the MapReduce computation model waits for the slowest task in each phase, which caused 10 % jobs to miss deadlines due to long tail. In contrast, Bistro runs each task independently, and slow data shards may catch up later. Figure 6 shows the empirical CDF of running time for both Hadoop jobs and individual map tasks, measured over one week in a production cluster. There are two different tables we run compression jobs on, with different amounts of data. The figure shows that the Hadoop job histogram starts roughly at the tail of the individual task CDF, confirming the barrier problem.

5 Related Work

Early data center schedulers focus on compute-intensive workload and task parallelism, often associated with High Performance Computing (HPC) and Grid Computing. In these environments, a job can be one task or a series of tasks organized by a workflow, each corresponding to a different binary. The scheduler assigns tasks to different machines, considering their resource requirements, placement constraints, and dependencies. Scheduling objectives include minimiz-

	Hadoop	Bistro
duration	Depends on the slowest map task	Each task runs independently
skipped jobs	10%, which happens when the previous job did not finish in its time window.	No missing job since tasks run independently. The percentage of skipped tasks are less than 0.1%
Resource throttling	No support. Starting a new job often slows down the entire cluster, causing service outage	No detectable query delay with our hierarchical resource constraints.
Data locality	99%. Speculative execution kicks in for long tails, which makes it worse since our tasks are I/O bound.	100% in multi/co-locate mode.
other issues	Difficulty of deployment, no counters/ monitoring/ alerting, no flexible retry setting, no slowing down/ pausing/ resuming, no job filtering.	All supported.

Table 4: Feedback from Hbase Compression operations team

ing makespan, minimizing mean completion time, maximizing throughput, fairness, or a combination of these; see [15, 19, 33] for good reviews of the topic. There are many open source and commercial schedulers for HPC and Grid workloads [4, 25, 29, 36].

Since MapReduce [18], data-intensive job scheduling has become the norm in the literature [10,13,22,26,28,32, 34, 38–40, 42, 43]. However, since MapReduce is an offline data processing framework, all schedulers of which we are aware assume no external resources needed and focus on interchangeable compute resources of the offline cluster. Typically the scheduler assigns tasks to workers from either a global queue or a set of queues corresponding to different jobs.

The *data locality* problem can be viewed as a special and simple case of our data resource constraints, where we want to place a task on the worker that hosts the data [6, 16, 28, 39, 42]. However, since queue-based scheduling is ill suited to non-interchangeable resources (Section 2.1), these schedulers treat data locality as a preference rather than a hard constraint. For example, Delay Schedule skips each job up to k times before launching its tasks non-locally [42]. A variation of the scheduling problem considers *task placement constraints*, where a task can only be assigned to a subset of workers due to dependencies on hardware architecture or kernel version [23, 32, 35]. Task placement constraints are associated with jobs, so we cannot use them to enforce data-local tasks. Mitigating stragglers or reducing job latency is another popular topic [8–10, 17, 20, 43].

Bistro moves one step further by scheduling data-parallel jobs against online systems directly. It treats resources at data hosts, either local or remote, as first-class objects, and can strictly enforce data locality and other hierarchical constraints without sacrificing scheduling performance. Many of its data-centric features are not common in the literature, e.g., tree-based scheduling, updating resources and jobs at runtime, flexible and elastic setup.

Regarding performance, Bistro is optimized for high throughput, handling highly concurrent short-duration tasks. Many schedulers assume long running tasks, and sacrifice scheduling delays for the optimal schedule. For example, Quincy takes about one second to schedule a task in a 2,500-node cluster [28]. In contrast, our Database Scraping workload has a turnover rate of thousands of tasks per second. Recently, Sparrow aimed at query tasks of millisecond duration, and reduced scheduling latencies for fast responses [32, 40]. Bistro can incorporate Sparrow in its Task Runner module to reduce the task dispatching latency.

6 Conclusion

Data center scheduling has transitioned from compute-intensive jobs to data-intensive jobs, and it is progressing from offline data processing to online data processing. We present a tree-based scheduler called Bistro that can safely run large-scale data-parallel jobs against live production systems. The novel tree-based resource model enables hierarchical resource constraints to protect online clusters, efficient updates to capture resource and job changes, flexible partitioning for distributed scheduling, and parallel scheduling for high performance. Bistro has gained popularity at Facebook by replacing Hadoop and custom-built schedulers in many production systems. We are in the process of migrating more jobs to Bistro, and plan to extend its resource and scheduling models in the next version.

7 Acknowledgements

We thank Terence Kelly, Haibo Chen, Sanjeev Kumar, Benjamin Reed, Kaushik Veeraraghavan, Daniel Peek, and Benjamin Wester for early feedback on the paper. Anonymous reviews helped us improve the paper further, and we appreciate our shepherd Haryadi Gunawi for working on the final version.

References

[1] Apache Thrift. thrift.apache.org.

[2] Bistro. bistro.io.

[3] Phabricator. phabricator.org.

[4] Platform LSF. www.ibm.com.

[5] RocksDB. rocksdb.org.

[6] Under the hood: Scheduling MapReduce jobs more efficiently with Corona. www.facebook.com.

[7] L. Abraham, , et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, 2013.

[9] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *NSDI*, 2014.

[10] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[11] K. Bandaru and K. Patiejunas. Under the hood: Facebook's cold storage system. code.facebook.com.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[14] N. Bronson et al. Tao: Facebooks distributed data store for the social graph. In *Usenix Annual Technical Conference*, 2013.

[15] P. Brucker. *Scheduling algorithms*. Springer, 2007.

[16] F. Chung, R. Graham, R. Bhagwan, S. Savage, and G. M. Voelker. Maximizing data locality in distributed systems. *Journal of Computer and System Sciences*, 72(8):1309–1316, 2006.

[17] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job scheduling strategies for parallel processing*, pages 1–34. Springer, 1997.

[20] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[21] L. George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.

[23] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[24] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[25] R. L. Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995.

[26] 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 *NSDI*, 2011.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[28] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[29] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[30] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[31] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebooks warm blob storage system. In *OSDI*, 2014.

[32] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, 2013.

[33] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[35] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Symposium on Cloud Computing*, 2011.

[36] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, pages 307–350. MIT press, 2001.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *International Conference on Data Engineering*, 2010.

[38] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Symposium on Cloud Computing*, 2012.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop YARN: Yet another resource negotiator. In *Symposium on Cloud Computing*, 2013.

[40] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[41] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[42] 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 *EuroSys*, 2010.

[43] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.