Big Data over Networks

Edited by
Shuguang Cui, Texas A&M University, College Station, TX, USA

Alfred Hero, University of Michigan, Ann Arbor, MI, USA

Zhi-Quan Luo, University of Minnesota, Minneapolis, MN, USA

José M. F. Moura, Carnegie Mellon University, Pittsburgh PA, USA

Contents

Illustrations

Tables

Boxes

List of contributors

Ganesh Ananthanarayanan
Microsoft Research, Redmond WA

Ishai Menache
Microsoft Research, Redmond WA

Part I

Mathematical foundations

Part II

Big data over communication/computer networks

1 Big data analytics systems

Abstract

Performing timely analysis on huge datasets is the central promise of big data analytics. To cope with the high volumes of data to be analyzed, computation frameworks have resorted to "scaling out" – parallelization of analytics which allows for seamless execution across large clusters. These frameworks automatically compose analytics jobs into a DAG of small tasks, and then aggregate the intermediate results from the tasks to obtain the final result. Their ability to do so relies on an efficient scheduler and a reliable storage layer that distributes the datasets on different machines.

In this chapter, we survey the above two aspects, *scheduling* and *storage*, which are the foundations of modern big data analytics systems. We describe their key principles, and how these principles are realized in widely-deployed systems.

1.1 Introduction

Analyzing large volumes of data has become the major source for innovation behind large Internet services as well as scientific applications. Examples of such "big data analytics" occur in personalized recommendation systems, online social networks, genomic analyses, and legal investigations for fraud detection. A key property of the algorithms employed for such analyses is that they provide better results with increasing amount of data processed. In fact, in certain domains (like search) there is a trend towards using relatively simpler algorithms and instead rely on more data to produce better results.

While the amount of data to be analyzed increases on the one hand, the acceptable time to produce results is shrinking on the other hand. Timely analyses have significant ramifications for revenue as well as productivity. Low latency results in online services leads to improved user satisfaction and revenue. Ability to crunch large datasets in short periods results in faster iterations and progress in scientific theories.

To cope with the dichotomy of ever-growing datasets and shrinking times to

Term	Description
Task	Atomic unit of computation with a fixed input
Phase	A collection of tasks that can run in parallel, e.g., map, aggregate
Workflow	A directed acyclic graph denoting how data flows between phases
Job	An execution of the workflow
Block	Atomic unit of storage by the distributed file system
File	Collection of blocks
Slot	Computational resources allotted to a task on a machine

Table 1.1 Definitions of terms used in data analytics frameworks.

analyze them, analytics clusters have resorted to *scaling out*. Data is spread across many different machines, and the computations on them are executed in *parallel*. Such scaling out is crucial for fast analytics and allows coping with the trend of datasets growing faster than Moore's law's increase in processor speeds.

Many *data analytics frameworks* have been built for large scale-out parallel executions. Some of the widely used frameworks are MapReduce [1], Dryad [2] and Apache Yarn [3]. The frameworks share important commonalities, and we will use the following common terminology throughout this chapter. Frameworks compose a computation, referred to as a *job*, into a DAG of *phases*, where each phase consists of many fine grained *tasks*. Tasks of a phase have no dependencies among them and can execute in parallel. The job's input (*file*) is divided into many *blocks* and stored in the cluster using a distributed file system. The input of each task consists of one or more blocks of a file. A centralized scheduler assigns a *compute slot* to every task[1]. Tasks in the input phase produce *intermediate* outputs that are passed to other tasks downstream in the DAG. Table 1.1 summarizes the terminology.

As a concrete example, consider an analysis of web access logs (of many TB's). Each row in the log consists of a URL being accessed (e.g., www.cnn.com) and details about its access (e.g., time of access, user accessing it); the aim of the analysis is to count the number of accesses of each web URL to understand popularities. The distributed file systems splits the access logs into small 256 MB blocks and stores them over the different machines. A job to analyze them would consist of two phases. Every task in the first phase would read a block of data and generate a $\langle url, 1\rangle$ tuple for each row in the log (the "1" indicates a single access). Each task in the second phase is responsible for some fraction of the URL's and collects the corresponding tuples from the first phase's outputs. It then sums up the accesses per URL and produces the output.

The key benefit of the analytics frameworks is their ability to scale out to

[1] Slot is a virtual token, akin to a quota, for sharing cluster resources among multiple jobs. One task can run per slot at a time.

thousands of commodity machines for computation as well as storage. They do so by automatically and logically dividing data as well as analyses on them into fine-grained units of blocks and tasks, respectively. As commodity machines are susceptible to failures and unavailabilities, the frameworks are resilient to these failures to ensure data availability as well as successful execution of the tasks. In this chapter, we survey the important aspects of systems running big-data analytics frameworks. In particular, we focus on two fundamental requirements: (i) scheduling jobs efficiently, and (ii) managing data storage across distributed machines. Scheduling principles and solutions are described in Section 1.2, and storage architectures in Section 1.3. Additional related and upcoming topics are briefly outlined in Section 1.4.

1.2 Scheduling

Analytics frameworks typically use a centralized scheduler where all the tasks in the cluster are queued. The scheduler manages the machines in the cluster, where each machine has a *worker* process. The worker processes send periodic *heartbeats* to the scheduler informing it of the status of running tasks as well as the resource usages and general health of the machines. The scheduler aggregates the information from the heartbeats and makes scheduling decisions to allocate tasks to machines. When a machine does not send heartbeats for a certain period of time, the scheduler assumes the machine to be lost and does not schedule any further tasks to it. It also reschedules the unfinished tasks on the machine elsewhere.

Recent designs have advocated (and deployed) a hierarchical two-level scheduling model where the individual jobs talk to a central "resource manager" to register their demands and queue their tasks (e.g., Mesos [4], Apache Yarn [3]). The resource manager, then, allocates compute slots to the different tasks. Nonetheless, the abstraction of a central scheduler allocating slots to tasks holds.

Our focus in this section is on describing the principles and logic behind the central scheduler decisions. Scheduling models have been considered for numerous applications in both academia in industry for at least half a century. The emergence of big-data jobs, running on large-scale clusters, has brought novel challenges that cannot be readily solved by traditional scheduling mechanisms. We therefore highlight below the distinctive aspects of scheduling big-data jobs. In Section 1.2.1 we outline how fairness policies are adapted to deal with several complexities such as multiple resource types, intra-job dependencies and machine fragmentation. In Section 1.2.2 we discuss how scheduling solutions accommodate *placement constraints*, such as the necessity to place job tasks alongside their input data. In Section 1.2.3, we describe scheduling solutions that incorporate different objective criteria, such as mean completion time and finishing jobs by pre-specified deadlines. Finally, in Section 1.2.4 we present the problem of *stragglers* (tasks running on slow machines), and the main approaches for mitigating their effect.

1.2.1 Fairness

Allocating resources to multiple jobs has been a fundamental problem in shared computer systems. While several design criteria have been considered for scheduling mechanisms, *fairness* is perhaps the most prominent one. There are different definitions for what constitutes a fair allocation; we do not attempt to cover all here, but rather focus on the ones that are used in modern big-data clusters. *Max-min fairness* is one such policy – it simply maximizes the minimum allocation across users. If each user has enough "demand" for the resource, the policy boils down to allocating the resource in equal shares among users. A natural generalization of this policy is *weighted* max-min fairness, in which each user receives resources in proportion to its pre-specified weight. Several algorithms have been proposed to implement weighted max-min fairness in different engineering contexts, (e.g., deficit round robin [5] and weighted fair queuing [6]). However, the original algorithms do not cover important considerations associated with bing-data jobs, which require adjustments of both the fairness policies and the algorithms to sustain them. We list below the main considerations and how they have been addressed.

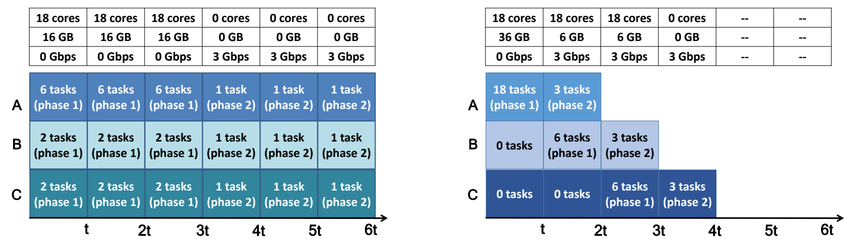
Multiple resources. Big-data jobs such as map-reduce jobs utilize different resources, such as CPU, memory and I/O resources. Consider the following numeric example (taken from [7]): Suppose the system consists of 9 CPUs, 18 GB RAM. Two users ought to share the system – user A runs tasks with demand vector (1CPU, 4GB), and user B runs tasks with demand vector (3CPUs, 1GB) each. Note that each task of user A consumes 1/9 of the total CPU and 2/9 of the total memory. Each task from user B consumes 1/3 of the total CPUs and 1/18 of the total memory. In order to divide the system resources, [7] defines the notion of *dominant resource*, which is the resource which is utilized the most (percentage-wise) by the tenant. In our example, user A's dominant resource is memory, while user B's dominant resource is CPU. The authors in [7] propose a new policy, Dominant Resource Fairness (DRF) which extends max-min fairness to the multiple resource case. DRF simply applies max-min fairness across users' dominant shares. In the example, the DRF allocation would be 3 tasks for user A and 2 tasks for user B, which will equalize the dominant resource shares of A and B (2/3 of the RAM for A, and 2/3 of the CPU for B). The DRF solution has some appealing properties such as pareto efficiency, bottleneck fairness, sharing incentive, envy-free, and strategy proofness. Without going into details, the first two properties indicate that the DRF solution is efficient and fair, while the latter properties mean that users have incentives to participate in a system which divides resources according to DRF, and further would not game the system. [7] also proposes an iterative greedy algorithm, which gears the allocation towards the DRF solution. The algorithm was implemented in the Mesos cluster resource manager, and leads to better fairness compared to schemes which divide single-dimension slots.

Intra-job dependencies. As mentioned before, big-data jobs often consist of multiple phases with data dependencies among them. A fair but naive scheduler that does not take the inner structure of jobs into account might actually result in an unfair allocation of resources. For simplicity, we focus here on the issues arising in the MapReduce framework, although the problems and solutions can apply to other analytics frameworks. Hadoop launches reduce tasks for a job as soon as some mappers finish, so that reduces can start copying the maps outputs while the remaining maps are still running. Assuming that the cluster is initially not congested, a large job with many mappers would keep getting reduce slots from a naive fair scheduler. With a lack of preemption mechanism, the problem here is that these reduce slots would not be released until all mappers finish, because only then can the reduce function be carried out. This means that small jobs that arrive to the system during the map phase of the big job might be starved. [8] proposes the following solution to this problem: Split reduce tasks into two logically distinct types of tasks - copy tasks and compute tasks, and have separate admission control mechanisms for both types. In particular, [8] limits the total number of slots for reduce-compute on each machine, and further sets a per-job upper bound on the reduce-copy slots on each machine. The combination of these mechanisms constraints the amount of simultaneous resources given to jobs without needing to use more aggressive preemption mechanisms.

Fragmentation and over-allocation. As described above, the basic fair schedulers divide resources to slots, where each slot represents some amount of memory and CPU; slots are allocated to the job which is furthest from its (weighted) fair-share. Such bundling of resources is obviously inefficient, because it might lead to either wasting resources in some dimensions or to over allocation if some dimensions are overlooked. DRF partially resolves these issues by considering the allocation problem as a multi-dimensional one. However, DRF might leave some resources idle, since it only attempts to maximize the the dominant resource share across all jobs. Further, DRF does not explicitly take into account the available capacity in each machine, but rather considers the total available capacity for each resource.

To overcome fragmentation, [9] proposes a multi-resource scheduler that *packs* tasks to machines based on their vector of requirements. The underlying scheduling mechanism is based on an online (multi-dimensional) *bin-packing* heuristic, which attempts to utilize the available resources in each machine as much as possible. The tasks are the “balls” and the “bins” are machines over time. Intuitively, because the bin packing heuristic attempts to pack balls to bins while using a minimal number of bins, resources are well utilized and job can finish quickly.

We illustrate the advantages of a packing-based scheduler via the following example [9]. Consider a cluster with 18 cores and 36GB of memory. Three jobs A, B and C have one phase each consisting of 18, 6, and 6 tasks, respectively.



18 cores	18 cores	18 cores	0 cores	0 cores	0 cores
16 GB	16 GB	16 GB	0 GB	0 GB	0 GB
0 Gbps	0 Gbps	0 Gbps	3 Gbps	3 Gbps	3 Gbps

18 cores	18 cores	18 cores	0 cores	--	--
36 GB	6 GB	6 GB	0 GB	--	--
0 Gbps	3 Gbps	3 Gbps	3 Gbps	--	--

Figure 1.1 Impact of packing on DAGs of tasks compared to fairness based allocation (like DRF). Jobs A and B have two phases with a strict barrier between them. The tables on top show utilization of the resources. Packing results in better utilization of resources using complementarity of task requirements.

Each task in job A requires 1 core and 2GB of memory, while job B's and C's tasks require 3 cores and 1GB of memory. Assume all tasks run for t time units. DRF will schedule 6 tasks of job A and 2 tasks each of jobs B and C, at a time, giving each job a dominant resource share of $\frac{1}{3}$. This leads to an average job duration of $3t$. Such an allocation, however, leaves 20GB of memory in the cluster idle. A scheduler that packs the tasks schedules all tasks of job A initially because they use up all the cores and memory in the cluster, followed by tasks of job B and then job C. This leads to job durations of t, $2t$ and $3t$ for the jobs for an average of $2t$ and a 33% improvement over DRF.

The advantages of packing tasks carry over to jobs with multiple *phases* (e.g., a map-reduce computation consists of a map phase and a reduce phase). Extending the above example, let all three jobs have two phases separated by a barrier, i.e., tasks of their second phase begin only after all tasks of their first phase finish. For simplicity, assume tasks of the second phase require 1Gbps of network resource but no cores or memory. Suppose that the cluster has total network bandwidth of 3Gbps; tasks in the first phase have no network usage but need cores and memory as listed above.[2] Let the jobs have 18, 6, and 6 tasks in their first phase, as before, and 3 tasks each in their second phases (each task, again, of t time units). Figure 1.1 compares DRF allocation with a packing based scheduler. Again, by scheduling tasks of job A initially and not wasting resources, the packing based scheduler is able to exploit the complementarity in resource requirements between tasks of the first and second phases. This results in all three jobs finishing faster than under a DRF-based allocation. The average job duration is $3t$ versus $6t$, or a speed up of 50% compared to DRF. The makespan improves by 33% from $6t$ to $4t$.

[9] incorporates the packing-based scheduler in a system called Tetris. Tetris achieves substantial *makespan* reductions (over 30%) over existing schedulers.

[2] Note that this example is typical of a map-reduce computation; map tasks are indeed CPU and memory intensive while reduce tasks are network-intensive.

When resources on a machine become available, Tetris chooses a task which can fit on the machine, and whose score is maximal; the score is an inner product of the residual capacity of the machine and the pick resource requirement of the task. This choice maximizes the utilization and diminishes fragmentation. Tetris takes fairness into account by considering only a fraction $(1 - f)$ of the tasks, which are ordered according to a fairness criterion (e.g., max-min), and then picking the highest-score job among this subset.

Placement constraints. Finally, scheduling algorithms have to take into account placement constraints, such as scheduling tasks where their input data is. Since this topic is quite broad, we address it in a separate subsection below.

1.2.2 Placement constraints

Our focus so far has been on scheduling under fairness considerations. In its simplest form, sustaining fairness means that each user gets "enough" cluster resources according to pre-specified fairness criterion (e.g., max-min fairness, or weighted fair sharing). However, in the big-data analytics context, it greatly matters exactly *which* resources are given to jobs; jobs might be constrained on the set of resources (e.g., machines) they can run on. Placement constraints can be roughly classified into three classes [10]: (i) hard; (ii) soft and (iii) combinatorial. Examples of hard constraints include jobs that must run on machine with public IP address, particular kernel version, specific SKU or hardware (e.g., GPU machines or machines with SSDs). The most prominent soft constraint is data locality; while job tasks could be scheduled on machines which do not necessarily hold the required data for the task, a "cost" is incurred, in the form of additional latency and excess network bandwidth usage. Combinatorial constraints specify rules for a collection of machines that are used for the job, such as fault tolerance constraints. We survey below the related work for each class.

Hard constraints. [11] provides a comprehensive study of Google workloads, and in particular the impact of hard placement constraints on task scheduling delays. It turns out that these constraints increase task scheduling delays by a factor of 2–6. Accordingly, that paper develops a methodology that enables predicting the impact of hard constraints on task scheduling delays. Specifically, a metric termed Utilization Multiplier (UM) is introduced; this metric measures for each relevant resource the utilization ratio between tasks with constraints and the average utilization of the resource. Accordingly, the higher UM is, the higher is the expected scheduling delay. Finally, [11] describes how to incorporate placement constraints into existing performance benchmarks, so that they are properly taken into account when evaluating new cluster architectures or scheduling mechanisms. [10] proposes scheduling algorithms for dealing with hard placement constraints, while sustaining some notion of fairness between users (jobs). In particular, this paper defines a new notion of fairness, termed Constrained Max-Min Fairness (CMMF), which extends max-min fairness while taking into account that some jobs cannot run on some machines. A CMMF allocation is a machine assignment in which it is not possible to increase the minimum allocation within any subset of users by reshuffling the machines given to these users. A CMMF allocation has appealing properties such as incentive compatibility with regard to a user reporting his placement constraints. Calculating a CMMF solution, however, might be too costly in practice, as it involves solving a sequence of LPs. Consequently, the authors propose a simple greedy online scheduler called Choosy: Whenever a resource becomes available, it is assigned to the user with the lowest current allocation, which is allowed to get that resource according to its placement constraints. Perhaps surprisingly, Choosy achieves allocations which are very similar to those of a CMMF solution, and as a result, the latencies of jobs are on average at most 2% higher than the CMMF solution.

Soft constraints. We focus here on a particular soft constraint,termed *data locality*. Preserving data locality means placing the computation of the job near its input data. Locality is important because network bandwidth might be a bottleneck, hence transferring input data over the network might lead to substantial task delays [12]. Simply scheduling jobs near their input data would in general violate the fairness requirements across multiple users. Hence, it is crucial to design schedulers which are both *fair* and locality-aware. [12] uses a simple algorithm to address the tension between locality and fairness: when a job that should be scheduled next according to the underlying fairness criterion cannot launch a local task, it waits for a small amount of time, while letting other jobs use the available resource. If after a pre-specified timeout the job still cannot execute tasks locally, it is assigned available resources which are not necessarily local to the task. Accordingly, this basic algorithm is termed *delay scheduling*.

The authors in [12] extend the basic algorithm described above to address several practical considerations, including rack locality, hotspots, long tasks and more; we omit the details here for brevity. The resulting scheduling system is termed the Hadoop Fair Scheduler (HFS). One important extension in HFS is a two-layer hierarchical architecture. The higher layer divides resources between organizations in a fair manner, while the lower layer divides resources between the organization's jobs according to a local policy (e.g., FIFO or fair share). Evaluation on Facebook workloads shows that delay scheduling achieves nearly 100% locality, leading to substantial improvements in job response times (especially for small jobs) and throughput (especially for IO-heavy workload).

The underlying engineering principle which makes delay scheduling appealing in practice, is that slots running Hadoop free up in a *predictable rate*. The time-threshold used for delay scheduling is set accordingly – i.e., taking into account typical execution times of tasks, it is expected that some local slots would free up by the time-threshold. We now describe an alternative scheduler which does not rely on such predictions. [13] maps the problem of scheduling with locality and fairness considerations into a *min-flow problem* on a graph. The underlying idea here is that every scheduling decision can be assigned a cost. E.g., there is a data transfer cost when task is scheduled far from its input, there is a cost for killing a task that takes too long, etc. These costs are embedded in a graph, where the nodes are the tasks and physical resources such as racks and servers; The graph-based algorithm operates by assigning a unit flow to each task node in the system (either running or waiting for execution). There is a a single sink to which all flows are drained. The flows could traverse through either a path consisting of resources (meaning that the respective task is scheduled on the respective resources), or through their job's "unscheduled" node, in which case the task is (still) not executed. Fair sharing constraints are added by setting lower and upper bounds on the edges from the unscheduled node to the sink. The cost of killing a job is modeled by gradually increasing the costs of all edges related to the task, but the edge that "connects" the task node to its executing resource. The authors manage to scale the solution to large instances by reducing the effective number of nodes through cost aggregation techniques. [13] implements this graph-based algorithm in a system called Quincy. Quincy is evaluated against a queue-based algorithm, and shows sizable gains in both amount of data transferred and throughput (up to 40%).

It is of interest to qualitatively compare the delay-scheduling approach to the graph min-flow approach. The graph-based algorithm is more sophisticated, and can incorporate the *global* state of the cluster and multiple cost considerations (e.g., further delaying a waiting task vs. killing a task that has been running for a long time). While fast min-cost procedures can be used for the solution, scale issues might arise when the cluster and number of jobs is very large. Another potential weakness of the graph-based algorithm is that it is greedy by nature, and optimizes based on the current snapshot of the cluster. The delay-scheduling algorithm is arguably simpler and easier to implement in production. The framework does not "code" all cost tradeoffs, which could be problematic, especially in more complicated job models which do not obey systematic execution patterns. Another advantage of delay scheduling is that it uses knowledge about task durations which allows it to take into account the future evolvement of the cluster state (rather than act solely as a function of the present state).

Combinatorial constraints. Combinatorial constraints may arise, e.g., due to security and fault tolerance considerations. An example of such constraints could be - "not more than x% of each job's tasks should be allocated in the same fault domain". [14] considers the problem of assigning physical machines to applications while taking into account fault tolerance performance. [14] tackles the difficult combinatorial problem by using a smooth convex cost function, which serves as a proxy for fault tolerance. In particular, this function incentivizes stripping machines belonging to the same application across fault domains. [14] also deals with the fundamental cost tradeoff between fault tolerance and bandwidth consumption - a solution that spreads machines across fault domains is usually bad in terms of the bandwidth consumption, as communication might heavily use the network core. Accordingly, [14] introduces an additive penalty to the above cost function, which carefully balances the optimization of the two metrics.

1.2.3 Additional system-wide objectives

Sustaining fairness has been the primary objective of cluster schedulers. However, there are additional objectives that may be as important. For example, Tetris [9] (described above) optimizes the *makespan* by packing tasks to machines. Tetris also has a knob for reducing *average job completion time*. It sorts jobs by estimated remaining work and chooses tasks whose jobs have the least remaining work; this is commensurate with the Shortest Remaining Time (SRTF) scheduling discipline, which is known to be effective in minimizing job completion times. Tetris combines the packing score (described in Section 1.2.1) and the remaining-work score into a single score per-task, with a tuneable parameter that controls how much weight is given to each component.

Another potential system-wide objective is meeting job *deadlines*. Big-data jobs are often used for business critical decisions, hence have strict deadlines associated with them. For example, outputs of some jobs are used by business analysts; delaying job completion would significantly lower their productivity. In other cases, a job computes the charges to customers in cloud computing settings, and delays in sending the bill might have serious business consequences. Recently, several works have addressed the resource allocation problem where meeting job deadlines is the primary objective. Jockey [15] dynamically predicts the remaining run time at different resource allocations, and chooses an allocation which is expected to meet the job's deadline without wasting unnecessary resources. Jockey treats each job in isolation with the assumption that allocating resources "economically" would lead to solutions that satisfy multiple jobs deadlines. [16] proposes an interface in which users submit resource requirements along with deadlines, and the scheduler objective is to meet the jobs deadlines. The resource requirements are *malleable* in the sense that the scheduler can often pick alternative allocations that would lead to job completion. As such, the scheduler can choose the specific allocations which will allow multiple jobs to finish without violating the cluster's capacity. In a similar context, [17–19] develop scheduling algorithms for the model where jobs have different values, and the scheduler objective is to maximize the value of jobs that complete before the deadline. [17–19] provide constant-factor approximation algorithms for the problem. As may be expected, the approximation quality improves when deadlines are less stringent with respect the available cluster capacity.

1.2.4 Stragglers

Classic scheduling models usually assume that a task has a predictable/deterministic execution time, regardless of the physical server chosen to run it; e.g., a task would complete in 30 seconds if run on any server at any point in time. However, in the context of current big-data analytics framworks, there are plenty of factors that could vary the actual running time of the task. The original map-reduce paper [1] introduces the notion of a *straggler* - "a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation". [1] lists a couple of reasons for stragglers, such as machine with bad disk, other tasks running on the same machine, bugs in machine initialization, etc. The paper proposes a general mechanism to deal with stragglers: When a MapReduce job is close to completion, *backup* executions of the remaining in-progress tasks are scheduled. The task completes whenever either the primary or backup execution completes. This assignment of backup task(s) is often termed *speculative* execution (or speculative task). The simple mechanism described above can lead to substantial improvement in job response times (up to 44% according to [1]. While the principle of speculative execution is natural for dealing with stragglers, it is not a-priori clear under which conditions duplication should take place. Obviously, this knob can lead to negative congestion effects if used injudiciously. We survey below recent works that have addressed this problem for different scenarios.

The Hadoop scheduler uses a simple threshold rule for speculative execution. It calculates the average progress score of each category of tasks (maps and reducers) and defines a threshold accordingly: When the task's progress score is less than the average minus some parameter (say 0.2), it is considered a straggler. [20] emphasizes the shortcomings of this mechanism: It does not take into account machine heterogeneity, it ignores the hardware overhead in executing stragglers, it does not take into account the *rate* at which the task progresses; further, it assumes that tasks finish in waves and that tasks require the same amount of work, assumptions that are imprecise even in homogenous environment. Accordingly, [20] designs a new algorithm Longest Approximate Time to End (LATE). As can be inferred from its name, the distinctive feature of LATE compared to previous approaches is that it selects tasks for speculative execution based on the estimated time left, rather than based solely on the progress itself. In particular, it uses $(1 - \mathit{ProgressScore})/\mathit{ProgressRate}$ to rank each task, where *ProgressScore* is the progress estimator given by Hadoop. The authors point out that this formula has some drawbacks, yet it is simple and works well in most cases. In addition to the above, LATE also sets a hard constraints on the number of speculative tasks allowed in the system. When a slot becomes available and the number of speculative tasks running is below the threshold, LATE launches a copy of the highest-ranked task whose progress rate is low enough (below some pre-specified threshold). Finally, LATE does not launch copies on machines that are statistically slow. Implementation of LATE on an EC2 virtual cluster with 200 machines shows 2X improvement in response time compared to Hadoop scheduler.

Mantri [21] proposes a refined method for limiting the resource overhead of speculative execution. A speculative task is scheduled if the probability of it finishing before the original task is high. Mantri provides more knobs with regard to the speculative execution itself – it supports duplication (including multiple duplicates), kill-restart and pruning based on re-estimations of progress. In addition, Mantri extends the scope of speculative execution to preventing and mitigating stragglers and failures. It does so in different ways. First, it replicates critical task output to prevent situations where tasks wait for lost data; as before the decision to do so is based on estimating the probability of the bad event and evaluating the tradeoff between re-computation and excess resource cost. Second, Mantri executes tasks in descending order of their input size. The idea here is that tasks with heavier input take longer to execute, and thus should be prioritized since what matters is the makespan of the tasks. Finally, Mantri chooses the placement of reduce tasks in a *network-aware* manner, taking into account the data that has to be read across the network and the available bandwidth. Mantri has been deployed on Bing's cluster with thousands of machines, and exhibited median job speedup of 32%.

The aforementioned techniques for handling stragglers generally operate at the task level – e.g., rank tasks based on progress or completion time, limit the number speculative tasks, etc. Instead, [22] proposes to apply different mitigation logic at the *job* level. The main observation here is that for *small* jobs, i.e., jobs that consist of few tasks, a system can tolerate a more aggressive issuing of duplicates. In particular, [22] uses full *cloning* of tasks belonging to small jobs, avoiding waiting and speculating, hence improving the chances that the tasks output is ready on time. Analyzing production traces, [22] shows that the smallest 90% of jobs consume only 6% of the total resources. Hence, cloning can be done to a substantial chunk of the jobs with a rather small resource overhead. On the algorithmic front, cloning introduces the following challenge: efficient cloning means that the system uses the (intermediate) output data of the upstream clone that finishes first; however, this creates contention for the IO bandwidth at that upstream clone, since multiple downstream clones require its data. [22] solves this problem through a heuristic reminiscent to delay scheduling, hence named *delay assignment*. Every downstream clone waits for a small amount of time (a parameter ω of the heuristic) to see if it can get an exclusive copy of the intermediate data from a preassigned upstream clone. If the downstream clone does not get its exclusive copy by ω, it reads from the upstream clone that finishes first. [22] implements the cloning framework in a system called Dolly. The paper reports substantial improvements in completion time of small jobs – 34%-46% compared to LATE and Mantri, using less than 5% extra resources.

1.3 Storage

Storage is a key component of big data analytics clusters. Clusters store petabytes of data, distributed over many machines. Key to analytics is *reliable* and *efficient* storage of the data. Data should be accessible even in the face of machine failures (which are common) and should be read/written efficiently (without significant overheads).

Distributed file systems present the abstraction of a single unified storage to applications by abstracting away as much of the details of the individual

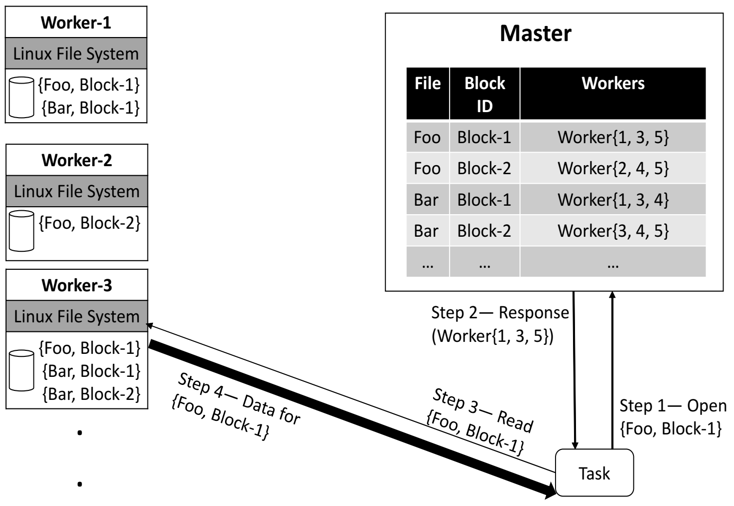

Worker-1
Linux File System
{Foo, Block-1}
{Bar, Block-1}
Worker-2
Linux File System
{Foo, Block-2}
Worker-3
Linux File System
{Foo, Block-1}
{Bar, Block-1}
{Bar, Block-2}
Master
File
Block ID
Workers
Foo Block-1 Worker{1, 3, 5}
Foo Block-2 Worker{2, 4, 5}
Bar Block-1 Worker{1, 3, 4}
Bar Block-2 Worker{3, 4, 5}
Step 2— Response (Worker{1, 3, 5})
Step 1— Open {Foo, Block-1}
Step 3— Read {Foo, Block-1}
Step 4— Data for {Foo, Block-1}
Task

Figure 1.2 Architecture of Distributed File Systems (DFS).

storage machines as possible; the machines store the data on their local disks. We describe how distributed file systems enable such storage while presenting the unified storage abstraction in Section 1.3.1. An important and upcoming class of storage solutions, driven by falling RAM prices is in-memory caching. Such caching offers much faster access to data compared to disks but require special handling along certain aspects, which we cover in Section 1.3.2.

1.3.1 Distributed file system

Typically, a *file* in a distributed file system is divided into many smaller *blocks*, which are stored on different machines. Every file has a unique identifier, and every block within a file is also referenced uniquely. The distributed file system is oblivious to the local storage mechanisms used by the disk subsystems of the machines. The machine could employ various disk architectures like just-bunch-of-disks, RAID or simple striping. It could also employ its own error-recovery and caching mechanisms.

Architecture

Distributed file systems have a centralized architecture: a single central master that maintains metadata about the blocks in the cluster. Figure 1.2 presents a simple architectural representation. Metadata information for every block contains the locations storing it, its last access time, size, the file it is part of, and so forth. Each of the machines have a file system worker that is responsible for managing the local data blocks. It interfaces with the local storage subsystem and allows the master to be oblivious. It also periodically informs the master about the available space on the machine, and other performance characteristics.

The widely used distributed file systems provide interfaces similar to the POSIX interface [23] for reading and writing data. File namespaces are hierarchical and can be identified via path names. The operators supported are *create*, *open*, *read*, *write*, *close* and *delete*.
Writing: Writing data to the file system involves the following steps.

1. Application calls *create* to the master with a file name. The master responds positively with a handle if the filename is not in use currently.
2. Application uses the handle to *open* a file to the master. The master checks its metadata to pick the machine to write the first block of the file. It returns the details of the corresponding worker process.
3. Application calls *write* to the worker to store its data until the size is equal to the upper-limit for a block. Once the block is full, the worker informs the application.
4. Application gets another worker to write its next block of data. A file is written only one block at a time.
5. When all the blocks are written, the application calls *close* on the file to the master.

While the master controls where the data is written and automatically obtains metadata information, the actual writes themselves go directly to the worker. Such a design helps significantly with the scalability of the master. Imagine an alternate design where the application provides its data to the master and lets it send to the worker machine transparently. While such an approach would marginally simplify the application, it places a huge burden on the master.
Append-only: Modern distributed file systems do not support *updates* to the blocks, only *appends*. While applications can add blocks to an existing file, they cannot modify any of the existing blocks. Such an append-only decision is suited for these clusters where data is written once and read many times over, and rarely updated.

While the single master scales well since it only deals with requests as opposed to data, its scalability can be improved by federating its namespace. Recall that the files are organized as a hierarchical namespace. Therefore, the master can be scaled by partitioning the namespace appropriately. Thus each sub-space in the hierarchy will be independently handled which reduces the load on any single machine. Partitioning happens based on hierarchical boundaries as well as popularity of data access.
Reading: Reading data from the file system involves the following steps.

1. Application calls *open* to the master with the desired file name. The master returns with the handle.
2. Application calls *read* to the master and optionally provides block identifiers. The master responds with the set of worker machines on which the desired blocks of data are stored.
3. Application contacts the worker nodes and requests data blocks.

4. After reading all the blocks, application calls *close* on the file to the master.

As earlier, the master only deals with providing applications with the locations of the workers, and does not directly pass the data blocks. However, different from above, applications can read from multiple workers in parallel or in series. Not allowing applications to write in parallel simplifies the design of the master as it does not have to deal with load balancing the write speeds across different machines.

Authentication: An important piece missing in the above workflows for reading and writing data is authentication. Files should be written to and read by only authenticated users. This is highly important in multi-tenant clusters. For this purpose, the master enforces explicit access controls and requires users to authenticate themselves. Further, when writing data, applications are provided an explicit token that helps authenticate themselves to the workers. The tokens are time-specific and cannot be reused.

Fault tolerance

Fault tolerance is an important concern for distributed file systems. The master is a single point of failure as the loss of its metadata will require hours to rebuild as each worker has to inform the new master of the blocks stored on its machine. To prevent such expensive rebuilding, file systems use two approaches: periodic checkpointing and hot standby. The master periodically checkpoints its metadata to persistent storage. In addition there is also a hot standby that can take over immediately when the primary master fails. The standby picks up from the persisted metadata state.

Data is replicated (typically, three times) to provide fault tolerance. Replication, however, is transparent to the application. The worker to which a block is written contacts the master to obtain another location where data is to be replicated. The second location, in turn, creates another replica of the data.

File systems also support reverting to previous versions via snapshots. Snapshots allow users to revert to previous versions. Traditionally, snapshotting is implemented by time-stamping and storing the old copy of a block whenever it changes. The append-only model greatly simplifies snapshotting by avoiding the need to store any old copies. Whenever blocks are appended to a file, a simple log that timestamps the action is sufficient to roll back to any desired time. Of course, when files are deleted, they still have to be persisted for rollback.

Variable replication

In time, replication is used for both performance as well as reliability. In these distributed clusters, reading from local storage is often faster than reading from remote machines. Thus, analytics frameworks schedule their tasks on the machines that contain data locally. Achieving locality for concurrently executing tasks, however, is dependent on the number of replicas as well as cluster utilization.

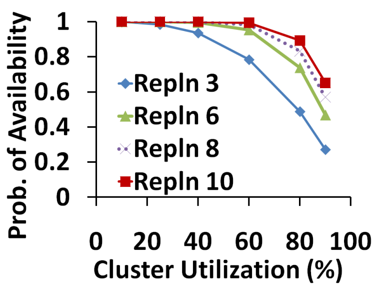


Figure 1.3 The probability of finding a replica on a free machine for different values of file replication factor and cluster utilization.

We present a simple analysis that demonstrates the intuition behind how increased replication reduces contention. With m machines in the cluster, k of which are available for a task to run, the probability of finding one of r replicas of a file on the available machines is $1-(1-\frac{k}{m})^r$. This probability increases with the replication factor r, and decreases with cluster utilization $(1-\frac{k}{m})$.

Figure 1.3 plots the results of a numerical analysis to understand how this probability changes with replication factors and cluster utilizations. At a cluster utilization of 80%, for example, with the current replication factor (r=3), we see that the probability of finding a replica among the available machines is less than half. Doubling the replication factor raises the probability to over 75%. Even at higher utilizations of 90%, a file with 10 replicas has a 60% chance of finding a replica on a free machine. By replicating files proportionally to their number of concurrent accesses, the chances of finding a replica on a free machine improves.

Therefore, some file systems perform automatic variable replication of files based on their popularity. Using historical access statistics—total number of accesses as well as number of concurrent accesses—systems create extra replicas of popular data blocks. Such variable replication ensure sufficient number of replicas that help in providing locality for future tasks concurrently acccesing the same data block.

1.3.2 In-memory storage

Hardware trends, driven by falling costs, indicate a steep increase in memory capacities of large clusters. This presents an opportunity to store the input data of the analytics jobs in memory and speed them up. As mentioned earlier, data-intensive jobs have a phase where they process the input data (e.g., *map* in MapReduce [1], *extract* in Dryad [2]). This phase simply reads the raw input and writes out parsed output to be consumed during further computations. Naturally, this phase is IO-intensive. Workloads from Facebook and Microsoft Bing datacenters, consisting of thousands of servers, show that this IO-intensive phase constitutes 79% of a job's duration and consumes 69% of its resources.

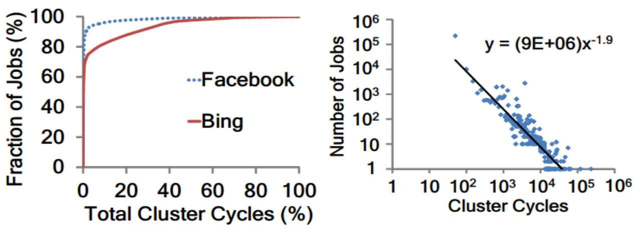


Figure 1.4 Power-law distribution of jobs in the resources consumed by them. Power-law exponents are 1.9 and 1.6 in the two traces, when fitted with least squares regression [22].

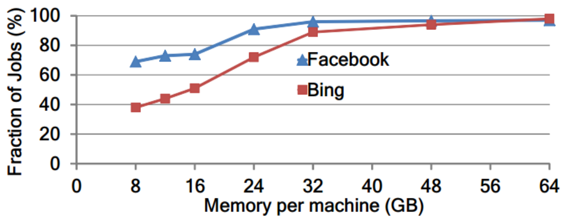


Figure 1.5 Fraction of active jobs whose data fits in the aggregate cluster memory, as the memory per machine varies [24].

Storing *all* the data currently present in disks is, however, infeasible because of the three orders of magnitude difference in the available capacities between disk and memory, notwithstanding the growing memory sizes. Datacenter jobs, however, exhibit a heavy-tailed distribution of their input sizes thus offering the potential to cache the inputs of a large fraction of jobs.

Heavy-tailed input sizes: Workloads consist of many small jobs and relatively few large jobs. In fact, 10% of overall data read is accounted by a disproportionate 96% and 90% of the smallest jobs in the Facebook and Bing workloads. As Figure 1.4 shows, job sizes indeed follow a power-law distribution, as the log-log plot shows a linear relationship [22].

The skew in job input sizes is so pronounced that a large fraction of active jobs can simultaneously fit their entire data in memory.[3] Consider a simple simulation that looks at jobs in the order of their arrival time. The simulator assumes the memory and computation slots across all the machines in the cluster to be aggregated. It loads a job's entire input into memory when it starts and deletes it when the job completes. If the available memory is insufficient for a job's entire input, none of it is loaded. Figure 1.5 plots the results of the simulation. For the workloads from Facebook and Bing, 96% and 89% of the active jobs respectively can have their data entirely fit in memory, given an allowance of 32GB memory per server for caching [24].

[3] By active jobs we mean jobs that have at least one task running.

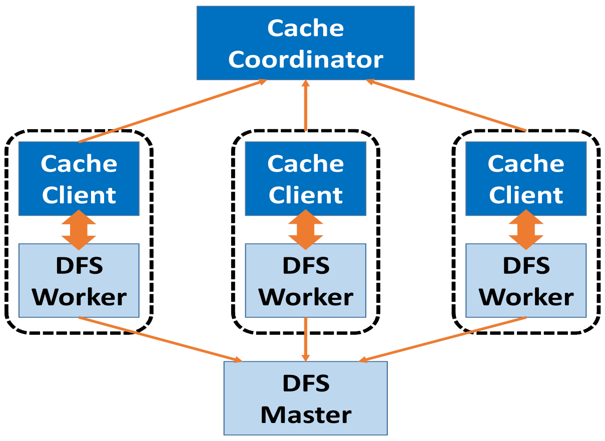


Figure 1.6 Coordinated cache architecture [24]. The central *coordinator* manages the distributed *clients*. Thick arrows represent data flow while thin arrows denote meta-data flow.

Coordination architecture

Data is cached on the different distributed machines, and globally coordinated. Global coordination enables the abstraction of viewing different input blocks in unison to implement cache replacement policies. A coordinated cache infrastructure, (a) supports queries for the set of machines where a block is cached, and (b) mediates cache replacement globally across the machines (covered shortly).

The architecture of a caching system consists of a central *coordinator* and a set of *clients* located at the storage nodes of the cluster (see Figure 1.6). Blocks are added to the cache clients. The clients update the coordinator when the state of their cache changes (i.e., when a block is added or removed). The coordinator uses these updates to maintain a mapping between every cached block and the machines that cache it. As part of the map, it also stores the file that a block belongs to.

The client's main role is to serve cached blocks, as well as cache new blocks. Blocks are cached at the *destination*, i.e., the machine where the task executes as opposed to the *source*, i.e., the machine where the input is stored. This allows an uncapped number of replicas in cache, which in turn increases the chances of achieving memory locality especially when there are hotspots due to popularity skew [25]. Memory local tasks contact the local cache client to check if its input data is present. If not, they fetch it from the distributed file system (DFS). If the task reads data from the DFS, it puts it in cache of the local cache client and the client updates the coordinator about the newly cached block. Data flow is designed to be local in the architecture as remote memory access could be constrained by the network.

Fault tolerance: The coordinator's failure does not hamper the job's execution as data can always be read from disk. However, the architecture includes a secondary coordinator that functions as a cold standby. Since the secondary coordinator has no cache view when it starts, clients periodically send updates to the coordinator informing it of the state of their cache. The secondary coordinator uses these updates to construct the global cache view. Clients do not update their cache when the coordinator is down.

Resilient distributed datasets

An important class of applications *reuse* data across computations. Examples include iterative machine learning and graph algorithms like K-Means clustering and logistic regression. In-memory storage is an efficient way to support such data reuse. Storing it to the distributed file system incurs substantial overheads due to data replication and serialization, which can dominate execution times. Reuse also occurs in interactive data explorations when the same input is loaded once and used by many queries.

To enable efficient reuse of data, there is an abstraction proposed called *resilient distribtued datasets (RDDs)*. RDDs are fault-tolerant parallel data structures that let users store data in memory, optimize their placement and execute generic queries over them. RDDs can persist input data as well as intermediate results of a query to memory. A simple example to illustrate the programming model is as follows.

```
   lines = spark.textFile(“hdfs://”)
errors = lines.filter(_.startsWith(“ERROR”)
errors.persisit()
```

The main challenge in designing RDDs is providing efficient fault tolerance. Traditional options like replication or logging updates across machines are resource-expensive and time-consuming. RDDs provide fault tolerance by storing the *lineage* of the dataset. The lineage represents the set of steps (e.g., map, filter) used to create the RDD. Therefore, when a machine fails, its data can be regenerated by replaying the corresponding lineage steps. Maintaining lineage is a simple and inexpensive way to provide fault tolerant in-memory storage *without* replication.

The Spark system, that supports RDDs, has a simple Scala programming interface for managing RDDs. Spark's execution engine automatically converts the Scala program to a data parallel job of many parallel tasks, and executes them on the cluster. Figure 1.7 illustrates the execution.

Cache replacement for parallel jobs

Maximizing cache hit-ratio does not minimize the average completion time of parallel jobs. In this section, we explain that using the concept of wave-width of parallel jobs and use it to devise the LIFE cache replacement algorithm.

All-or-nothing property: Achieving memory locality for a task will shorten its completion time. But this need not speed up the job. Jobs speed up when an entire *wave-width* of input is cached (Figure 1.8)[4]. The wave-width of a job

[4] This assumes similar task durations, which turns out to be true in practice. The 95th

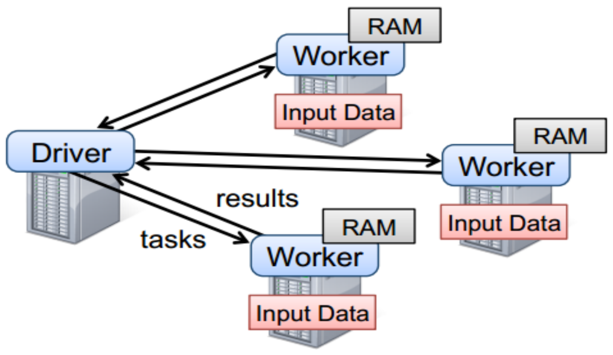


Figure 1.7 Resilient Distributed Datasets in Spark [26].

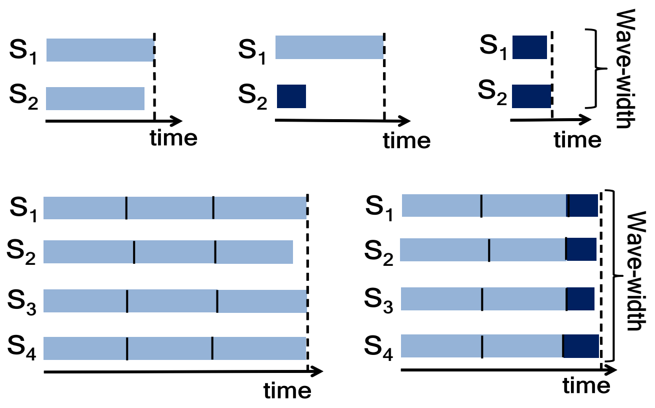


Figure 1.8 Example of a single-wave (2 tasks, simultaneously) and multi-wave job (12 tasks, 4 at a time). S_i's are slots. Memory local tasks are dark blocks. Completion time (dotted line) reduces when a wave-width of input is cached.

is defined as the number of simultaneously executing tasks. Therefore, jobs that consist of a single wave need 100% memory locality to reduce their completion time. We refer to this as the *all-or-nothing* property. Jobs consisting of many waves improve as we incrementally cache inputs in multiples of their wave-width. In Figure 1.8, the single-waved job runs both its tasks simultaneously and will speed up only if the inputs of both tasks are cached. The multi-waved job, on the other hand, consists of 12 tasks and can run 4 of them at a time. Its completion time improves in steps when any 4, 8 and 12 tasks run memory locally.

Average completion time: In a cluster with multiple jobs, favoring jobs with the smallest wave-widths minimizes the average completion time of jobs. Assume

percentile of the coefficient-of-variance ($\frac{stdev}{mean}$) among tasks in the data-processing phase (e.g., map) of jobs is 0.08.

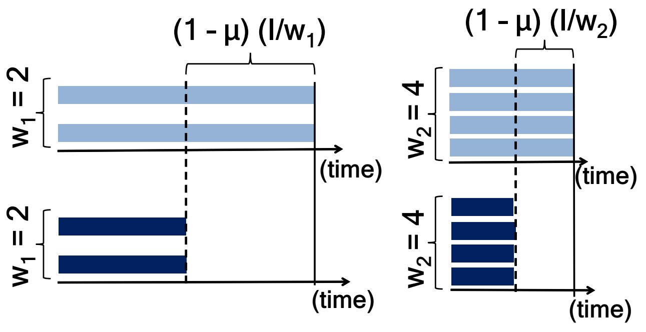


Figure 1.9 Gains in completion time due to caching decreases as wave-width increases. Solid and dotted lines show completion times without and with caching (for two jobs with input of I but wave-widths of 2 and 4). Memory local tasks are dark blocks, sped up by a factor of μ.

that all jobs in the cluster are single-waved. Every job j has a wave-width of w and an input size of I. Let us assume the input of a job is equally distributed among its tasks. Each task's input size is $\left(\frac{I}{w}\right)$ and its duration is proportional to its input size. Memory locality reduces its duration by a factor of μ. The factor μ is dictated by the difference between memory and disk bandwidths, but limited by additional overheads such as deserialization and decompression of the data after reading it.

To speed up a single-waved job, we need I units of cache space. On spending I units of cache space, tasks would complete in $\mu\left(\frac{I}{w}\right)$ time. Therefore the saving in completion time would be $(1-\mu)\left(\frac{I}{w}\right)$. Counting this savings for every access of the file, it becomes $f\,(1-\mu)\left(\frac{I}{w}\right)$, where f is the frequency of access of the file. Therefore, the ratio of the job's benefit to its cost is $f\,(1-\mu)\left(\frac{1}{w}\right)$. In other words, it is directly proportional to the frequency and inversely proportional to the wave-width. The smaller the wave-width, the larger the savings in completion time per unit of cache spent. This is illustrated in Figure 1.9 comparing two jobs with the same input size (and of the same frequency), but wave-widths of 2 and 4. Clearly, it is better to use I units of cache space to store the input of the job with a wave-width of two. This is because its work per task is higher and so the savings are proportionately more. Note that even if the two inputs are unequal (say, I_1 and I_2, and $I_1 > I_2$), caching the input of the job with lower wave-width (I_1) is preferred despite its larger input size. Therefore, in a cluster with multiple jobs, *average completion time is best reduced by favoring the jobs with smallest wave-widths* (LIFE).

This can be easily extended to a multi-waved jobs. Let the job have n waves, c of which have their inputs cached. This uses $cw\left(\frac{I}{nw}\right)$ of cache space. The benefit in completion time is $f\,(1-\mu)\,c\left(\frac{I}{nw}\right)$. The ratio of the job's benefit to its cost is $f\,(1-\mu)\left(\frac{1}{w}\right)$, hence best reduced by still picking the jobs that have the smallest wave-widths.

1.4 Concluding remarks

The concepts surveyed in this chapter enable the execution of big-data analytics on commodity clusters. These concepts address the principal aspects of large-scale computation, such as data-locality, consistency and fault-tolerance, provided by analytics frameworks while abstracting the details away from the user/application. While the concepts were presented in the context of batch analytics, we would like to point out that the template of jobs composed as DAG of tasks is by no means restricted to batch analytics alone. Many *interactive* frameworks [27–29] are using this template for fast interactive analytics, while also employing in-memory caching techniques explained in the chapter. Recently, *stream* processing—continuous processing of data as it streams in—is also employing this template. These systems divide the streaming input data into time buckets, and execute jobs on each of the time buckets. Thus, we expect the solutions presented here to remain the relevant core as analytics frameworks evolve in the future with new models and requirements of big data processing.

Our focus here has been on scheduling and storage, which can be viewed as basic ingredients of any such systems. We conclude this chapter by briefly outlining additional elements which are subject to ongoing research.

Approximate computing

An emerging class of applications is geared towards *approximation.* Approximation jobs are based on the premise that providing a timely result, even if on only part of the dataset is *good enough.* Approximation jobs are growing in interest to cope with the deluge in data since processing the entire dataset can take prohibitively long.

Approximation is explored along two dimensions—time to produce the result (deadline), and error (accuracy) in (of) the result [30, 31].

1. *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time limit. Such jobs are common in real-time advertisement systems and web search engines. Generally, the job is spawned on a large dataset and accuracy is proportional to the fraction of data processed.
2. *Error-bound* jobs strive to minimize the time taken to reach a specified error limit in the result. Again, accuracy is measured in the amount of data processed (or tasks completed). Error-bound jobs are used in scenarios where the value in reducing the error below a limit is marginal, e.g., counting of the number of cars crossing a section of a road to the nearest thousand is sufficient for many purposes.

Approximation jobs raise several challenges that are part of ongoing research. Sample selection is crucial towards meeting the desired approximation bounds. The selected sample impacts the achieved accuracy within a deadline, as well as the ability to meet an error bound. Also, understanding the query patterns and data access characteristics help provide *confidence intervals* on the produced result. Designing smart samplers that balance the accuracy of the result, sample storage space and function generically is an open problem.

Another important problem is straggler mitigation. Approximation jobs require schedulers to *prioritize the appropriate subset of their tasks* depending on the deadline or error bound. Optimally prioritizing tasks of a job to slots is a classic scheduling problem with known heuristics. Stragglers, on the other hand, are unpredictable and require dynamic modification to the priority ordering according to the approximation bounds. The challenge is to achieve the approximation bound by dynamically weighing the gains due to speculation against the cost of using extra slots.

Energy efficiency

Energy costs are considered to be a major factor in the overall operational costs of datacenters. Therefore, there is strong motivation for utilizing clusters in *energy-efficient* manner. In our context of big-data jobs, there is an opportunity of saving on energy costs, as some of the workloads can tolerate delays in execution, and can be scheduled when the energy costs are cheaper. The associated decision process involves several factors, such as time-varying energy prices, the existence of renewable energy sources, cooling overhead, and more; see e.g., [32–34] and references therein.

On top of the original scheduling decision of when to run jobs, there are additional control knobs that may be incorporated. For example, tasks can be executing using different power levels [35], and perhaps some resources can be turned off when overall demand is not high [36]. At a higher level, it may be possible to migrate jobs across datacenters in order to execute the jobs where energy is currently cheap [37].

Pricing

In Section 1.2 we have briefly discussed mechanisms which incentivize users to utilize resources fairly and truthfully. Such mechanisms are mostly relevant for internal enterprize clusters (a.k.a. private clouds). In the public cloud context, however, the main provider objective is maximizing the net profit, consisting of revenues minus operating costs. Current cloud providers rent Virtual Machines (VMs) on an hourly basis using a variety of pricing schemes (such as on-demand, reserved, or spot pricing, see, e.g., [38]); recent research has accordingly been devoted to maximizing the customer's utility in face of the different pricing schemes (see, e.g., [39] and reference therein). From the provider perspective, public cloud offerings, such as Amazon EC2, have recently incorporated an additional premium price for using big-data services such as Hadoop. Nevertheless, an interesting research direction is to investigate more specific pricing schemes, e.g., such that take into account job SLAs, deadlines, etc.

References

[1] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” in *ACM EuroSys*, 2007.

[3] V. Vavilapalli and et. al, “Apache hadoop yarn: Yet another resource negotiator,” in *ACM SoCC*, 2013.

[4] 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 *USENIX NSDI*, 2011.

[5] M. Shreedhar and G. Varghese, “Efficient fair queuing using deficit round-robin,” *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996.

[6] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types.” in *NSDI*, vol. 11, 2011, pp. 24–24.

[8] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Job scheduling for multi-user mapreduce clusters,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[9] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, “Multi-resource packing for cluster schedulers.” ACM SIGCOMM, 2014.

[10] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, “Choosy: Max-min fair sharing for datacenter jobs with constraints,” in *Proceedings of the 8th ACM European Conference on Computer Systems.* ACM, 2013, pp. 365–378.

[11] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, “Modeling and synthesizing task placement constraints in google compute clusters,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing.* ACM, 2011, p. 3.

[12] 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 *Proceedings of the 5th European conference on Computer systems.* ACM, 2010, pp. 265–278.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, 2009, pp. 261–276.

[14] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, “Surviving failures in bandwidth-constrained datacenters,” in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication.* ACM, 2012, pp. 431–442.

[15] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, “Jockey: guaranteed job latency in data parallel clusters,” in *Proceedings of the 7th ACM european conference on Computer Systems.* ACM, 2012, pp. 99–112.

[16] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, “Reservation-based scheduling: If you’re late don’t blame us!” in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[17] N. Jain, I. Menache, J. Naor, and J. Yaniv, “Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters,” in *SPAA*, 2012, pp. 255–266.

[18] B. Lucier, I. Menache, J. Naor, and J. Yaniv, “Efficient online scheduling for deadline-sensitive jobs: extended abstract,” in *SPAA*, 2013, pp. 305–314.

[19] P. Bodík, I. Menache, J. S. Naor, and J. Yaniv, “Brief announcement: deadline-aware scheduling of big-data processing jobs,” in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 2014, pp. 211–213.

[20] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments.” in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[21] 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*, vol. 10, no. 1, 2010, p. 24.

[22] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Effective straggler mitigation: Attack of the clones.” in *NSDI*, vol. 13, 2013, pp. 185–198.

[23] “Posix,” http://pubs.opengroup.org/onlinepubs/9699919799/.

[24] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, “Pacman: Coordinated memory caching for parallel jobs,” 2012.

[25] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, “Scarlett: Coping with skewed popularity content in mapreduce clusters,” in *ACM EuroSys*, 2011.

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

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” in *Proc. of the 36th Int’l Conf on Very Large Data Bases*, 2010, pp. 330–339.

[28] R. 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.

[29] S. Agarwal, B. Mozafari, A. Panda, M. H., S. Madden, and I. Stoica, “Blinkdb: Queries with bounded errors and bounded response times on very large data,” in *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[30] J. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, “Imprecise computations.” in *IEEE*, 1994.

[31] S. Lohr, “Sampling: Design and analysis,” in *Thomson*, 2009.

[32] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, “Energy efficiency for large-scale mapreduce workloads with significant interactive analysis,” in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 43–56.

[33] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser, “Renewable and cooling aware workload management for sustainable

data centers," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 175–186.

[34] A. Beloglazov, R. Buyya, Y. C. Lee, A. Zomaya, *et al.*, "A taxonomy and survey of energy-efficient data centers and cloud computing systems," *Advances in Computers*, vol. 82, no. 2, pp. 47–111, 2011.

[35] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 157–168.

[36] A. Gandhi, V. Gupta, M. Harchol-Balter, and M. A. Kozuch, "Optimality analysis of energy-performance trade-off for server farm management," *Performance Evaluation*, vol. 67, no. 11, pp. 1155–1171, 2010.

[37] N. Buchbinder, N. Jain, and I. Menache, "Online job-migration for reducing the electricity bill in the cloud," in *NETWORKING 2011*. Springer, 2011, pp. 172–185.

[38] "EC2 pricing," http://aws.amazon.com/ec2/pricing/.

[39] I. Menache, O. Shamir, and N. Jain, "On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud," in *11th International Conference on Autonomic Computing (ICAC)*, 2014.

Part III

Big data over social networks

Part IV

Big data over bio networks