Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can

Virajith Jalaparti[†] Peter Bodik[‡] Ishai Menache[‡]
Sriram Rao[‡] Konstantin Makarychev[‡] Matthew Caesar[†]

[†]University of Illinois, Urbana-Champaign [‡]Microsoft

Abstract

To reduce the impact of network congestion on big data jobs, cluster management frameworks use various heuristics to schedule compute tasks and/or network flows. Most of these schedulers consider the job input data fixed and greedily schedule the tasks and flows that are ready to run. However, a large fraction of production jobs are *recurring* with predictable characteristics, which allows us to plan ahead for them. Coordinating the placement of data and tasks of these jobs allows for significantly improving their network locality and freeing up bandwidth, which can be used by other jobs running on the cluster. With this intuition, we develop Corral, a scheduling framework that uses characteristics of future workloads to determine an offline schedule which (i) *jointly* places data and compute to achieve better data locality, and (ii) isolates jobs both spatially (by scheduling them in different parts of the cluster) and temporally, improving their performance. We implement Corral on Apache Yarn, and evaluate it on a 210 machine cluster using production workloads. Compared to Yarn's capacity scheduler, Corral reduces the makespan of these workloads up to 33% and the median completion time up to 56%, with 20-90% reduction in data transferred across racks.

CCS Concepts

•Networks → Data center networks; •Computer systems organization → Cloud computing;

Keywords

Data-intensive applications; Cluster schedulers; Joint data and compute placement; Cross-layer optimization

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom

 ISBN 978-1-4503-3542-3/15/08…$15.00

DOI: http://dx.doi.org/10.1145/2785956.2787488

1 Introduction

Over the past decade, large big data compute clusters running thousands of servers have become increasingly common. Organizations such as Facebook, Google, Microsoft, and Yahoo! have deployed highly scalable data-parallel frameworks like MapReduce [25] and Cosmos [20] to process several petabytes of data every day, running tens of thousands of jobs [5].

These big data jobs are often constrained by the network for several reasons. First, they involve network-intensive stages such as shuffle and join, which transfer large amounts of data across the network. Second, while there is full bisection bandwidth within a rack, modern data intensive computing clusters typically have oversubscription values ranging from 3:1 to 10:1 from the racks to the core [22, 23, 24]. This trend is likely to continue in the future to limit operational costs. Finally, a large fraction of the cross-rack bandwidth (up to 50%) can be used for background data transfers [22], reducing the bandwidth available for these jobs even more.

Existing cluster schedulers try to overcome this problem by optimizing the placement of compute tasks or by scheduling network flows, while assuming that the input data locations are *fixed*. However, different stages in these big data jobs can still run into network bottlenecks. Techniques like delay scheduling [48] and flow-based scheduling [36] try to place individual tasks (e.g., maps) on the machines or racks where most of their input data is located. As this data is spread over the cluster randomly in a distributed file system (such as HDFS [6]), the subsequent job stages (e.g., shuffle) have to read data using cross-rack links, which are often heavily congested [22]. Recently introduced techniques like ShuffleWatcher [16] attempt to localize the shuffle of a job to one or a few racks, but end up using the cross-rack bandwidth to read input data. The benefits from using network flow-level techniques, such as in Varys [24] or Baraat [26], are also limited, as they only schedule network transfers after both the source and destination are fixed. Sinbad [22] schedules flows around network contention but its benefits are limited to file system writes.

However, a large number of business-critical jobs are *recurring*, with pre-defined submission times and predictable resource requirements, allowing us to carefully place job input data to improve network locality. For example, it has been reported that production workloads contain up to 40%

recurring jobs, which are run periodically as new data becomes available [15, 30]. In fact, we show that future job characteristics (e.g., input data size) for such jobs can be predicted with an error as low as 6.5% (Section 2). Using these characteristics, we can plan *ahead* and determine where the input data of a job can be placed in the cluster, e.g., in a particular subset of racks. By coordinating such data placement with job task placement, most *small* jobs can be run entirely within one rack with all their tasks achieving rack-level network locality and no oversubscription. As these small jobs do not use cross-rack links, larger jobs running across multiple racks will also benefit due to more available bandwidth. Further, by running jobs entirely within a subset of racks rather than across the whole cluster, we can achieve better performance isolation across jobs.

With this intuition, we design Corral, a scheduling framework that exploits the predictability of future workloads to *jointly* optimize the placement of data and compute in big data clusters, and improve job performance. Corral includes an *offline planner* which uses the characteristics of jobs (e.g., amount of data read, CPU and memory demands) that will run on the cluster to determine which set of racks should be used to run each job and when the job should start. Job execution is *decoupled* from the planner and only uses the planner's output as *guidelines* for placing data and tasks.

The goal of the offline planner is to optimize job execution while addressing important challenges. First, it needs to determine which racks to assign to a job to maximize network locality while providing sufficient parallelism. Second, it needs to ensure that jobs and their input data are spread across the cluster without overloading any part of it. We formulate this planning problem as a malleable job scheduling problem [19, 27, 45] and develop simple heuristics to solve it efficiently. We show that these heuristics achieve performance close (within 3-15%) to the solution of a Linear Program (LP) relaxation, which serves as a lower-bound to *any* algorithm used for the planning problem (which assigns resources to jobs at the granularity of racks). Using a data imbalance cost (Section 4.5) allows our heuristics to ensure that the input data is balanced across all racks in the cluster.

The offline planner optimizes for both recurring jobs and jobs whose arrival and characteristics are known in advance. Corral executes *ad hoc* jobs, whose characteristics cannot be predicted, using otherwise idle cluster resources. As the jobs planned with Corral finish faster, additional resources will be available for these ad hoc jobs and hence, they will also finish earlier. The techniques in Corral apply to both simple MapReduce jobs as well as complex DAG-structured workloads such as Hive [44] queries.

We note that during job execution, we do *not* exclusively allocate whole racks to a single job. Instead, given the set of racks assigned to a job by the offline planner, one replica of job input data is placed within those racks and all tasks of the job are *restricted* to run within those racks. This forces all job data transfers to stay within those racks. The remaining slots within these racks are used by ad hoc jobs and other planned jobs assigned to the same racks. When a significant fraction of machines in these racks fail (beyond a configured threshold), Corral ignores these constraints and uses any available resources in the cluster to run the planned jobs.

We have implemented Corral as an extension of Yarn [8] and deployed it on a 210 machine cluster. Using four different workloads including traces from Yahoo [21], Microsoft Cosmos (Section 6), and Hive queries derived from TPC-H benchmarks [12], we show that Corral reduces the makespan of a batch of jobs by 10-33% and the median job completion time by 30-56% compared to the Yarn capacity scheduler [7]. Further, using large-scale simulations over a 2000 machine topology, we show that Corral achieves better performance than flow-level techniques (such as those in Varys [24]) and that their gains are orthogonal.

In summary, our contributions are as follows.
(1) We quantify the high predictability of job characteristics in production clusters.
(2) We formulate the joint data and compute placement problem in big data clusters as a malleable job scheduling problem and solve it using efficient heuristics, which lead to significant benefits in practice.
(3) We design, implement and evaluate Corral, a data-driven planning framework which leverages the predictability of future workloads to assign resources to jobs, leading to significant performance improvements.
(4) We show that the benefits of Corral are more than those from just using network flow-level scheduling techniques, and that Corral can work together with such techniques to provide further performance benefits.

2 Motivation

Big data clusters use frameworks like Hadoop [3], Hive [44], or Scope [50] to run simple MapReduce [25] jobs, and more complex, DAG-structured jobs. Each job consists of *stages*, such as map or reduce, linked by data dependencies. Each stage consists of several *tasks*, which process the input data in parallel. When a task has all its inputs ready, it is allocated a *slot* on a machine in the cluster, which gives it a predefined amount of memory and CPU to execute. The input data for these jobs is stored in a distributed file system like HDFS [6]. The data is divided into multiple *chunks*, each of which is typically replicated three times across different machines in the cluster. For fault tolerance, two of the chunks reside on the same rack, while the third one is on a different rack. Each chunk is placed independently of the other chunks. In this section, we present observations about existing cluster frameworks that motivate the design of Corral.

Current schedulers do not jointly optimize over data and compute location. The location of data replicas in the distributed file system is typically determined by the file system itself, irrespective of the computation performed on it (as described above). Thus, the input chunks for a job are spread across the compute nodes uniformly at random.

At the job scheduling level, much effort has been devoted to avoiding and scheduling around network contention, given a fixed location of task input data. For example, Hadoop prefers running map tasks on machines with their input data, instead of loading the data over the network.

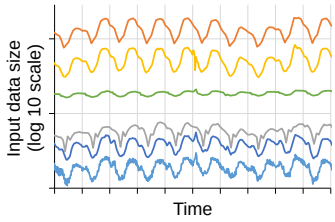


Figure 1: Normalized input data size of six different jobs during a ten-day period. X-axis shows time; each tick is a day. Y-axis shows input size in $\log_{10}$ scale; each tick is a 10x increase.

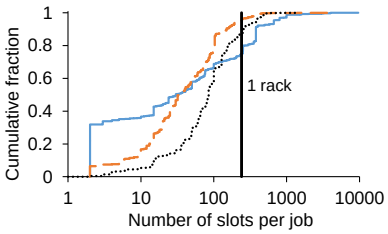


Figure 2: Cumulative fraction of the number of compute slots requested by jobs across three production clusters. The vertical line represents 240 slots, the size of one rack.

Quincy [36] aims at improving on this greedy heuristic by modeling the tradeoffs between latency and locality[1] preferences of all runnable tasks, and determines their placement using a holistic optimization framework. As the input data is randomly spread across several racks in the cluster, such approaches generally achieve locality for earlier stages of a job (e.g., map in MapReduce, Extract in Scope) but not for the subsequent stages (e.g., shuffle), which typically end up reading data across most of the racks. Given a fixed placement of both input data and tasks, systems such as Orchestra [23], Varys [24], and Baraat [26] schedule network flows to minimize makespan or average job duration. Improper placement of data and tasks can cause flows to utilize congested network links, limiting their potential benefits.

Job input data can be placed in designated locations before job execution. We identify two important scenarios where it is possible to place job input data. First, it is fairly typical in datacenters that data is continuously uploaded from front-end servers into the cluster (e.g., click logs), and data-processing jobs are triggered as new data becomes available [5]. As this data is being uploaded, it is possible to place it at the desired location.

Secondly, in some deployment scenarios (such as running Hadoop in Azure [9] or AWS [2]), the job input data is stored in a separate storage cluster (Azure Storage [10, 13] or Amazon S3 [1]). When a job starts, it processes data by fetching it from the remote storage cluster. By constraining tasks of the job to execute inside a particular subset of racks, we can ensure that the data is placed on those racks.

Job characteristics can often be accurately predicted. Cluster workloads are known to contain a significant number (up to 40%) of *recurring* jobs [15, 30]. A recurring job is one in which the same script runs whenever new data becomes available. Consequently, for every instance of that job, it has a fixed structure and similar characteristics (e.g., amount of data transferred in shuffle, or CPU and memory demands).

We confirm and quantify this intuition by examining twenty business-critical jobs from our production clusters (these jobs are part of the workload W3 described in Section 6). For each job, we compute the input data size of its instances during a recent one-month period. Figure 1 shows the normalized job sizes as a time series for six of those jobs over a 10-day period. Overall, these jobs have input sizes ranging from several gigabytes to tens of terabytes. To predict the input size of a job which is submitted at a particular time (e.g., 2PM), we average the input size of the same job type at the same time during several previous days. In particular, if the current day of the week is a weekday (weekend), we average only over weekday (weekend) instances. Using this, we can estimate the job input data size with a small error of 6.5% on average. We observe similar predictability of the intermediate (or shuffle) data and output data. In turn, this allows us to predict the amount of data transferred during job execution and utilization of network links.

It is possible to run a bulk of jobs within a few racks without losing parallelism. Figure 2 plots the requested number of slots for various jobs across three of our largest production clusters, each of them with more than 10,000 machines. While some jobs require up to 10,000 slots, we find that across these three clusters, 75%, 87%, and 95% of the jobs require less than one rack worth of compute resources (240 slots). Similar observations have been reported earlier [28]. Thus, a large number of jobs can each be run within a single rack without sacrificing their parallelism. This allows these jobs to communicate at full NIC speeds.

In summary, we can achieve better data locality by jointly optimizing the placement of data and compute in big data clusters. The ability to predict the future workload allows us to plan ahead, place job input data in specific racks and subsequently run its tasks in those racks. Thus, we can take advantage of running many small jobs in individual racks, freeing up more core bandwidth for large or ad hoc jobs.

3 Corral design principles

In this section, we describe the architecture of Corral (Section 3.1), design considerations for its offline planning component (Section 3.2), and our approach to formalizing and solving the planning problem (Section 3.3). We leave the details of the planning algorithm to Section 4.

3.1 System architecture

Corral consists of two components – (a) an *offline planner* and (b) a cluster scheduler (Figure 3). The offline planner receives estimates of characteristics of jobs (e.g., arrival time, input data size etc.) that will be submitted to the cluster in future (step 1). It uses these characteristics to estimate job latencies and solve an offline *planning problem* with the goal of minimizing a specified metric (e.g., makespan or average job completion time). The planner creates a schedule which

[1] We use the term locality to refer to the fact that a task has been scheduled on the machine or rack containing most of its input data.

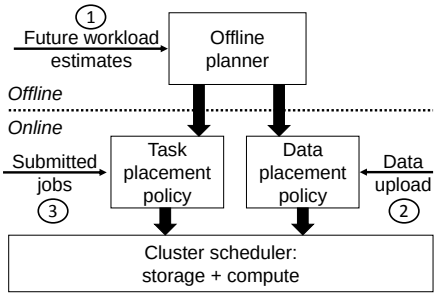


Figure 3: Corral **system architecture.**

consists of a tuple $\{R_j, p_j\}$ for each job j, where R_j is the set of racks on which job j has to run and p_j is its priority.

The cluster scheduler uses the schedule from the offline planner as guidelines to place data and tasks when running jobs on the cluster. As the input data of a job j is uploaded into the cluster (step 2) and stored in a distributed filesystem (such as HDFS), Corral places one replica of each data chunk in a randomly chosen rack from R_j. The remaining replicas are placed on another rack in the cluster, which is randomly chosen from the entire set of racks excluding the one chosen so far. We note that these choices are consistent with the *per-chunk* fault tolerance policy typically used in HDFS.

Finally, when job j is submitted to the cluster (step 3), the cluster scheduler constrains its tasks to be scheduled within the racks in R_j. Whenever a slot becomes empty in any rack, Corral's scheduler examines all jobs which have been assigned that rack and assigns the slot to the job with the highest priority[2]. With such placement of data and tasks in the same set of racks, Corral ensures that not only the initial stage (e.g., map, extract) but also subsequent stages of a job (e.g., reduce, join) achieve data locality. The priority (p_j) of a job ensures that the order in which jobs are scheduled by the cluster scheduler conforms to that assigned by the offline planner.

The offline planner will periodically receive updated estimates of future workload, rerun the planning problem, and update the guidelines to the cluster scheduler. Note that the cluster conditions might change after the guidelines are generated (e.g., racks/machines failing). If the assigned locations are not available for any job, the scheduler will ignore the guidelines for that job and use randomly selected machines in the cluster to hold its data (while ensuring fault tolerance using a policy similar to HDFS) and run its tasks.

The above scheduling approach in Corral applies not only to recurring jobs but also to jobs known a-priori and having predictable characteristics. For ad hoc jobs (whose characteristics are not known in advance), Corral places the input data using regular HDFS policies. The tasks of such jobs are scheduled using existing techniques [48].

3.2 Design considerations

Objective. The goal of the offline planner is to provide guidelines to the cluster scheduler for placing data and compute tasks. The objective is to optimize the network locality of each job and thus, improve the overall performance.

Scenarios. Based on the different cases that arise in practice, we consider two scenarios. In the *batch* scenario, we run a batch of jobs that are all submitted to the cluster at the same time (e.g., a collection of log analysis jobs). Here, our goal is to minimize their *makespan*, i.e., the time to finish the execution of all the jobs in the batch. In the *online* scenario, the jobs arrive over a longer period of time with known arrival times. In this case, our goal is to minimize the average job completion time.

Challenges. Designing the planner raises three important questions. First, at what granularity should the guidelines be provided? For example, one option is to provide a target machine for each task of the job. Second, how to formalize the planning problem to make it tractable at the scale of current big data systems? Finally, as the planner needs to choose among different data and task placement options, how do we estimate the job latency of a particular configuration?

Next, we outline how we address these challenges.

3.3 Solution approach

In order to formulate a tractable offline planning problem, we make several simplifying assumptions, highlighted below. We note that these assumptions apply only to the offline planner, and not to the actual job execution on the cluster.

Planning at the granularity of jobs and racks. The solution to the planning problem can be specified at different granularities. At one extreme, it can prescribe which task of a job should run on which machine in the cluster. However, generating such a solution is non-trivial. This is because while job input sizes can be predicted with small error, the input sizes of individual tasks depend on how the data is partitioned across tasks in a stage and thus, much less predictable. Further, the number of tasks can be several orders of magnitude more than the number of jobs, making the problem practically intractable.

Instead of planning at the level of tasks, one can plan at a stage level, i.e., specify which rack(s) each stage in a job should use. Complex, DAG-structured jobs could potentially benefit from stage-level planning; e.g., two parallel shuffles in a DAG could run in two separate racks, both benefiting from locality. However, after examining a large number of production jobs in our clusters, we found very few DAG jobs where such stage-level planning provides locality benefits.

Therefore, to improve scalability and robustness of the offline plan, we pose the planning problem at the granularity of racks and jobs. Further, most production clusters have full bisection bandwidth within a rack and oversubscribed links from the racks to the core [22, 23, 24]. Planning at the granularity of racks, allows us to assume that all tasks in a rack can communicate at NIC speeds, which in turn simplifies the modeling of job latency.

Planning as malleable job scheduling. We formulate the planning problem as a malleable job scheduling problem [19, 27, 45]. A *malleable* job is a job whose latency depends on the amount of resources allocated to it. While each

[2]To ensure machine-level locality, we use the same techniques as in delay scheduling [48].

job typically has a fixed requirement for the total number of compute slots, the planner's decision on over how many racks the job will execute would affect its latency.

To illustrate the dependency of job latency on the number of racks it uses, consider the shuffle stage in a MapReduce job with a total of S bytes spread over r racks (latency models for an entire job are described in Section 4.3). Suppose each rack has an internal bandwidth of B, and racks are connected with oversubscription ratio V; assume that network is the bottleneck resource affecting latency. When $r = 1$, almost all of S bytes have to be transferred across machines in the rack, with total latency of S/B. For $r > 1$, assuming symmetry for simplicity, each rack sends $(r-1)/r$ fraction of its data to other racks. Hence, the whole shuffle has to transfer $S(r-1)/r$ bytes using aggregate bandwidth of rB/V (from rack to core), resulting in approximate latency of $\frac{(r-1)SV}{r^2B}$ which approaches $\frac{V}{r}\frac{S}{B}$ for large r. Thus, the shuffle latency reduces with r in this example.

Characterizing job latency using simple response functions. With the above intuition, we model job latency using latency response functions $L_j(r)$, where j is the job index and r is the number of racks allocated to j. In particular, we model job latency as depending on the number of racks allocated, and also on fixed job characteristics such as the amount of data transferred between tasks, amount of CPU resources required, and the maximum parallelism of the job. Using these job characteristics, which can be estimated from earlier runs of the job, we derive simple analytical models to compute job latency (Section 4.3).

The actual job latencies can depend on additional runtime factors such as failures, outliers, and other jobs which run simultaneously on the same rack(s). However, as our latency approximation is used only for offline planning (and these factors affect all jobs), we tradeoff accurate (absolute) latency values for simpler and practical planning algorithms. As seen in our evaluation, using approximate latency models suffices for significant latency improvements.

These latency response functions also help us abstract various factors that affect job latency (e.g., number of maps or reducers) and enable us to plan for jobs to minimize the required metric. Such abstraction allows us to account for other factors such as data imbalance across racks (Section 4.5) without modifying our core algorithms.

4 Offline planning in Corral

In this section, we first formally present Corral's planning problem (Section 4.1). We then describe our heuristics to solve it (Section 4.2). Next, we provide details on how to calculate the functions, which are used to estimate job latency (Section 4.3). We then discuss the complexity of our heuristics (Section 4.4), and conclude by describing how we account for potential input data imbalance across racks.

4.1 Problem formulation

Given the design choices in Section 3, Corral's offline planning problem is formulated as follows. A set of jobs $\mathcal{J}$ (of size J) has to be scheduled on a cluster of R racks. Each job j is characterized by a *latency response function* $L_j : [1, R] \rightarrow \Re^+$, which gives the (expected) job completion time as a function of the number of racks assigned to the job. Our model assumes that once a subset of racks is allocated to a job, it is used by the job until completion. That is, we do not allow preemption or a change in the allocation throughout the execution of the job. This assumption simplifies our problem formulation significantly. To ensure work conservation during job execution, Corral's cluster scheduler does not enforce these constraints. Our evaluation results (Section 6) show that even with this deviation from the assumptions, Corral significantly outperforms existing schedulers. In the batch scenario, the goal is to minimize the makespan, *i.e.*, the time it takes to complete all jobs. In the online scenario, each job has its own arrival time. The goal now is to minimize the average completion time, *i.e.*, the average time from the arrival of a job until its completion. Both problems are NP-hard [27, 45]. Hence, we design efficient heuristics that run in low polynomial time.

4.2 Planning heuristics

Solving the planning problem consists of determining (a) the amount of resources (number of racks) to be allocated to a job, and (b) where in the cluster these resources have to be allocated. To address each of these sub-problems and use ideas from existing techniques (e.g., LIST scheduling [31]), we decouple the planning problem into two phases – the *provisioning* phase and the *prioritization* phase. In the provisioning phase, for each job j, we determine r_j, the number of racks allocated to the job. In the prioritization phase, given r_j for all jobs, we determine R_j, the specific subset of racks for j, and T_j, the time when j would start execution. We use T_j to determine the priority ordering of jobs (Section 3.1).

Provisioning phase. Initially, we set $r_j = 1$ for each job. This represents a schedule where each job runs on a single rack. In each iteration of this phase, we find the job j which is allocated less than R racks and has the longest execution time (according to the current r_j allocations), and increase its allocation by one rack. When a job is already allocated R racks it cannot receive additional racks. We proceed iteratively until all jobs reach $r_j = R$.

Intuitively, by spreading the longest job across more racks, we shorten the job that is "sticking out" the most. Note that if the latency of the longest job increases when its allocation is increased by one rack, it will continue to be the longest and thus, its allocation will be increased again in the next iteration. For the latency response curves we observed it practice, we found that the latency of the longest job eventually decreases.

As each job in $\mathcal{J}$ can be allocated any number of racks between 1 and R, a total of R^J different allocations exist for the provisioning phase. The above heuristic is designed to explore a plausible polynomial-size subset of candidate allocations ($J \cdot R$), which can be evaluated within practical time constraints. For each such allocation, we run the prioritization phase described below and pick the allocation (and respective schedule) which yields the lowest value for

Input: Set of jobs $\mathcal{J}$ of size J; $\forall j \in \mathcal{J}$, r_j, the number of racks to be assigned to job j for $L_j(r_j)$ time units and A_j, the arrival time of job j (0 in batch case).
Output: $\forall j \in \mathcal{J}$, R_j, the set of racks allocated to job j and T_j, the start time of job j.
Initialization: *Sort* and re-index jobs according to the scenario (batch or online).
$j := 0$
$F_i := 0$ for all racks $i = 1 \ldots R$
while $j < J$ **do**
  $R_j :=$set of r_j racks with smallest F_i
  $T_j := \max\{\max_{i \in R_j} F_i, A_j\}$
  for $i \in R_j$, $F_i := T_j + L_j(r_j)$
  $j := j + 1$
end while

Figure 4: Prioritization phase.

the relevant objective (makespan for batch scenario, average completion time for online scenario).

We note that this heuristic is similar to the one used in [19] for scheduling malleable jobs with the objective of makespan minimization. [19] terminates the heuristic when $\sum_{j|r_j>1} r_j = R$, and obtains an approximation ratio of roughly 2 on the makespan, by using LIST scheduling [31] on top of the provisioning heuristic. However, as each iteration is fast, we allow ourselves to run the heuristic for more iterations compared to [19], until reaching $r_j = R$ for every job j. This allows us to explore more options, and to obtain adequate results (including for the objective of average completion time, for which [19] does not provide guarantees).

Prioritization phase. If all jobs are constrained to run on a single rack, the longest processing time first (LPT) algorithm [31] is a well-known scheduling algorithm one could use to minimize makespan. However, jobs can run on multiple racks in Corral. We extend LPT to account for this case (pseudo-code in Figure 4).

In the batch scenario, we first sort jobs in decreasing order of number of racks allocated to them (r_j), i.e., *widest-job first*. Then, to break ties, we sort them in decreasing order of their processing times (similar to LPT). The widest-job first order helps us avoid "holes" in the schedule – for example, a job allocated R racks will not have to wait for a job allocated just one rack to complete, which would result in wasted resources. We then iterate over the jobs in this sorted order, assigning the required number of racks. We keep track of F_i, the time when rack i finishes running previously scheduled jobs. For each job j, we allocate the required set of racks by selecting the first r_j racks that are available (based on F_i). We then update F_i for the selected racks based on the start time of the job and its duration.

For the online scenario, we sort jobs in ascending order of their arrival time; in case of ties, we apply the sorting criteria of the batch case in the same order described above, and then use the algorithm described in Figure 4.

Estimating the quality of our heuristics. Decomposing the planning problem into two phases, as above, can be suboptimal. However, it turns out that our approach results in near-optimal performance. Our simulations (using workloads described in Section 6) show that the above heuristic for the batch (online) scenario finds a schedule with resulting makespan (average completion time) within 3% (15%) of the solution of an LP-relaxation to the planning problem. The LP-relaxation, which is described in the appendix, serves as a lower-bound to any solution (under the assumptions described in Section 4.1)

4.3 Latency response functions

We now demonstrate how to model the latency response functions for data-parallel jobs, which allow us to approximate the expected latency of a job running on a given number of racks. We borrow the basic modeling principles from Bazaar [37], and adapt them to our setting where racks are the basic allocation units. We would like to emphasize that the models we present here should be viewed as *proxies* for the actual latencies, and need not be highly accurate. More complex models exist in the literature (e.g., Starfish [34], MRPerf [46]), but our goal in designing the response functions is to be able to account for job latency using fast models and simple mathematical expressions.

In what follows, we first present the basic model for MapReduce jobs, and then show how to extend the MapReduce model to the case of general DAGs.

MapReduce model. We can represent any MapReduce job j by the 5-tuple $\langle D_j^I, D_j^S, D_j^O, N_j^M, N_j^R \rangle$, where D_j^I is the input data size, D_j^S is the shuffle data size, D_j^O is the output data size, N_j^M is the number of map tasks and N_j^R is the number of reduce tasks used by the job.

We model the execution of a MapReduce job as consisting of three stages: (a) the map stage, (b) the shuffle stage and (c) the reduce stage. For simplicity we assume here that these stages run sequentially (we note, however, that it is possible to extend the model to account for cases where the map and reduce stage run in parallel). Accordingly, the latency $L_j(r)$ of job j using r racks can be modeled as the sum of latencies of the three stages, i.e., $L_j(r) = l_j^{\text{map}}(r) + l_j^{\text{shuffle}}(r) + l_j^{\text{reduce}}(r)$.

Given r racks and k machines per rack, the map stage runs in $w^{\text{map}}(r) = \lceil \frac{N_j^M}{r \cdot k} \rceil$ waves. Using this, the latency of the map stage is given by $l_j^{\text{map}}(r) = w^{\text{map}}(r) \cdot \frac{D_j^I / N_j^M}{B_M}$, where B_M is the average rate at which a map task processes data (this quantity is estimated from previous runs of the same job). Similarly, the latency of the reduce stage is estimated as $l_j^{\text{reduce}}(r) = w^{\text{reduce}}(r) \cdot \frac{D_r^O / N_j^R}{B_R}$, where B_R is the average data processing rate of a reduce task (estimated from previous runs) and $w^{\text{reduce}}(r) = \lceil \frac{N_j^R}{r \cdot k} \rceil$.

The latency of the shuffle stage is determined by the maximum of two components:
(a) Time to transfer data across the core: This is calculated as $l_j^{\text{core}}(r) = \frac{D_j^{\text{core}}(r)}{B/V}$ where B is the bandwidth per machine, V (> 1) is the oversubscription ratio between the racks and the core, and D_j^{core} is the amount of shuffle data

transferred across the core from a single machine. It is given by $D_j^{\text{core}}(r) = \frac{D_j^S}{r \cdot k} \cdot \frac{r-1}{r}$, if $r > 1$. If $r = 1$, i.e., the job is allocated only one rack, $D_j^{\text{core}}(r) = 0$.

(b) Time to transfer data within a rack: Along with the data transferred across the core, each machine transfers data $D_j^{\text{local}}(r)$ within the rack which is given by $\frac{D_j^S}{r \cdot k} \cdot \frac{1}{r}$. While $\frac{1}{k}^{\text{th}}$ of this data remains on the same machine, the remaining data is transferred to other machines using a bandwidth of $B - B/V$. Thus, the time for transferring data within the rack is given by $l_j^{\text{local}}(r) = \frac{D_j^{\text{local}}(r)}{B - B/V} \cdot \frac{k-1}{k}$.

Thus, the shuffle latency $l_j^{\text{shuffle}}(r)$ is given by $w_{\text{reduce}}(r) \cdot \max\{l_j^{\text{core}}(r), l_j^{\text{local}}(r)\}$, where $w_{\text{reduce}}(r)$ is the number of reduce waves, as defined above.

For ease of presentation, the above model assumes that each machine runs only one task at a time. It can be easily extended to the case where multiple tasks run on a machine by modifying (i) the number of waves of the map and reduce stage, and (ii) the amount of bandwidth available per task during the shuffle stage.

General DAGs. To construct a latency response function for data-parallel DAGs (generated by frameworks such as Hive [44] and Tez [4]), we first model every stage of the DAG as a MapReduce job. Using this abstraction and the MapReduce model developed above, we then determine the latency response function $L_s(r)$ for every stage s in the DAG. Finally, the latency of the DAG is set as the latency of its *critical path* P (i.e., a path from a source stage to a sink stage in the DAG, which takes the longest time to run), namely $L_j(r) = \sum_{s \in P} L_s(r)$.

Assumptions. The above latency models make several assumptions. First, latency of each stage is assumed to be proportional to the amount of data processed by it and similar across all tasks within a particular stage. While this is valid for a wide variety of jobs [37], it may not hold for cases where the computation latency depends on the value of the data read or in the presence of significant data skew across tasks [39]. Second, the resource demands of the map and reduce stages are assumed to be similar to previous runs of the job (on different datasets). We observed this assumption to hold for a variety of workloads we used in our evaluation (Section 6), and has also been shown to hold elsewhere [14, 30]. Finally, we assume that the data (both input and intermediate) and tasks of the various stages in a job are spread uniformly across all the machines in the racks allocated to the job.

While deviations from the above assumptions can lead to errors in predicting the latency of a job, our results (Section 6) show that Corral is robust to such errors.

4.4 Complexity

The complexity of the prioritization phase is $O(JR)$, because for each job we make a pass over all R racks to determine the first r_j racks that become available. The provisioning phase has JR iterations. Hence, the overall complexity of our heuristic is $O(J^2R^2)$. The complexity of calculating the latency response functions is linear in R and thus does not increase the overall complexity[3].

Figure 5: Running time of the offline planner heuristic in Corral for a 4000 machine cluster with varying number of jobs.

In terms of actual running time, our heuristic is highly scalable as shown in Figure 5. Running our heuristic on a single desktop machine with 6 cores and 24GB RAM, we found that it requires around 55 seconds to generate the schedule for 500 jobs on a 4000 node cluster with 100 racks (40 machines per rack). As the planner runs offline, this results in minimal overhead to the cluster scheduler.

4.5 Accounting for data (im)balance.

Using the latency response functions as described above, Corral would directly optimize for latency-related metrics, but would ignore how the input data is spread across different racks. Consequently, it is possible that a large fraction of the input data would be placed in a single rack, leading to high data imbalance and excess delays (e.g., in reading the input data). To address this issue and achieve better data distribution, we add a penalty term to the latency response function, given by $\alpha \cdot D_j^I/r$, where D_j^I/r is the amount of input data of job j in a single rack and α is a tradeoff coefficient. Accordingly, the modified latency response function is given by $L'_j(r) = L_j(r) + \alpha \cdot D_j^I/r$, where $L_j(r)$ is the original response function as described in Section 4.3.

In our experiments (Section 6), we set α to be the inverse of the bandwidth between an individual rack and the core network. The intuition here is to have the penalty term serve as a proxy for the time taken to upload the input data of a job to a rack. Increasing α favors schedules with better data balance. We note that in practice, we supplement this approach by greedily placing the last two data replicas on the least loaded rack. The combination of these approaches leads to a fairly balanced input data distribution (Section 6.2).

5 Implementation

We implemented Corral on top of the Apache Yarn framework (Hadoop 2.4) [8] and HDFS. Corral's offline planner determines the set of racks where (a) the input data of a job has to be stored and (b) its tasks have to be executed. To ensure that these rack preferences are respected (as described in Section 3.1), we made the following changes to the dif-

[3]With DAGs, we find critical path via an efficient shortest path algorithm; using BFS, this adds an $O(V+E)$ to the complexity, where V is the number of stages and E is the number of edges in the DAG. However, because the DAGs are typically small compared to number of racks and jobs, this does not change the complexity of our heuristic.

	50%-tile	95%-tile
Number of tasks	180	2,060
Input Data Size (GB)	7.1	162.3
Intermediate data size (GB)	6	71.5

Table 1: Characteristics of workload W3 from Microsoft Cosmos [20]

ferent components in Yarn. The changes involve about 300 lines of Java code.

Data placement policy. We modified HDFS's `create()` API to include a set of ⟨rack, number of replicas⟩ tuples, which allows Corral to specify the racks where different replicas of a file's data chunk are to be placed. These specifications are passed on to the block placement policy in HDFS, which was modified to ensure that at least one replica of the data chunk is placed on a machine which belongs to the rack specified.

Task placement policy. Every job in Yarn uses an Application Manager (AM) to request slots from a centralized Resource Manager (RM). The AM can also specify preferences for locations (e.g., specific machine or rack) where it would like slots to be allocated in the cluster. By default, the MapReduce AM in Yarn specifies location preferences for map tasks only and not for the reducers. For Corral, we modified the Yarn MapReduce AM to specify locality preferences for *all* tasks of a job.

As the subset of racks where a job needs to be scheduled is determined by Corral's offline planner, we pass this to the AM using a new configuration parameter. This is further passed to the RM as a location preference. The RM makes every effort to respect this locality preference while allocating slots to the job. However, in the event that a majority of the machines in the racks preferred by a job are unreachable, the RM will ignore the locality guidelines and allocate slots on the available nodes in the cluster.

6 Evaluation

We evaluate Corral on a 210 machine cluster, using a variety of workloads drawn from production traces. Our main results are as follows.

(a) Compared to Yarn's capacity scheduler, Corral achieves 10-33% reduction in makespan and 26-36% reduction in average job completion time for workloads consisting of MapReduce jobs (Section 6.2). For Hive queries derived from the TPC-H benchmark [12], Corral improves completion times by 21% on average (Section 6.3).

(b) When a workload consists of both recurring and ad hoc jobs, using Corral to schedule the recurring jobs improves the completion times of the recurring and ad hoc jobs by 33% and 20% (respectively), on average (Section 6.4).

(c) Corral's improvements increase by more than 2X when network utilization goes up by 33%, and are robust to errors in predicted job characteristics (Section 6.5).

(d) Using large-scale simulations, we show that the benefits from flow-level schedulers like Varys [24] improve significantly when used in combination with Corral and that Corral's benefits are orthogonal to those of using Varys alone.

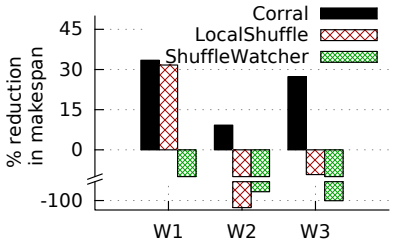


Figure 6: Reduction in makespan for different workloads, compared to Yarn-CS in the batch scenario.

6.1 Methodology

Cluster setup. We deployed our implementation of Corral in Yarn/HDFS on a 210 machine cluster, organized into 7 racks with 30 machines per rack. Each machine has 32 cores, 10Gbps NICs, and runs CentOS 6.4. The racks are connected in a folded CLOS topology at 5:1 oversubscription, i.e., each rack has a 60Gbps connection to the core. To match network conditions in production clusters, we emulate background traffic, accounting for up to 50% of the core bandwidth usage [5, 22].

Workloads. We first evaluate Corral using the scenario where all jobs in the workload are assumed to be recurring (or have predictable characteristics). We then consider workloads with both recurring and ad hoc jobs. We use jobs from the following workloads for our evaluation.

(a) W1: Starting from the Quantcast workloads [42], we constructed this workload to incorporate a wider range of job types, by varying the job size, and task selectivities (i.e., input to output size ratio). The job size is chosen from small ($\leq$ 50 tasks), medium ($\leq$ 500 tasks) and large ($\geq$ 1000 tasks). The selectivities are chosen between 4:1 and 1:4.

(b) W2: This workload is derived from the SWIM Yahoo workloads [21] and consists of 400 jobs.

(c) W3: We have chosen 200 jobs, randomly, from a 24 hour trace collected from production big data clusters at Microsoft and constructed this workload. Some characteristics of this workload are given in Table 1.

(d) TPC-H: We run queries from the TPC-H benchmark using Hive [44] to evaluate the performance of Corral for general DAG-structured workloads.

Baselines. We compare Corral against three baselines.

(a) the capacity scheduler in Yarn [7] (referred to as Yarn-CS from now on). The capacity scheduler uses techniques like delay scheduling [48] to achieve locality for map tasks but does not plan for data placement. Comparison with Yarn-CS allows us to show the benefits of achieving better locality for all stages of a job using Corral.

(b) ShuffleWatcher [16], which schedules each job in a subset of racks to reduce the amount of cross-rack data transferred by them. ShuffleWatcher does not place the input data of the job in these racks and as a result, most maps end up reading their input across the core network. It also fails to account for contention between jobs and schedules them independently from each other. Comparison with Shuffle-

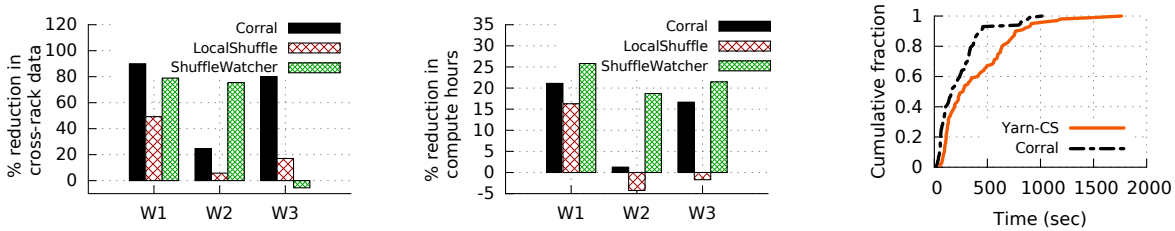


Figure 7: Comparing (a) reduction in cross-rack data transferred and (b) compute hours relative to Yarn-CS, and (c) cumulative fraction of average reduce time, for workload W1 in the batch scenario.

Watcher allows us to show the benefits of careful planning, and joint data and compute placement in Corral.
(c) LocalShuffle, which uses the task placement of Corral but the data placement policy of HDFS. The comparison of Corral with LocalShuffle allows us to quantify the benefits of proper placement of input data. We note that, unlike ShuffleWatcher, LocalShuffle schedules jobs using the same offline planning phase as Corral.

For all the above baselines, the input data is placed using HDFS's default (random) placement. When using Corral, we run its offline planner taking data balance into account (Section 4.5). Using the generated schedule, the input data of the jobs is placed on the assigned racks while it is uploaded and jobs are run using Corral's task placement policy (as described in Section 3.1).

Metrics. The primary metrics of interest are (a) makespan, in the batch scenario, and (b) average job completion time, in the online scenario.

6.2 MapReduce workloads

In this section, we show the benefits of using Corral to schedule MapReduce jobs in the batch and online scenarios, when all jobs have predictable characteristics.

6.2.1 Batch scenario

Figure 6 shows the improvement in makespan relative to Yarn-CS, for different workloads, when run as a batch – Corral achieves a 10% to 33% reduction. The reduction in makespan for W2 is lower than that for the other workloads because W2 is highly skewed. Almost 90% of the jobs are *tiny* with less than 200MB (75MB) of input (shuffle) data and two (out of the 400) jobs are relatively *large*, reading nearly 5.5TB each. These large jobs determine the makespan of W2 and do not suffer significant contention from the tiny jobs. Out of the 7 racks available, Corral allocates 3 racks each to the two large jobs and packs most of the tiny jobs on the remaining rack. Compared to Yarn-CS, the benefits of Corral stem from running each of the large jobs in isolation, on separate subsets of racks.

Corral's improvements are a consequence of its better locality and reduced contention on the core network. Figure 7a shows that Corral reduces the amount of cross-rack data transferred by 20-90% compared to Yarn-CS. This, in turn, improves task completion times. To quantify this, we use two additional metrics, namely, (a) compute hours, which measures the total time spent by all the tasks in the workload; compared to Yarn-CS, using Corral reduces the compute hours by up to 20% (Figure 7b), and (b) average reduce time, which measures the average execution time of all the reduce tasks in a job. Figure 7c plots the cumulative fraction (over jobs) of this metric for Corral and Yarn-CS, showing that Corral is approximately 40% better at the median, with higher benefits at the tail.

Comparison with other baselines. Corral outperforms LocalShuffle showing that proper input data placement is key for good performance (Figure 6). Even with better shuffle locality, LocalShuffle performs worse than Yarn-CS for W2 and W3 due to its lack of input data locality.

ShuffleWatcher optimizes for each job individually and ends up scheduling several large jobs on the same subset of racks. This leads to increased completion times for all those jobs. In the worst case, it can schedule all jobs on a single rack as it doesn't directly optimize for makespan or job completion time but tries to minimize the cross-rack data transferred. Thus, ShuffleWatcher results in significantly worse makespan compared to Yarn-CS for all workloads (Figure 6). Note that using ShuffleWatcher results in lesser cross rack data for W2 compared to Corral (Figure 7a). This is because the large jobs in W2 have nearly 1.8 times more shuffle data than input data, and Corral spreads them on 3 racks each (as mentioned above, for better makespan) while ShuffleWatcher places them in a single rack.

ShuffleWatcher achieves better compute hours than Corral (Figure 7b) because it loads racks unevenly with some racks running significantly lower number of jobs than others. The tasks of these jobs finish faster, resulting in lower compute hours. However, as shown in Figure 6, ShuffleWatcher is significantly worse than Corral for makespan.

Data balance. Corral optimizes for reducing the imbalance in the data allocated to different racks (Section 4.5). To evaluate this, we measure the coefficient of variation (CoV) of the size of input data stored on each rack. Our results show that Corral has a low CoV of at most 0.004 and performs better than HDFS, which spreads data randomly, resulting in a CoV of at most 0.01[4].

[4]Note that the CoV of a random distribution can be higher than that of uniform distribution, which is 0.

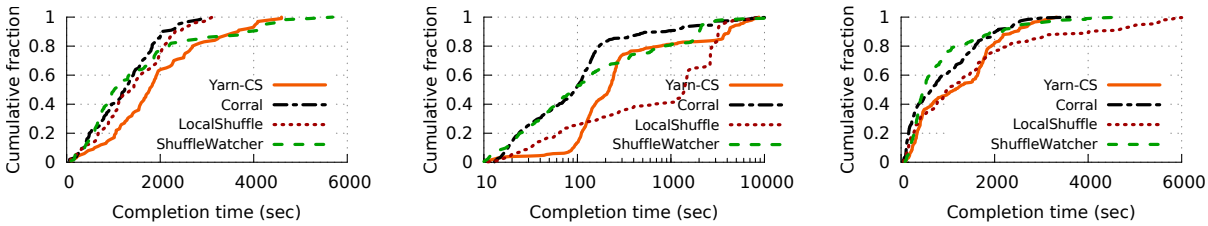


Figure 8: Cumulative fraction of job completion times for different workloads, when jobs arrive online.

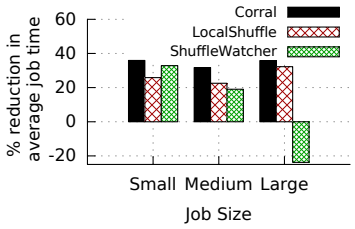


Figure 9: Reduction in average job completion time relative to Yarn-CS, for workload W1 in the online scenario.

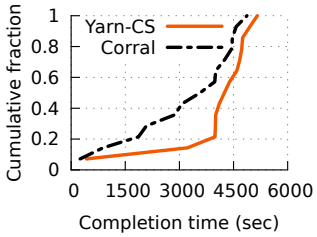


Figure 10: Benefits of running TPC-H queries with Corral.

6.2.2 *Online scenario*

In this scenario, jobs arrive over a period of time instead of as a batch. We pick the arrival times uniformly at random in $[0, 60\text{min}]$. Figure 8 shows the cumulative fraction of job completion times for workloads W1, W2 and W3. Corral outperforms Yarn-CS, with 30%-56% improvement at the median and nearly 26-36% improvement for the average job time (not shown). Further, Corral equally benefits jobs of all sizes. Figure 9 shows the reduction in average job completion time for workload W1, binned by the job size. Corral achieves 30-36% reduction in average job time across the various bins.

Comparison with other baselines. Similar to the batch case, LocalShuffle performs worse than Corral due to the lack of proper input data placement (Figure 8). While ShuffleWatcher is close to Corral at the lower percentiles, it is significantly worse at the higher percentiles. ShuffleWatcher schedules jobs independently. It ends up placing a large fraction of jobs on a few racks and a smaller fraction of jobs on the remaining racks. Jobs on the lightly loaded racks run faster due to lesser contention and those on the heavily loaded racks slow down. Figure 9 further confirms this, as ShuffleWatcher reduces the completion times of the small/medium jobs relative to Yarn-CS but performs worse for large jobs.

6.3 DAG workloads

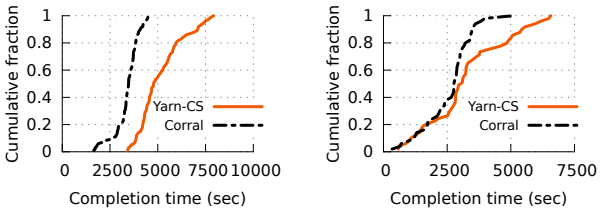
To evaluate the benefits of Corral for data-parallel DAGs, we ran 15 queries from the TPC-H benchmark [12], using Hive 0.14.0 [44]. Each query reads from a 200GB database organized in ORC format [11]. These queries are assumed to be recurring, and thus, can be scheduled using Corral. The queries are submitted over a period of 25 minutes, with arrival times chosen uniformly at random. To emulate conditions in a real cluster, along with the queries, we also submit a batch of MapReduce jobs chosen from the workload W1, which are run using Yarn-CS.

Figure 10 plots the cumulative fraction of the query completion times for two cases: (i) the queries are run using Corral (dashed black line) and (ii) the queries are scheduled using Yarn-CS (solid orange line). We see that Corral reduces the median execution time by nearly 18.5% with the average time being reduced by 21%. We found that these queries spend only up to 20% of their time in the shuffle stage, which shows that Corral can also provide benefits for workloads which are mostly CPU or disk bound.

6.4 Scheduling ad hoc jobs

A significant portion of jobs in a production cluster can be ad hoc, e.g., those run for research or testing purposes. Such jobs arrive at arbitrary times and cannot be planned for, in advance. Corral uses the same scheduling policies as Yarn's capacity scheduler (Yarn-CS) for such jobs. To explore the benefits of Corral in this scenario, we run a mix of 50 ad hoc and 100 recurring MapReduce jobs, drawn from W1. The ad hoc jobs are run as a batch with Yarn-CS, while the recurring jobs arrive uniformly over $[0, 60\text{min}]$.

Our observations are two-fold. First, even in the presence of ad hoc jobs, using Corral to schedule the recurring jobs is beneficial. Figure 11a shows the cumulative fraction of completion time for recurring jobs in the workload. Corral reduces the average (median) completion times by 33% (27%). Second, using Corral to schedule the recurring jobs leads to faster execution of ad hoc jobs, especially at the tail with a 37% reduction at the 90^{th} percentile compared to using Yarn-CS (Figure 11b). The makespan of the ad hoc jobs reduces by around 28% (not shown). As jobs run with Corral use significantly lower core bandwidth and complete earlier, more network and compute resources are available for the ad



(a) Recurring jobs **(b) Ad hoc jobs**

Figure 11: Using Corral with a mix of jobs.

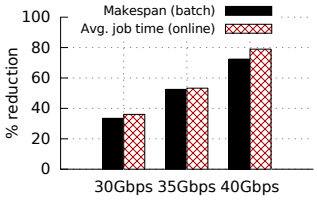


Figure 12: Benefits of using Corral relative to Yarn-CS, as background traffic increases for workload W1.

hoc jobs, allowing them to also finish faster.

While it is possible to use (a) techniques such as profiling (e.g., [33]) to estimate the latency of ad hoc jobs, or (b) simple scheduling decisions such as running the next ad hoc job on the least loaded rack, we leave exploration of such *adaptive* techniques to future work.

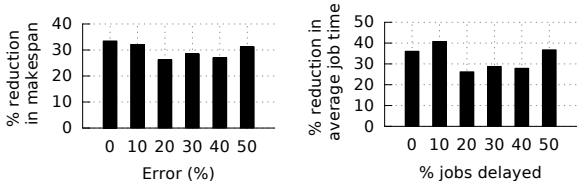
6.5 Sensitivity analysis

The benefits of Corral depend on (a) the load on the network and (b) the accuracy with which job characteristics can be predicted. Here, we evaluate Corral's robustness to variation in these factors.

Varying network load. Our results indicate that the gains of Corral increase significantly as the network utilization increases. For workload W1, Figure 12 shows that as the per-rack core network usage of background traffic increases from 30Gbps (50%) to 40Gbps (67%), Corral achieves more than 2X higher benefits compared to Yarn-CS in makespan (batch scenario) and average job time (online scenario).

Error in predicted job input data size. Compared to the 6.5% observed in practice (Section 2), we varied the amount of data processed by jobs up to 50% and found that the benefits of Corral relative to Yarn-CS remain between 25-35% (Figure 13a). This shows that Corral's schedule is robust to errors in job sizes seen in practice.

Error in job start times. In practice, the start of a job can vary due to various reasons such as (a) input data upload does not finish in time, or (b) the jobs on which it depends on are delayed. To evaluate the effect of such error in job start times, we choose a fraction f of jobs in a workload and add a random delay between $[-t, t]$, for a fixed t, in their start times. Figure 13b shows the results of running the online scenario for workload W1 with such perturbation in arrival times. We set t at 4 minutes, which is nearly 6.67 times the expected job inter-arrival time and 20% of the average job completion time (and thus, represents a significant error).



(a) Error in data size **(b) Error in arrival times**

Figure 13: Variation in benefits of Corral (relative to Yarn-CS) with error in job characteristics for workload W1.

Varying f from 0% to 50%, we found that the benefits of Corral reduce from 40% to at most 25%.

6.6 Using Corral with flow-level schedulers

Corral schedules the tasks in a job with better locality but does not explicitly schedule the network flows between them. In all experiments above, we use TCP as the transport protocol for Corral. However, several flow-level schedulers proposed in literature (e.g., [24, 26, 35]) have been shown to outperform TCP in datacenter environments. Here, we evaluate how Corral performs when used with such schedulers.

As our cluster does not support such schedulers, we built a flow-based event simulator for this purpose. The simulator also allows us to evaluate Corral on larger topologies. We use pluggable policies for the job and network schedulers. We have implemented Yarn-CS and Corral to represent job schedulers. For network schedulers, we implemented a max-min fair bandwidth allocation mechanism to emulate TCP, and Varys [24], which uses application communication patterns to better schedule flows.

We simulate a topology of 2000 machines, organized into 50 racks with 40 machines each, connected using the folded CLOS topology with 5:1 oversubscription. Each machine can run up to 20 tasks and has a 1Gbps NIC. We run 200 jobs from W1, arriving uniformly over 15min.

Figure 14 shows the cumulative fraction of the job completion times, when run using all the 4 possible combinations of job and network schedulers. Our main observations are as follows. First, using Varys with Yarn-CS improves the median job completion time by 46% compared to Yarn-CS+TCP. This is consistent with the improvements claimed previously [24]. Second, Corral+TCP outperforms Yarn-CS+Varys across all jobs, with nearly 45% gains at the median. This shows that the benefits of using schedulers like Varys are limited if flow end-points are not placed properly. On the other hand, Corral schedules jobs in their own set of racks, reducing their core bandwidth usage and improving completion times.

Finally, Corral+Varys results in much better job completion times compared to Corral+TCP or Yarn-CS+Varys. Thus, Corral's benefits are orthogonal to those attained with better flow-level scheduling and combining them performs better than using either one of them. Also, Corral is agnostic to the flow-level scheduler used and can potentially perform even better if its decisions take underlying flow-level schedule into account. Such cross-layer optimizations are part of our ongoing work.

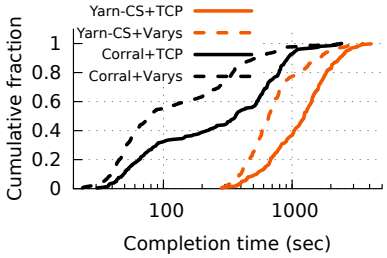


Figure 14: Simulation results: Cumulative fraction of job completion times with different flow-level and job schedulers.

7 Discussion

Dealing with failures. While Corral places one copy of job input data in the racks assigned to it, it spreads the other two copies across the rest of the cluster (similar to HDFS). This ensures that even if the assigned racks fail, the dataset can be recovered. Further, in the event that a majority of the machines (above a threshold) in a rack fail or a whole rack fails, Corral reverts back to using the existing placement policies (e.g., [48]) to run the jobs assigned to that rack, ensuring that they are not slowed down due to insufficient resources.

Data-job dependencies. Corral assumes that each job reads its own dataset. This simplifies the offline planning problem and allows each job to be scheduled independently. However, in general, the relation between datasets and jobs can be a complex bipartite graph. This can be incorporated into Corral by using the schedule of the offline planner and formulating a simple LP with variables representing what fraction of each dataset is allocated to each rack and the cost function capturing the amount of cross-rack data transferred, for any partition of datasets across the racks.

In-memory systems. Systems such as Spark [49] try to use memory to store job input data. While this decreases the dependence on disk I/O, jobs can still be bottlenecked on the network. The benefits of Corral extend to such scenarios as it reduces dependence on the core network and contention across jobs, by scheduling each of them on only a few racks.

Remote storage. As discussed earlier, in a public cloud setting (e.g., Amazon AWS [2]), the storage is separate from the compute cluster. This gives rise to the opportunity to schedule data transfers on the interconnect. One goal here would be to fully utilize both the interconnect and the compute cluster without having to be blocked on either resource. Extending Corral to this scenario is part of our ongoing work.

8 Related work

The techniques in Corral are related to the following areas of research in the context of datacenter applications.

Scheduling techniques for data analytics systems. Improving data-locality in big data clusters has been the focus of several recent works. Techniques like delay scheduling [48] and Quincy [36] try to improve the locality of individual tasks (e.g., maps) by scheduling them close to their input. ShuffleWatcher [16] tries to improve the locality of the shuffle by scheduling both maps and reducers on the same set of racks. Others such as Tetris [32] schedule recurring jobs to ensure better packing of tasks at machines. Corral, on the other hand, couples the placement of data and compute, achieving improved locality for all stages of a job.

Data placement techniques. CoHadoop [29] aims to colocate different datasets processed by a job on the same set of nodes, but does not guarantee locality for subsequent stages (e.g., shuffle). PACMan [18] caches repeatedly accessed data in memory but does not provide locality for intermediate data transfers. Techniques like Scarlett [17] use application access patterns to determine data replication factor or replica placement. None of these techniques coordinate data and compute placement for datacenter applications, which is the main focus in Corral.

Cross-layer scheduling techniques. The idea of placing data and compute together has been explored in systems like CAM [41] and Purlieus [43]. Unlike such systems, Corral exploits the recurring nature of datacenter applications and carefully assigns resources to execute them efficiently. Further, Corral deals with complex DAG-structured jobs which have not been considered previously.

Flow-level scheduling techniques. Several network-level techniques such as D3 [47], PDQ [35], Varys [24] and Baraat [26] have been proposed to finish network flows or groups of network flows faster. The benefits from such techniques are inherently limited as the end-points of the network transfers are fixed. Corral exploits the flexibility in placing input data and the subsequent stages of the jobs, and provides benefits orthogonal to such network-level schedulers (Section 6.6). Sinbad [22] takes advantage of flexibility in the placement of output data in big data clusters, but does not consider other stages in a job.

Malleable job scheduling. The problem of task scheduling across identical servers has been studied for over four decades (e.g., [38]). The basic formulation appears in [31], where tasks have precedence constraints and each task can run on a single server. A different variant of the problem considers malleable tasks, where each task can run on multiple servers [19, 40]. Our offline planning algorithm is inspired by these papers but none of them addresses the issues of (a) malleability in the context of shared networks, and (b) balancing input data required for executing the jobs.

9 Conclusion

In this paper, we argue that current data and compute placement heuristics in big data clusters do not couple the placement of data and compute, and hence, result in increased contention in an already oversubscribed network. Production clusters run a large fraction of recurring jobs with predictable communication patterns. Leveraging this, we propose Corral which considers the future workload and jointly optimizes the location of the job data (during upload) and tasks (during execution). By solving an offline planning problem, Corral improves job performance as it separates large shuffles from each other, reduces network contention in the cluster and runs jobs across fewer racks, improving their data locality. We implemented Corral on top of Yarn.

Running production workloads on a 210 machine cluster, we show that Corral can result in 10-33% reduction in makespan and 30-56% reduction in median job completion time, compared to Yarn's capacity scheduler.

Acknowledgements. We thank Ganesh Ananthanarayan, Hitesh Ballani, Carlo Curino, Chris Douglas, Solom Heddaya, Srikanth Kandula, Subramaniam Krishnan, Rahul Potharaju, Ant Rowstron, Yuan Yu, our shepherd Minlan Yu and the anonymous SIGCOMM'15 reviewers for their useful feedback.

10 References

[1] Amazon S3. https://aws.amazon.com/s3/.
[2] Amazon Web Services. http://aws.amazon.com/.
[3] Apache Hadoop. http://hadoop.apache.org/.
[4] Apache Tez. http://hortonworks.com/hadoop/tez/.
[5] Facebook data grows by over 500 TB daily. http://tinyurl.com/96d8oqj/.
[6] Hadoop Distributed Filesystem. http://hadoop.apache.org/hdfs.
[7] Hadoop mapreduce next generation - capacity scheduler. http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.
[8] Hadoop YARN Project. http://tinyurl.com/bnadg9l.
[9] Microsoft Azure. https://azure.microsoft.com/.
[10] Microsoft Azure Storage. https://azure.microsoft.com/en-us/services/storage/.
[11] ORC File Format. http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.0.2/ds_Hive/orcfile.html.
[12] TPC Benchmark H. http://www.tpc.org/tpch/.
[13] Windows Azure's Flat Network Storage and 2012 Scalability Targets. http://bit.ly/1A4Hbjt.
[14] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. In *NSDI 2012*.
[15] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Reoptimizing Data Parallel Computing. In *NSDI'12*, 2012.
[16] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *USENIX ATC*, 2014.
[17] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys*, 2011.
[18] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI*, 2012.
[19] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. *Urbana*, 51:61801, 1990.
[20] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[21] Y. Chen, A. Ganapathi, R. Griffith, and Y. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS*, 2011.
[22] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *ACM SIGCOMM*, 2013.
[23] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.
[24] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *ACM SIGCOMM*, 2014.
[25] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[26] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *ACM SIGCOMM*, August 2014.
[27] J. Du and J. Y.-T. Leung. Complexity of Scheduling Parallel Task Systems. *SIAM J. Discret. Math.*, 1989.
[28] K. Elmeleegy. Piranha: Optimizing Short Jobs in Hadoop. *Proc. VLDB Endow.*, 2013.
[29] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *Proc. VLDB Endow.*, 2011.
[30] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.
[31] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.
[32] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource Packing for Cluster Schedulers. In *SIGCOMM*, 2014.
[33] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *SOCC*, 2011.
[34] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, 2011.
[35] C. Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.
[36] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.
[37] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the Tenant-provider Gap in Cloud Services. In *SOCC*, 2012.
[38] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors.

ACM Computing Surveys (CSUR), 1999.

[39] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in Mapreduce Applications. In *ACM SIGMOD*, 2012.

[40] R. Lepère, D. Trystram, and G. J. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *International Journal of Foundations of Computer Science*, 13(04):613–627, 2002.

[41] M. Li, D. Subhraveti, A. R. Butt, A. Khasymski, and P. Sarkar. CAM: A Topology Aware Minimum Cost Flow Based Resource Manager for MapReduce Applications in the Cloud. In *HPDC*, 2012.

[42] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. VLDB Endow.*

[43] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In *SC*, 2011.

[44] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive-a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[45] J. Turek, J. L. Wolf, and P. S. Yu. Approximate Algorithms Scheduling Parallelizable Tasks. In *SPAA*, 1992.

[46] G. Wang, A. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*, 2009.

[47] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM*, 2011.

[48] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[50] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. Scope: parallel databases meet mapreduce. *VLDB J.*, 21(5):611–636, 2012.

APPENDIX

A LP Relaxation

To estimate the quality of our heuristics for the planning problem, we formulate a related integer linear program (ILP). The ILP would provide a lower bound on the makespan and the average completion time for *any* algorithm which plans at the granularity of racks and jobs. However, as the ILP is computationally expensive to solve in practice, we relax it to a linear program (LP), whose solution is still a lower bound to our problem. Thus, if the solution from our heuristics is close to that of the LP, it is guaranteed to be close to optimal. We emphasize that the LP relaxation is not used by Corral, but only serves as a benchmark for the solutions we developed in Section 4.2.

We first describe the Integer Linear Program (ILP). Let T be the makespan of the (unknown to us) optimal solution to the planning problem. For every job j and every number of racks $r \in \{1, \ldots, R\}$, we introduce a variable $x_{jr} \in \{0, 1\}$. In the ILP, x_{jr} equals 1 if job j is assigned r racks, and 0 otherwise. Relaxing the integrality constraint, we obtain the following LP.

$$\text{Minimize}_{\{x_{jr}\}} \quad T \qquad (LP - Batch) \tag{1}$$

$$\text{Subject to} \quad \sum_r x_{jr} = 1, \quad \forall j \tag{2}$$

$$T \geq \sum_r x_{jr} L_j(r), \quad \forall j \tag{3}$$

$$TR \geq \sum_{j,r} x_{jr} L_j(r) \cdot r, \tag{4}$$

$$x_{jr} \in [0, 1], \forall j, \forall r \tag{5}$$

The constraint (2) ensures that all jobs are completed. In the integral solution corresponding to a feasible schedule, for each j exactly one x_{jr} equals 1, so the constraint (2) is satisfied. The constraints (3) and (4) give a lower bound on the makespan. Constraint (3) asserts that the makespan is at least as large as the completion time of every job j (since in the integral solution corresponding to a feasible schedule, the right hand side of (3) exactly equals the running time of job j). Constraint (4) is a capacity constraint. It indicates that the capacity used by the LP schedule (the right hand side of inequality) is at most the available capacity (the left hand side).

We emphasize that LP-Batch is a *relaxation* of our original planning problem (Section 4.1) as it does not give an actual schedule (e.g., does not specify the time when a job should start) but only provides the number of racks a job has to be allocated. Still, any feasible schedule (particularly, an optimal schedule) should satisfy the constraints of the LP and thus, the cost (i.e, makespan) returned by the LP is a lower bound on the cost of any schedule.

In the online scenario, we are interested in minimizing the average completion time. Accordingly, we formulate a mixed integer program with the objective function:

$$\text{Minimize}_{\{x_{jr}\},\{d_{jr}\}} \quad \frac{1}{J} \sum_{j,r} x_{jr} L_j(r) + d_{jr} \quad (LP{-}Online) \tag{6}$$

where $d_{jr} > 0$ is the delay in scheduling job j relative to its arrival. The LP for the online case is more involved and we omit the full description for brevity.

The above bounds are useful for bounding the gap between the performance of our heuristics and the optimal solution to the planning problem (defined in Section 3.3). They should not be viewed as theoretical guarantees for the entire Corral framework, which includes also the (work-conserving) cluster scheduler. Nevertheless, as our experiments indicate, obtaining good solution for the planning problem leads to overall performance enhancements.