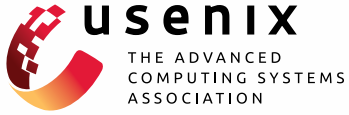


The Power of Choice in Data-Aware Cluster Scheduling

Shivaram Venkataraman and Aurojit Panda, ***University of California, Berkeley;***
Ganesh Ananthanarayanan, ***Microsoft Research;***
Michael J. Franklin and Ion Stoica, ***University of California, Berkeley***

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/venkataraman

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the
11th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

The Power of Choice in Data-Aware Cluster Scheduling

Shivaram Venkataraman[1], Aurojit Panda[1], Ganesh Ananthanarayanan[2], Michael J. Franklin[1], Ion Stoica[1]
[1]*UC Berkeley* [2] *Microsoft Research*

Abstract

Providing *timely* results in the face of rapid growth in data volumes has become important for analytical frameworks. For this reason, frameworks increasingly operate on only a *subset* of the input data. A key property of such sampling is that combinatorially many subsets of the input are present. We present KMN, a system that leverages these choices to perform *data-aware* scheduling, i.e., minimize time taken by tasks to read their inputs, for a DAG of tasks. KMN not only uses choices to co-locate tasks with their data but also percolates such combinatorial choices to downstream tasks in the DAG by launching a *few additional* tasks at every upstream stage. Evaluations using workloads from Facebook and Conviva on a 100-machine EC2 cluster show that KMN reduces average job duration by 81% using just 5% additional resources.

1 Introduction

Data-intensive computation frameworks drive many modern services like web search indexing and recommendation systems. Computation frameworks (e.g., Hadoop [14], Spark [60], Dryad [38]) translate a *job* into a DAG of many small *tasks*, and execute them efficiently on compute *slots* across large clusters. Tasks of *input* stages (e.g., map in MapReduce or extract in Dryad) read their data from distributed storage and pass their outputs to the downstream *intermediate* tasks (e.g., reduce in MapReduce or full-aggregate in Dryad).

The efficient execution of these predominantly I/O-intensive tasks is predicated on *data-aware scheduling*, i.e., minimizing the time taken by tasks to read their data. Widely deployed techniques for data-aware scheduling execute tasks on the same machine as their data (if the data is on one machine, as for input tasks) [8, 59] and avoid congested network links (when data is spread across machines, as for intermediate tasks) [13, 24]. However, despite these techniques, we see that production jobs in Facebook's Hadoop cluster are slower by 87% compared to *perfect data-aware scheduling* (§2.3). This is because, in multi-tenant clusters, compute slots that are ideal for data-aware task execution are often unavailable.

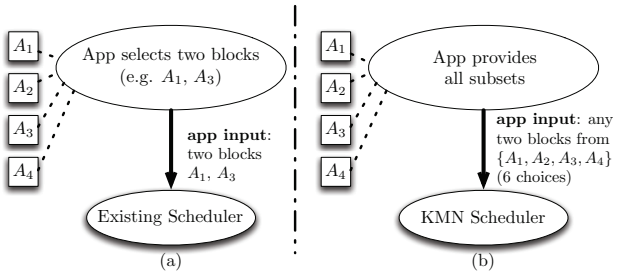


Figure 1: **"Late binding" allows applications to specify more inputs than tasks and schedulers dynamically choose task inputs at execution time.**

The importance of data-aware scheduling is increasing with rapid growth in data volumes [31]. To cope with this data growth and yet provide timely results, there is a trend of jobs using only a *subset* of their data. Examples include sampling-based approximate query processing systems [5,12] and machine learning algorithms [16,42]. A key property of such jobs is that they can compute on *any of the combinatorially many* subsets of the input dataset without compromising application correctness. For example, say a machine learning algorithm like stochastic gradient descent [16] needs to compute on a 5% uniform random sample of data. If the data is spread over 100 blocks then the scheduler can choose any 5 blocks and has $\binom{100}{5}$ input choices for this job.

Our goal is to *leverage the combinatorial choice of inputs for data-aware scheduling*. Current schedulers require the application to select a subset of the data on which the scheduler runs the job. This prevents the scheduler from taking advantage of available choices. In contrast, we argue for "late binding" *i.e.,* choosing the subset of data dynamically depending on the current state of the cluster (see Figure 1). This dramatically increases the number of data local slots for input tasks (e.g., map tasks), which increases the probability of achieving data locality even during high cluster utilizations.

Extending the benefits of choice to intermediate stages (e.g., reduce) is challenging because they consume all the outputs produced by upstream tasks. Thus, they have no choice in picking their inputs. When upstream outputs are not evenly spread across machines, the oversubscribed network links, typically cross-rack switch

links [24], become the bottleneck. We introduce choices for intermediate stages by launching a *small number of additional tasks* in the previous stage. As an example, consider a job with 400 map tasks and 50 reduce tasks. By launching 5% extra map tasks (420 tasks), we can pick the 400 map outputs that best avoid congested links.

Choosing the best upstream outputs used by intermediate stages is non-trivial due to complex communication patterns like many-to-many (for reduce tasks) and many-to-one (for joins). In fact, selecting the best upstream outputs can be shown to be NP-hard. We develop an efficient round-robin heuristic that attempts to balance data transfers evenly across cross-rack switch links. Further, upstream tasks do not all finish simultaneously due to *stragglers* [13, 61]. We handle stragglers in upstream tasks using a *delay*-based approach that balances the gains in balanced network transfers against the time spent waiting for stragglers. In the above example, for instance, it may schedule reduce tasks based on the earliest 415 map tasks and ignore the last 5 stragglers.

In summary, we make the following contributions:

- Identify the trend of combinatorial choices in inputs of jobs and leverage this for data-aware scheduling.
- Extend the benefits of choices to a DAG of stages by running a *few* extra tasks in each upstream stage.
- Build KMN, a system for analytics frameworks to seamlessly benefit from the combinatorial choices.

We have implemented KMN inside Spark [60]. We evaluate KMN using jobs from a production workload at Conviva, a video analytics company, and by replaying a trace from a production Hadoop cluster at Facebook. Our experiments on an EC2 cluster with 100 machines show that we can reduce average job duration by 81% (93% of ideal improvements) compared to Spark's scheduler. Our gains are due to KMN achieving 98% memory locality for input tasks and improving intermediate data transfers by 48%, while using $\leq 5\%$ *extra resources*.

2 Choices and Data-Awareness

In this section we first discuss application trends that result in increased choices for scheduling (§2.1). We then explain data-aware scheduling (§2.2) and quantify its potential benefit in production clusters (§2.3).

2.1 Application Trends

With the rapid increase in the volume of data collected, it has become prohibitively expensive for data analytics frameworks to operate on all of the data. To provide timely results, there is a trend towards trading off accuracy for performance. Quick results obtained from just part of the dataset are often *good enough*.

(1) Approximate Query Processing: Many analytics frameworks support approximate query processing (AQP) using standard SQL syntax (e.g., BlinkDB [5], Presto [29]). They power many popular applications like exploratory data analysis [19, 54] and interactive debugging [3]. For example, products analysts could use AQP systems to quickly decide if an advertising campaign needs to be changed based on a sample of click through rates. AQP systems can bound both the time taken and the quality of the result by selecting appropriately sized inputs (samples) to met the deadline and error bound. Sample sizes are typically small relative to the original data (often, one-twentieth to one-fifth [43]) and many equivalent samples exist. Thus, sample selection presents a significant opportunity for smart scheduling.

(2) Machine Learning: The last few years has seen the deployment of large-scale distributed machine learning algorithms for commercial applications like spam classification [40] and machine translation [18]. Recent advances [17] have introduced stochastic versions of these algorithms, for example stochastic gradient descent [16] or stochastic L-BFGS [53], that can use *small random data samples* and provide statistically valid results even for large datasets. These algorithms are iterative and each iteration processes only a small sample of the data. Stochastic algorithms are agnostic to the sample selected in each iteration and support flexible scheduling.

(3) Erasure Coded Storage: Rise in data volumes have also led to clusters employing efficient storage techniques like erasure codes [50]. Erasure codes provide fault tolerance by storing k extra parity blocks for every n data blocks. Using *any* n data blocks of the $(n+k)$ blocks, applications can compute their input. Such storage systems also provide choices for data-aware scheduling.

Note that while the above applications provide an opportunity to pick *any* subset of the input data, our system can also handle custom sampling functions, which generate samples based on application requirements.

2.2 Data-Aware Scheduling

Data aware scheduling is important for both the input as well as intermediate stages of jobs due to their IO-intensive nature. In the input stage, tasks reads their input from a single machine and the natural goal is *locality* i.e. to schedule the task on a machine that stores its input (§2.2.1). For intermediate stages, tasks have their input spread across multiple machines. In this case, it is not possible to co-locate the task with all its inputs. Instead, the goal in this case is to schedule the task at a machine

that minimizes the time it takes to transfer all remote inputs. As over-subscribed cross-rack links are the main bottleneck in reads [22], we seek to *balance* the utilization of these links (§2.2.2).

2.2.1 Memory Locality for Input Tasks

Riding on the trend of falling memory prices, clusters are increasingly caching data in memory [11, 58]. As memory bandwidths are about $10\times - 100\times$ greater than the fastest network bandwidths, data reads from memory provide dramatic acceleration for the IO-intensive analytics jobs. However, to reap the benefits of in-memory caches, tasks have to be scheduled with *memory locality*, i.e., on the same machine that contains their input data. Obtaining memory locality is important for timely completion of interactive approximate queries [9]. Iterative machine learning algorithms typically run 100's of iterations and lack of memory locality results in huge slowdown per iteration and the overall job.

Achieving memory locality is a challenging problem in clusters. Since in-memory storage is used only as a cache, data stored in memory is typically not replicated. Further, the amount of memory in a cluster is relatively small (often by three orders of magnitude [9]) when compared to stable storage: this difference means that replicating data in memory is not practical. Therefore, techniques for improving locality [8] developed for disk-based replicated storage are insufficient; they rely on the probability of locality increasing with the number of replicas. Further, as job completion times are dictated by the slowest task in the job, improving performance requires memory locality for *all* its tasks [11].

These challenges are reflected in production Hadoop clusters. A Facebook trace from 2010 [8, 21] shows that less than 60% of tasks achieve locality *even with* three replicas. As in-memory data is not replicated, it is harder for jobs to achieve memory locality for all their tasks.

2.2.2 Balanced Network for Intermediate Tasks

Intermediate stages of a job have communication patterns that result in their tasks reading inputs from many machines (e.g., all-to-all "shuffle" or many-to-one "join" stages). For I/O intensive intermediate tasks, the time to access data across the network dominates the running time, more so when intermediate outputs are stored in memory. Despite fast network links [56] and newer topologies [6, 35], bandwidths between machines connected to the same rack switch are still $2\times$ to $5\times$ higher than to machines outside the rack switch via the network core. Thus the runtime for an intermediate stage is dictated by the amount of data transferred across racks. Prior work has also shown that reducing cross-rack hotspots, i.e., optimizing the *bottleneck* cross-rack link [13, 24] can significantly improve performance.

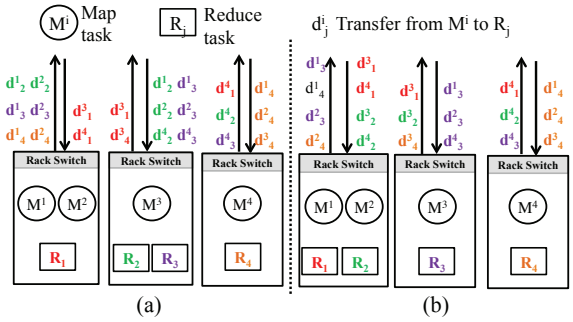


Figure 2: **Value of balanced network usage for a job with 4 map tasks and 4 reduce tasks. The left-hand side has unbalanced cross-rack links (maximum of 6 transfers, minimum of 2) while the right-hand side has better balance (maximum of 4 transfers, minimum of 3).**

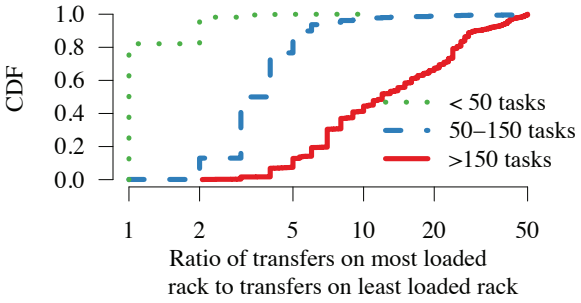


Figure 3: **CDF of cross-rack skew for the Facebook trace split by number of map tasks. Reducing cross-rack skew improves intermediate stage performance.**

Given the over-subscribed cross-rack links and the slowest tasks dictating job completion, it is important to *balance* traffic on the cross-rack links [15]. Figure 2 illustrates the result of having unbalanced cross-rack links. The schedule in Figure 2(b) results in a *cross-rack skew*, i.e., ratio of the highest to lowest used network links, of only $\frac{4}{3}$ (or 1.33) as opposed to $\frac{6}{2}$ (or 3) in Figure 2(a).

To highlight the importance of cross-rack skew, we used a trace of Hadoop jobs run in a Facebook cluster from 2010 [21] and computed the cross-rack skew ratio. Figure 3 shows a CDF of this ratio and is broken down by the number of map tasks in the job. From the figure we can see that for jobs with $50-150$ map tasks more than half of the jobs have a cross-rack skew of over $4\times$. For larger jobs we see that the median is $15\times$ and the 90^{th} percentile value is in excess of $30\times$.

2.3 Potential Benefits

How much do the above-mentioned lack of locality and imbalanced network usage hurt jobs? We estimate the potential for data-aware scheduling to speed up jobs using the same Facebook trace (described in detail in §6).

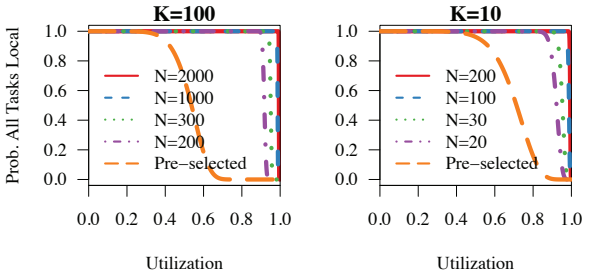


Figure 4: **Probability of input-stage locality when choosing any K out of N blocks. The scheduler can choose to execute a job on any of $\binom{K}{N}$ samples.**

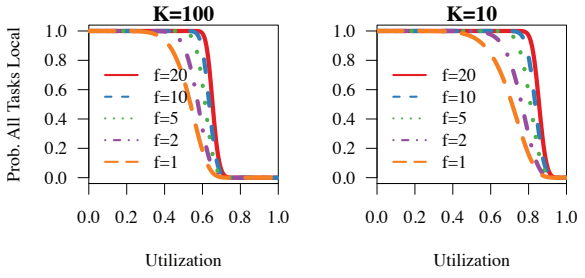


Figure 5: **Probability of input-stage locality when using a sampling function which outputs f disjoint samples. Sampling functions specify additional constraints for samples.**

We mimic job performance with an ideal data-aware scheduler using a "what-if" simulator. Our simulator is unconstrained and (i) assigns memory locality for *all* the tasks in the input phase (we assume $20\times$ speed up for memory locality [45] compared to reading data over the network based on our micro-benchmark) and (ii) places tasks to *perfectly balance* cross-rack links. We see that jobs speed up by 87.6% on average with such ideal data-aware scheduling.

Given these potential benefits, we have designed KMN, a scheduling framework that exploits the available choices to improve performance. At the heart of KMN lie scheduling techniques to increase locality for input (§3) stages and balance network usage for intermediate (§4) stages. In §5, we describe an interface that allows applications to specify all available choices to the scheduler.

3 Input Stage

For the input stage (*i.e.,* the map stage in MapReduce or the extract stage in Dryad) accounting for combinatorial choice leads to improved locality and hence reduced completion time. Here we analyze the improvements in locality in two scenarios: in §3.1 we look at jobs which can use any K of the N input blocks; in §3.2 we look at jobs which use a custom sampling function.

We assume a cluster with s compute slots per machine. Tasks operate on one input block each and input blocks are uniformly distributed across the cluster, this is in line with the block placement policy used by Hadoop. For ease of analysis we assume machines in the cluster are uniformly utilized (*i.e.,* there are no hot-spots). In our evaluation §6) we consider hot-spots due to skewed input-block and machine popularity.

3.1 Choosing any K out of N blocks

Many modern systems *e.g.,* BlinkDB [5], Presto [29], AQUA [2] operate by choosing a random subset of blocks from shuffled input data. These systems rely on the observation that block sampling [20] is statistically equivalent to uniform random sampling (page 243 in [55]) when each block is itself a random sample of the overall population. Given a sample size K, these systems can operate on any K input blocks *i.e.,* for an input of size N the scheduler can choose any one of $\binom{N}{K}$ combinations.

In the cluster setup described above, the probability that a task operating on an input block gets locality is $p_t = 1 - u^s$ where u is the cluster utilization (probability that all slots in a machine are busy is $= u^s$). For such a cluster the probability for K out of N tasks getting locality is given by the binomial CDF function with the probability of success $= p_t$, i.e., $1 - \sum_{i=0}^{K-1} \binom{N}{i} p_t^i (1-p_t)^{N-i}$.

The dominant factor in this probability is the ratio between K and N. In Figure 4 we fix the number of slots per machine to $s = 8$ and plot the probability of $K = 10$ and $K = 100$ tasks getting locality in a job with varying input size N and varying cluster utilization. We observe that the probability of achieving locality is high even when 90% of the cluster is utilized. We also compare this to a baseline that does not exploit this combinatorial choice and pre-selects a random K blocks *beforehand*. For the baseline the probability that all tasks are local drops dramatically even with cluster utilization of 60% or less.

3.2 Custom Sampling Functions

Some systems require additional constraints on the samples used and use custom sampling functions. These sampling functions can be used to produce several K-block samples and the scheduler can pick *any* sample. The scheduler is however constrained to use *all* of the K-blocks from one sample. We consider a sampling function that produces f disjoint samples and analyze locality improvements in this setting.

As noted previous, the probability of a task getting locality is $p_t = 1 - u^s$. The probability that all K blocks in a sample get locality is p_t^K. Since the f samples are disjoint (and therefore the probability of achieving locality is independent) the probability that at least one among the f samples can achieve locality is $p_j = 1 -$

$(1-p_i^K)^f$. Figure 5 shows the probability of $K = 10$ and $K = 100$ tasks achieving locality with varying utilization and number of samples. We see that the probability of achieving locality significantly increases with f. At $f = 5$ we see that small jobs (10 tasks) can achieve complete locality even when the cluster is 80% utilized.

We thus find that accounting for combinatorial choices can greatly improve locality for the input stage. Next we analyze improvements for intermediate stages.

4 Intermediate Stages

Intermediate stages of jobs commonly involve one-to-all (broadcast), many-to-one (coalesce) or many-to-many (shuffle) network transfers [23]. These transfers are network-bound and hence, often slowed down by congested cross-rack network links. As described in §2.2.2, data-aware scheduling can improve performance by better placement of both upstream and downstream tasks to balance the usage of cross-rack network links.

While effective heuristics can be used in scheduling downstream tasks to balance network usage (we deal with this in §5), they are nonetheless limited by the locations of the outputs of upstream tasks. Scheduling upstream tasks to balance the locations of their outputs across racks is often complicated due to many dynamic factors in clusters. First, they are constrained by data locality (§3) and compromising locality is detrimental. Second, the utilization of the cross-rack links when downstream tasks start executing are hard to predict in multi-tenant clusters. Finally, even the size of upstream outputs varies across jobs and are not known beforehand.

We overcome these challenges by scheduling a *few additional* upstream tasks. For an upstream stage with K tasks, we schedule M tasks ($M > K$). Additional tasks increase the likelihood that task outputs are distributed across racks. This allows us to choose the "best" K out of M upstream tasks, out of $\binom{M}{K}$ choices, to minimize cross-rack network utilization. In the rest of this section, we show analytically that a few additional upstream tasks can significantly reduce the imbalance (§4.1). §4.2 describes a heuristic to pick the best K out of M upstream tasks. However, not all M upstream tasks may finish simultaneously because of stragglers; we modify our heuristic to account for stragglers in §4.3.

4.1 Additional Upstream Tasks

While running additional tasks can balance network usage, it is important to consider how many additional tasks are required. Too many additional tasks can often lead to worsening of overall cluster performance.

We analyze this using a simple model of the scheduling of upstream tasks. For simplicity, we assume that

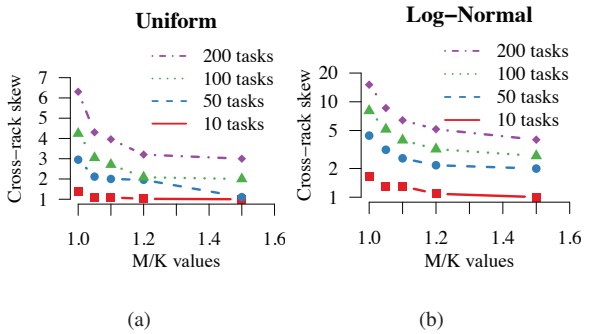


Figure 6: **Cross-rack skew as we vary M/K for uniform and log-normal distributions. Even 20% extra upstream tasks greatly reduces network imbalance for later stages.**

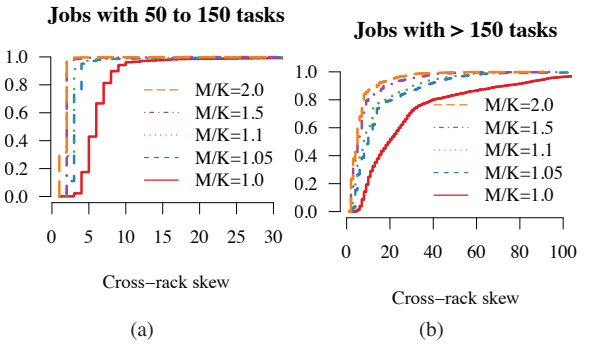


Figure 7: **CDF of cross-rack skew as we vary M/K for the Facebook trace.**

upstream task outputs are equal in size and network links are equally utilized. We only model tasks at the level of racks and evaluate the cross-rack skew (ratio of the rack with largest and smallest number of upstream tasks) using both synthetic distributions of upstream task locations as well as data from our Facebook trace.

Synthetic Distributions: We first consider a scheduler that places tasks on racks uniformly at random. Figure 6(a) plots the cross-rack skew in a 100 rack cluster for varying values of K (*i.e.,* the stage's desired number of tasks) and M/K (*i.e.,* the fraction of additional tasks launched). We can see that even with a scheduler that places the upstream tasks uniformly, there is significant skew for large jobs when there are no additional tasks ($\frac{M}{K} = 1$). This is explained by the balls and bins problem [46] where the maximum imbalance is expected to be $O(logn)$ when distributing n balls.

However, we see that even with 10% to 20% additional tasks ($\frac{M}{K} = 1.1 - 1.2$) the cross-rack skew is reduced by $\geq 2\times$. This is because when the number of upstream tasks, n is > 12, $0.2n > logn$. Thus, we can avoid most of the skew with just a few extra tasks.

We also repeat this study with a log normal distribution ($\theta = 0, m = 1$) of upstream task placement; this is more skewed compared to the uniform distribution.

However, even with a log-normal distribution, we again see that a few extra tasks can be very effective at reducing skew. This is because the expected value of the most loaded bin is still linear and using $0.2n$ additional tasks is sufficient to avoid most of the skew.

Facebook Distributions: We repeat the above analysis using the number and location of upstream tasks of a phase in the Facebook trace (used in §2.2.2). Recall the high cross-rack skew in the Facebook trace. Despite that, again, a few additional tasks suffices to eliminate a large fraction of the skews. Figure 7 plots the results for varying values of $\frac{M}{K}$ for different jobs. A large fraction of the skew is reduced by running just 10% more tasks. This is nearly 66% of the reductions we get using $\frac{M}{K} = 2$.

In summary we see that running a few extra tasks is an effective strategy to reduce skew, both with synthetic as well as real-world distributions. We next look at mechanisms that can help us achieve such reduction.

4.2 Selecting Best Upstream Outputs

The problem of selecting the best K outputs from the M upstream tasks can be stated as follows: We are given M upstream tasks $U = u_1 ... u_M$, R downstream tasks $D = d_1 ... d_R$ and their corresponding rack locations. Let us assume that tasks are distributed over racks $1...L$ and let $U' \subset U$ be some set of K upstream outputs. Then for each rack we can define the uplink cost C_{2i-1} and downlink cost C_{2i} using a cost function $C_i(U', D)$. Our objective then is to select U' to minimize the most loaded link i.e.

$$\arg\min_{U'} \max_{i \in 2L} C_i(U', D)$$

While this problem is NP-Hard [57], many approximation heuristics have been developed. We use a heuristic that corresponds to spreading our choice of K outputs across as many racks as possible.[1]

Our implementation for this approximation heuristic is shown in Algorithm 1. We start with the list of upstream tasks and build a hash map that stores how many tasks were run on each rack. Next we sort the tasks first by their index within a rack and then by the number of tasks in the rack. This sorting criteria ensures that we first see one task from each rack, thus ensuring we spread our choices across racks. We use an additional heuristic of favoring racks with more outputs to help our downstream task placement techniques (§5.2.2). The main computation cost in this method is the sorting step and hence this runs in $O(MlogM)$ time for M tasks.

[1]This problem is an instance of the facility location problem [26] where we have a set of clients (downstream tasks), set of potential facility locations (upstream tasks), a cost function that maps facility locations to clients (link usage). Our heuristic follows from picking a facility that is farthest from the existing set of facilities [30].

Algorithm 1 Choosing K upstream outputs out of M using a round-robin strategy

```
1:  Given: upstreamTasks - list with rack, index within rack for each task
2:  Given: K - number of tasks to pick
3:  // Number of upstream tasks in each rack
4:  upstreamRacksCount = map()
5:
6:  // Initialize
7:  for task in upstreamTasks do
8:      upstreamRacksCount[task.rack] += 1
9:  end for
10:
11: // Sort the tasks in round-robin fashion
12: roundRobin = upstreamTasks.sort(CompareTasks)
13: chosenK = roundRobin[0 : K]
14: return chosenK
15:
16: procedure CompareTasks(task1, task2)
17:     if task1.idx != task2.idx then
18:         // Sort first by index
19:         return task1.idx < task2.idx
20:     else
21:         // Then by number of outputs
22:         numRack1 = upstreamRacksCount[task1.rack]
23:         numRack2 = upstreamRacksCount[task2.rack]
24:         return numRack1 > numRack2
25:     end if
26: end procedure
```

4.3 Handling Upstream Stragglers

While the previous section described a heuristic to pick the best K out of M upstream outputs, waiting for all M can be inefficient due to *stragglers*. Stragglers in the upstream stage can delay completion of some tasks which cuts into the gains obtained by balancing the network links. Stragglers are a common occurrence in clusters with many clusters reporting significantly slow tasks despite many prevention and speculation solutions [10, 13, 61]. This presents a trade-off in waiting for all M tasks and obtaining the benefits of choice in picking upstream outputs against the wasted time for completion of all M upstream tasks including stragglers. Our solution for this problem is to schedule downstream tasks at some point after K upstream tasks have completed but not wait for the stragglers in the M tasks. We quantify this trade-off with analysis and micro-benchmarks.

4.3.1 Stragglers vs. Choice

We study the impact of stragglers in the Facebook trace when we run 2%, 5% and 10% extra tasks (i.e., $\frac{M}{K} = 1.02, 1.05, 1.1$). We compute the difference between the time taken for the fastest K tasks and the time to com-

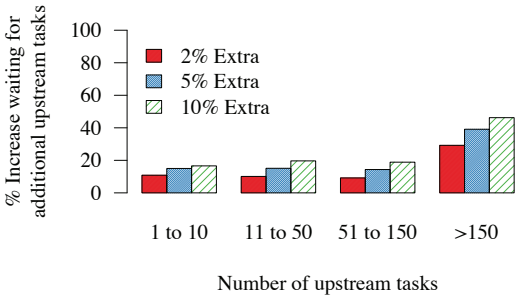


Figure 8: **Percentage of time spent waiting for additional upstream tasks when running 2%, 5% or 10% extra tasks. Stage completion time can be increased by up** $20\% - 40\%$ **due to stragglers.**

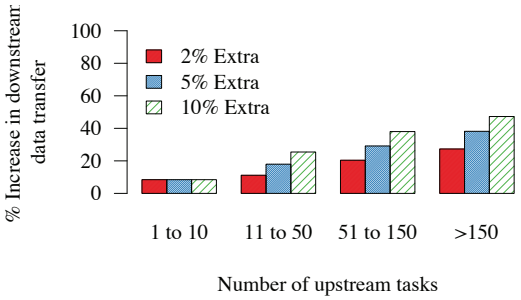


Figure 9: **Percentage of additional time spent in downstream data transfer when not using choices from 2%, 5% or 10% extra tasks. Decrease in choice increases data transfer time by** $20\% - 40\%$**.**

plete all M tasks. Figure 8 shows that waiting for the extra tasks can inflate the completion of the upstream phase by $20\% - 40\%$ (for jobs with > 150 tasks). Also, the trend of using a large number of small tasks [48] for interactive jobs will only worsen such inflation. On the other hand avoiding upstream stragglers by using the fastest tasks reduces the available choice. Consequently, the time taken for downstream data transfer increases. The lack of choices from extra tasks means we cannot balance network usage. Figure 9 shows that not using choice from additional tasks can increase data transfer time by 20% for small jobs (11 to 50 tasks) and up to 40% for large jobs (> 150 tasks). We now devise a simple approach to balance between the above two factors—waiting for upstream stragglers versus losing choice for downstream data transfer.

4.3.2 Delayed Stage Launch

The problem we need to solve can be formulated as: we have M upstream tasks $u_1, u_2, ..., u_M$ and for each task we have corresponding rack locations. Our goal is to find the optimal *delay* after the first K tasks have finished, such that the overall time taken is minimized. In other words, our goal is to find the optimal K' tasks to wait for before starting the downstream tasks.

We begin with assuming an oracle that can give us the task finish times for all the tasks. Given such a oracle we can sort the tasks in an increasing order of finish times such that $F_j \geq F_i \; \forall j > i$. Let us define the waiting delay for tasks $K+1$ to M as $D_i = F_i - F_k \; \forall i > k$. We also assume that given K' tasks, we can compute the optimal K tasks to use (§4.2) and the estimated transfer time $S_{K'}$.

Our problem is to pick K' ($K \leq K' \leq M$) such that the total time for the data transfer is minimized. That is we need to pick K' such that $F_k + D_{k'} + S_{k'}$ is minimized. In this equation F_k is known and independent of K'. Of the other two, $D_{k'}$ increases as k' goes from K to M, while $S_{k'}$ decreases. However as the sum of an increasing and decreasing function is not necessarily convex [2] it isn't easy to minimize the total time taken.

Delay Heuristic: While the brute-force approach would require us to try all values from K to M, we develop two heuristics that allow us to bound the search space and quickly find the optimal value of K'.

- *Bounding transfer*: At the beginning of the search procedure we find the maximum possible improvement we can get from picking the best set of tasks. Whenever the delay $D_{K'}$ is greater than the maximum improvement, we can stop the search as the succeeding delays will increase the total time.
- *Coalescing tasks*: We can also coalesce a number of task finish events to further reduce the search space. For example we can coalesce task finish events which occur close together by time i.e., cases $D_{i+1} - Di < \delta$. This will mean our result is off by at most δ from the optimal, but for small values of δ we can coalesce tasks of a wave that finish close to each other.

Using these heuristics we can find the optimal number of tasks to wait for quickly. For example, in the Facebook trace described before using $M/K = 1.1$ or 10% extra tasks, determining the optimal wait time for a job requires looking at less than 4% of all configurations when we use a coalescing error of 1%. We found coalescing tasks to be particularly useful as even with a δ of 0.1% we need to look at around 8% of all possible configurations. Running without any coalescing is infeasible since it takes ≈ 1000 ms.

Finally, we relax our assumption of an oracle as follows. While the task finish times are not exactly known beforehand, we use job sizes to figure out if the same job has been run before. Based on this we use the job history to predict the task finish times. This approach should work well for clusters that have many jobs run periodically [36]. In case the job history is not available we can fit the tasks length distribution using the first few task finish times and use that to get approximate task finish times for the rest of the tasks [28].

[2] Take any non-convex function and make its increasing region F_i and its decreasing region F_d and it can be seen that the sum isn't convex.

```
// SQL Query
SELECT status, SUM(quantity)
FROM items
GROUP BY status

// Spark Query
kv = file.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()

// KMN Query
sample = file.blockSample(0.1, sampler=None)
kv = sample.map{ li =>
  (li.l_linestatus,li.quantity)}
result = kv.reduceByKey{(a,b) =>
  a + b}.collect()
```

Figure 10: **An example of a query in SQL, Spark and KMN**

5 System Implementation

We have built KMN on top of Spark [60], an open-source cluster computing framework. Our implementation is based on Spark version 0.7.3 and KMN consists of 1400 lines of Scala code. In this section we discuss the features of our implementation and implementation challenges.

5.1 Application Interface

We define a `blockSample` operator which jobs can use to specify input constraints (for instance, use K blocks from file F) to the framework.The `blockSample` operator takes two arguments: the ratio $\frac{K}{N}$ and a *sampling function* that can be used to impose constraints. The sampling function can be used to choose user-defined sampling algorithms (*e.g.,* stratified sampling). By default the sampling function picks any K of N blocks.

Consider an example SQL query and its corresponding Spark [60] version shown in Figure 10. To run the same query in KMN we just need to prefix the query with the `blockSample` operator. The *sampler* argument is a Scala closure and passing `None` causes the scheduler to use the default function which picks any K out of the N input blocks. This design can be readily adapted to other systems like Hadoop MapReduce and Dryad.

KMN also provides an interface for jobs to introspect which samples where used in a computation. This can be used for error estimation using algorithms like Bootstrap [4] and also provides support for queries to be repeated. We implement this in KMN by storing the K partitions used during computation as a part of a job's lineage. Using the lineage also ensures that the same samples are used if the job is re-executed during fault recovery [60].

5.2 Task Scheduling

We modify Spark's scheduler in KMN to implement the techniques described in earlier sections.

5.2.1 Input Stage

Schedulers for frameworks like MapReduce or Spark typically use a slot-based model where the scheduler is invoked whenever a slot becomes available in the cluster. In KMN, to choose any K out of N blocks we modify the scheduler to run tasks on blocks local to the first K available slots. To ensure that tasks don't suffer from resource starvation while waiting for locality, we use a timeout after which tasks are scheduled on any available slot. Note that, choosing the first K slots provides a sample similar or slightly better in quality compared to existing systems like Aqua [2] or BlinkDB [5] that reuse samples for short time periods. To schedule jobs with custom sampling functions, we similarly modify the scheduler to choose among the available samples and run the computation on the sample that has the highest locality.

5.2.2 Intermediate Stage

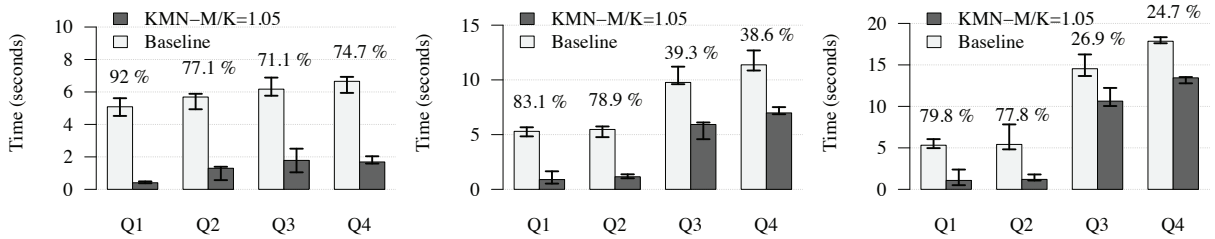
Existing cluster computing frameworks like Spark and Hadoop place intermediate stages without accounting for their dependencies. However smarter placement which accounts for a tasks' dependencies can improve performance. We implemented two strategies in KMN:

Greedy assignment: The number of cross-rack transfers in the intermediate stage can be reduced by co-locating map and reduce tasks (more generally any dependent tasks). In the greedy placement strategy we maximize the number of reduce tasks placed in the rack with the most map tasks. This strategy works well for small jobs where network usage can be minimized by placing all the reduce tasks in the same rack.

Round-robin assignment: While greedy placement minimizes the number of transfers from map tasks to reduce tasks it results in most of the data being sent to one or a few racks. Thus the links into these racks are likely to be congested. This problem can be solved by distributing tasks across racks while simultaneously minimizing the amount of data sent across racks. This can be achieved by evenly distributing the reducers across racks with map tasks. This strategy can be shown to be optimal if we know the map task locations and is similar in nature to the algorithm described in §4.2. We perform a more detailed comparison of the two approaches in §6

5.3 Support for extra tasks

One consequence of launching extra tasks to improve performance is that the cluster utilization could be af-



(a) Conviva queries with 1% Sampling (b) Conviva queries with 5% Sampling (c) Conviva queries with 10% Sampling

Figure 11: **Comparing baseline and KMN-1.05 with sampling-queries from Conviva. Numbers on the bars represent percentage improvement when using KMN-$M/K = 1.05$.**

fected by these extra tasks. To avoid utilization spikes, in KMN the value for M/K (the percentage of extra tasks to launch) can only be set by the cluster administrator and not directly by the application. Further, we implemented support for killing tasks once the scheduler decides that the tasks' output is not required. Killing tasks in Spark is challenging as tasks are run in threads and many tasks share the same process. To avoid expensive clean up associated with killing threads [1], we modified tasks in Spark to periodically poll and check a status bit. This means that tasks sometimes could take a few seconds more before they are terminated, but we found that this overhead was negligible in practice.

In KMN, using extra tasks is crucial in extending the flexibility of many choices throughout the DAG. In §3 and §4 we discussed how to use the available choices in the input and intermediate stages in a DAG. However, jobs created using frameworks like Spark or DryadLINQ can extend across many more stages. For example, complex SQL queries may use a map followed a shuffle to do a group-by operation and follow that up with a join. One solution to this would be run more tasks than required in every stage to retain the ability to choose among inputs in succeeding stages. However we found that in practice this does not help very much. In frameworks like Spark which use lazy evaluation, every stage following than the first stage is treated as an intermediate stage. As we use a round-robin strategy to schedule intermediate tasks (§5.2.2), the outputs from the first intermediate stage are already well spread out across the racks. Thus there isn't much skew across racks that affects the performance of following stages. In evaluation runs we saw no benefits for later stages of long DAGs.

6 Evaluation

We evaluate the benefits of KMN using two approaches: first we run approximate queries used in production at Conviva, a video analytics company, and study how KMN compares to using existing schedulers with pre-selected samples. Next we analyze how KMN behaves in a shared cluster, by replaying a workload trace obtained from Facebook's production Hadoop cluster.

Metric: In our evaluation we measure percentage improvement of job completion time when using KMN. We define percentage improvement as:

$$\%\ \text{Improvement} = \frac{\text{Baseline Time} - \text{KMN Time}}{\text{Baseline Time}} \times 100$$

Our evaluation shows that,

- KMN improves real-world sampling-based queries from Conviva by more than 50% on average across various sample sizes and machine learning workloads by up to 43%.
- When replaying the Facebook trace, on an EC2 cluster, KMN can improve job completion time by 81% on average (92% for small jobs)
- By using 5% – 10% extra tasks we can balance bottleneck link usage and decrease shuffle times by 61% – 65% even for jobs with high cross-rack skew.

6.1 Setup

Cluster Setup:We run all our experiments using 100 m2.4xlarge machines on Amazon's EC2 cluster, with each machine having 8 cores, 68GB of memory and 2 local drives. We configure Spark to use 4 slots and 60 GB per machine. To study memory locality we cache the input dataset before starting each experiment. We compare KMN with a baseline that operates on a pre-selected sample of size K and does not employ any of the shuffle improvement techniques described in §4, §5. We also label the fraction of extra tasks run (*i.e.,* M/K), so KMN-$M/K = 1.0$ has $K = M$ and KMN-$M/K = 1.05$ has 5% extra tasks. Finally, all experiments were run at least three times and we plot median values across runs and use error bars to show minimum and maximum values.

Workload: Our evaluation uses a workload trace from Facebook's Hadoop cluster [21]. The traces are from a mix of interactive and batch jobs and capture over half a million jobs on a 3500 node cluster. We use a scaled down version of the trace to fit within our cluster and use the same inter-arrival times and the task-to-rack mapping

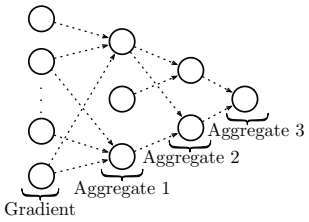


Figure 12: **Execution DAG for Stochastic Gradient Descent (SGD).**

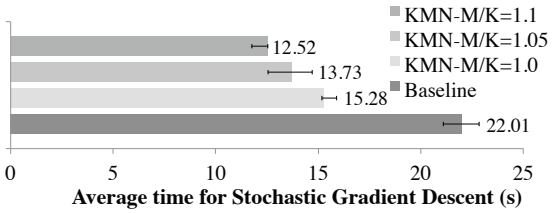


Figure 13: **Overall improvement when running Stochastic Gradient Descent using KMN**

as in the trace. Unless specified, we use 10% sampling when running KMN for all jobs in the trace.

We begin by showing overall gains with KMN (§6.2), then present benefits for input stages from KMN (§6.3) and finally show how KMN affects intermediate stages (§6.4).

6.2 Benefits of KMN

We evaluate the benefits of using KMN on three workloads: real-world approximate queries from Conviva, a machine learning workload running Stochastic Gradient Descent and a Hadoop workload trace from Facebook.

6.2.1 Conviva Sampling jobs

We first present results from running 4 real-world sampling queries obtained from Conviva, a video analytics company. The queries were run on access logs obtained across a 5-day interval. We treat the entire data set as N blocks and vary the sampling fraction (K/N) to be 1%, 5% and 10%. We run the queries at 50% cluster utilization and run each query multiple times.

Figure 11 shows the median time taken for each query and we compare KMN-$M/K = 1.05$ to the baseline that uses pre-selected samples. For query 1 and query 2 we can see that KMN gives 77%–91% win across 1%, 5% and 10% samples. Both these queries calculate summary statistics across a time window and most of the computation is performed in the map stage. For these queries KMN ensures that we get memory locality and this results in significant improvements. For queries 3 and 4, we see around 70% improvement for 1% samples, and this reduces to around 25% for 10% sampling. Both

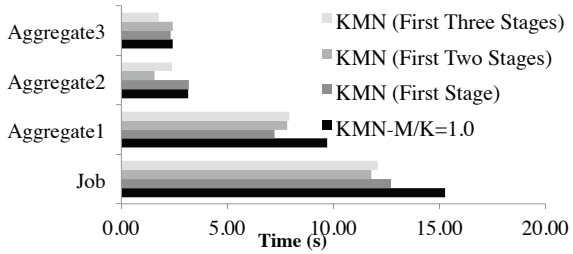


Figure 14: **Breakdown of aggregation times when using KMN for different number of stages in SGD**

Job Size	% Overall	% Map Stage	% Shuffle
1 to 10	92.8	95.5	84.61
11 to 100	78	94.1	28.63
> 100	60.8	95.4	31.02

Table 1: **Improvements over baseline, by job size and stage**

these queries compute the number of distinct users that match a specified criteria. While input locality also improves these queries, for larger samples the reduce tasks are CPU bound (while they aggregate values).

6.2.2 Machine learning workload

Next, we look at performance benefits for a machine learning workload that uses sampling. For our analysis, we use Stochastic Gradient Descent (SGD). SGD is an iterative method that scales to large datasets and is widely used in applications such as machine translation and image classification. We run SGD on a dataset containing 2 million data items, where each each item contains 4000 features. The complete dataset is around 64GB in size and each of our iterations operates on a 1% sample 1% of the data. Thus the random sampling step reduces the cost of gradient computation by 100× but maintains rapid learning rates [52]. We run 10 iterations in each setting to measure the total time taken for SGD.

Each iteration consists of a DAG comprised of a map stage where the gradient is computed on sampled data items and the gradient is then aggregated from all points. The aggregation step can be efficiently performed by using an aggregation tree as shown in Figure 12. We implement the aggregation tree using a set of shuffle stages and use KMN to run extra tasks at each of these aggregation stages.

The overall benefits from using KMN are shown in Figure 13. We see that KMN-$M/K = 1.1$ improves performance by 43% as compared to the baseline. These improvements come from a combination of improving memory locality for the first stage and by improving shuffle performance for the aggregation stages. We further break down the improvements by studying the effects of KMN at every stage in Figure 14.

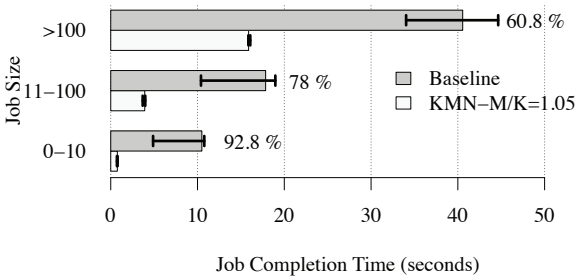


Figure 15: **Overall improvement from KMN compared to baseline. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.**

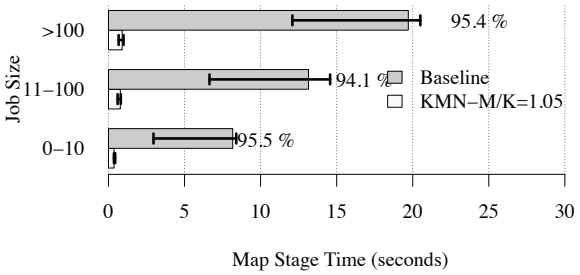


Figure 16: **Improvement due to memory locality for the Map Stage for the Facebook trace. Numbers on the bar represent percentage improvement using KMN-$M/K = 1.05$.**

When running extra tasks for only the first stage (gradient stage), we see improvements of around 26% for the first aggregation (Aggregate 1); see KMN (First Stage). Without extra tasks the next two aggregation stages (Aggregate 2 and Aggregate 3) behave similar to KMN-$M/K = 1.0$. When extra tasks are spawned for later stages too, benefits propagate and we see 50% improvement in the second aggregation (Aggregate-2) while using KMN for the first two stages. However, propagating choice across stages does impose some overheads. Thus even though we see that KMN (First Three Stages) improves the performance of the last aggregation stage (Aggregate 3), running extra tasks slows down the overall job completion time (Job). This is because the final aggregation steps usually have fewer tasks with smaller amounts of data, which makes running extra tasks not worth the overhead. We plan to investigate techniques to estimate this trade-off and automatically determine which stages to use KMN for in the future.

6.2.3 Facebook workload

We next quantify the overall improvements across the trace from using KMN. To do this, we use a baseline configuration that mimics task locality from the original trace while using pre-selected samples. We compare this to KMN-$M/K = 1.05$ that uses 5% extra tasks and a round-robin reducer placement strategy (§5.2.2). The results showing average job completion time broken down by job size is shown in Figure 15 and relative improvements are shown in Table 1. As seen in the figure, using KMN leads to around 92% improvement for small jobs with < 10 tasks and more than 60% improvement for all other jobs. Across all jobs KMN-$M/K = 1.05$ improves performance by 81%, which is 93% of the potential win (§2.3).

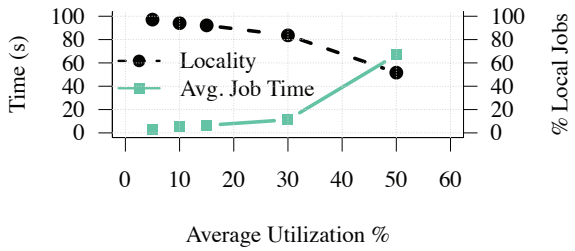


Figure 17: **Job completion time and locality as we increase utilization.**

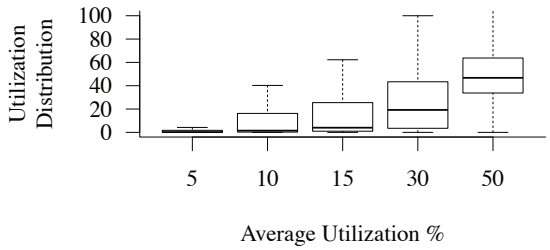


Figure 18: **Boxplot showing utilization distribution for different values of average utilization.**

To quantify where we get improvements from, we break down the time taken by different stages of a job. Improvements for the input stage or the map stage are shown in Figure 16. We can see that using KMN we are able to get memory locality for almost all the jobs and this results in around 94% improvement in the time taken for the map stage. This is consistent with the predictions from our model in §3 and shows that pushing down sampling to the run-time can give tremendous benefits. The improvements in the shuffle stage are shown in Figure 19. For small jobs with < 10 tasks we get around 85% improvement and these are primarily because we co-locate the mappers and reducers for small jobs and thus avoid network transfer overheads. For large jobs with > 100 tasks we see around 30% improvement due to reduction in cross-rack skew.

6.3 Input Stage Locality

Next, we attempt to measure how the locality obtained by KMN changes with cluster utilization. As we vary the cluster utilization, we measure the average job completion time and fraction of jobs where all tasks get locality. The results shown in Figure 17 show that for up to 30% average utilization, KMN ensures that more than 80% of

Job Size	$M/K = 1.0$	M/K =1.05	M/K =1.1
1 to 10	85.04	84.61	83.76
11 to 100	27.5	28.63	28.18
> 100	14.44	31.02	36.35

Table 2: **Shuffle time improvements over baseline while varying M/K**

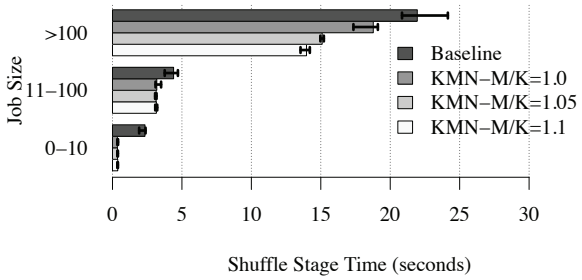


Figure 19: **Shuffle improvements when running extra tasks.**

Cross-rack skew	M/K=1.0	M/K =1.05	M/K =1.1
≤ 4	24.45	29.22	30.81
4 to 8	15.26	27.60	33.92
≥ 8	48.31	61.82	65.82

Table 3: **Shuffle improvements with respect to baseline as cross-rack skew increases.**

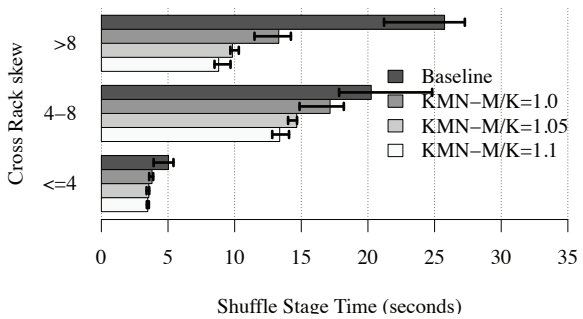


Figure 20: **Difference in shuffle performance as cross-rack skew increases**

jobs get perfect locality. We also observed significant variance in the utilization during the trace replay and the distribution of utilization values is shown as a boxplot in Figure 18. From this figure we can see that while average utilization is 30% we observe utilization spikes of up to 90%. Because of such utilization spikes, we see periods of time where all jobs do not get locality.

Finally, at 50% average utilization (utilization spikes $> 90\%$) only around 45% of jobs get locality. This is lower than predictions from our model in §3. There are two reasons for this difference: First, our experimental cluster has only 400 slots and as we do 10% sampling ($K/N = 0.1$), the setup doesn't have enough choices for jobs with > 40 map tasks. Further the utilization spikes also are not taken into account by the model and jobs which arrive during a spike do not get locality.

6.4 Intermediate Stage Scheduling

In this section we evaluate scheduling decisions by KMN for intermediate stages. First we look at the benefits from running additional map tasks and then evaluate the delay heuristic used for straggler mitigation. Finally we also measure KMN's sensitivity to reducer placement strategies.

6.4.1 Effect of varying M/K

We evaluate the effect of running extra map tasks (i.e $M/K > 1.0$) and measure how that influences the time taken for shuffle operations. For this experiment we wait until all the map tasks have finished and then calculate the best reducer placement and choose the best K map outputs as per techniques described in §4.2. The average time for the shuffle stage for different job sizes is shown in Figure 19 and the improvements with respect to the baseline are shown in Table 2. From the figure, we see that for small jobs with less than 10 tasks there is almost no improvement from running extra tasks as they usually do not suffer from cross-rack skew. However for large jobs with more than 100 tasks, we now get up to 36% improvement in shuffle time over the baseline.

Further, we can also analyze how the benefits are sensitive to the cross-rack skew. We plot the average shuffle time split by cross-rack skew in Figure 20. Correspondingly we list the improvements over the baseline in Table 3. We can see that for jobs which have low cross-rack skew, we get up to 33% improvement when using KMN-$M/K = 1.1$. Further, for jobs which have cross-rack skew > 8, we get up to 65% improvement in shuffle times and a 17% improvement over $M/K = 1$.

6.4.2 Delayed stage launch

We next study the impact of stragglers and the effect of using the delayed stage launch heuristic from §4.3. We run the Facebook workload at 30% cluster utilization with KMN-$M/K = 1.1$ and compare our heuristic to two baseline strategies. In one case we wait for the first K map tasks to finish before starting the shuffle while in the other case we wait for all M tasks for finish. The performance break down for each stage is shown in Figure 21. From the figure we see that for small jobs (< 10 tasks) which don't suffer from cross-rack skew, KMN performs similar to picking the first K map outputs. This is because in this case stragglers dominate the shuffle wins possible from using extra tasks. For larger tasks we see that our heuristic can dynamically adjust the stage delay to ensure we avoid stragglers while getting the benefits of balanced

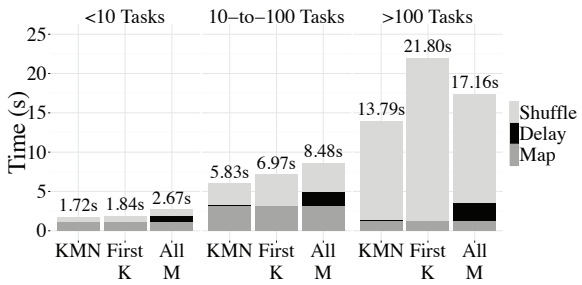


Figure 21: **Benefits from straggler mitigation and delayed stage launch.**

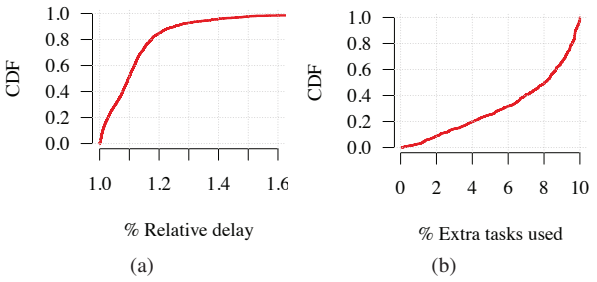


Figure 22: **(a) CDF of % time that the job was delayed (b) CDF of % of extra map tasks used.**

shuffle operations. For example for jobs with > 10 tasks KMN adds $5\% - 14\%$ delay after first K tasks complete and still gets most of the shuffle benefits. Overall, this results in an improvement of up to 35%.

For more fine-grained analysis we also ran an event-driven simulation that uses task completion times from the same Facebook trace. The CDF of extra map tasks used is shown in Figure 22(b), where we see that around 80% of the jobs wait for 5% or more map tasks. We also measured the time relative to when the first K map tasks finished and to normalize the delay across jobs we compute the relative wait time. Figure 22(a) shows the CDF of relative wait times and we see that the delay is less than 25% for 62% of the jobs. The simulation results again show that our relative delay is not very long and that job completion time can be improved when we use extra tasks available within a short delay.

6.4.3 Sensitivity to reducer placement

To evaluate the importance of reduce placement strategy, we compare the time taken for the shuffle stage for the round-robin strategy described in §5.2.2 against a greedy assignment strategy that attempts to pack reducers into as few machines as possible. Note that the baseline used in our earlier experiments used a random reducer assignment policy and §6.2.3 compares the round-robin strategy to random assignment. Figure 23 shows the results

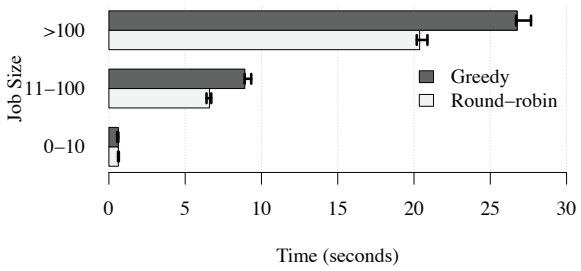


Figure 23: **Difference between using greedy assignment of reducers versus using a round-robin scheme to place reducers among racks with upstream tasks.**

from this experiment with the results broken down by job size. From the results we can see that for jobs with > 10 tasks using a round-robin placement can improve performance by 10%-30%. However for very small jobs, running tasks on more machines increases the variance and the greedy assignment in fact performs 8% better.

7 Related Work

Cluster schedulers: Cluster scheduling has been an area of active research and recent work has proposed techniques to enforce fairness [32, 39], satisfy job constraints [33] and improve locality [39, 59]. In KMN, we focus on applications that have input choices and propose techniques to exploit the available flexibility while scheduling tasks. Straggler mitigation solutions launch extra copies of tasks to mitigate the impact of slow running tasks [10, 12, 61]. While KMN shares the similarity of executing extra copies, our goals are different. Further, straggler mitigation solutions are limited by the number of replicas of the input data, and can leverage our observation of combinatorial choices towards more effective speculation. Prior efforts in improving shuffle performance [7, 24] have looked at either provisioning the network better or scheduling flows to improve performance. On the other hand, in KMN we use additional tasks and better placement techniques to balance data transfers across racks. Finally, recent work [49] has also looked at using the power of many choices to reduce scheduling latency. In KMN we exploit the power choices to improve network balance using just a few additional tasks.

Approximate Query Processing Systems: Approximate query processing (AQP) systems such as Aqua [2], STREAM [47], and BlinkDB [5] use pre-computed samples to answer queries. These works are complimentary to our work, and we expect that projects like BlinkDB can use KMN to improve performance, while maintaining, or in some cases even improving response quality. Prior work in databases has also proposed Online Aggregation [37] (OLA) methods that can be used to

present approximate aggregation results while the input data is processed in a streaming fashion. Recent extensions [25, 51] have also looked at supporting OLA-style computations in MapReduce. In contrast, KMN can be used for scheduling sampling applications which do not process the entire dataset and process a fixed and small sample of data.

Machine learning frameworks: Recently, a large body of work has focused on building cluster computing frameworks that support machine learning tasks. Examples include GraphLab [34, 44], Spark [60], DistBelief [27], and MLBase [41]. Of these, GraphLab and Spark add support for abstractions commonly used in machine learning. Neither of these frameworks provide any explicit system support for sampling. For instance, while Spark provides a sampling operator, this operation is carried out entirely in application logic, and the Spark scheduler is oblivious to the use of sampling.

8 Conclusion

The rapid growth of data stored in clusters, increasing demand for interactive analysis, and machine learning workloads have made it inevitable that applications will operate on subsets of data. It is therefore imperative that schedulers for cluster computing frameworks exploit the available choices to improve performance. As a first step towards this goal we have presented KMN, a system that improves data-aware scheduling for jobs with combinatorial choices. Using our prototype implementation, we have shown that KMN can improve performance by increasing locality and balancing intermediate data transfers.

Acknowledgments

We are indebted to Ali Ghodsi, Kay Ousterhout, Colin Scott, Peter Bailis, the various reviewers and our shepherd Yuanyuan Zhou for their insightful comments and suggestions. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adobe, Apple, Inc., Bosch, C3Energy, Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, and Yahoo!.

References

[1] Why is Thread.stop deprecated. `http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html`.

[2] S. Acharya, P. Gibbons, and V. Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the International Conference on Very Large Data Bases*, pages 754–757, 1999.

[3] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: interactive queries on very large data. *Proceedings of the VLDB Endowment*, 5(12):1902–1905, 2012.

[4] S. Agarwal, H. Milner, A. Kleiner, A. Talwarkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing When Youre Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*. ACM, 2014.

[5] 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.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM 2008*, Seattle, WA.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[8] 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 *Proceedings of the European conference on Computer systems (Eurosys'11)*, pages 287–300. ACM, 2011.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 12–12. USENIX Association, 2011.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

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

[13] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010.

[14] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[15] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *Proceedings of ACM SIGCOMM 2012*, pages 431–442. ACM, 2012.

[16] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer.

[17] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. NIPS Foundation, 2008.

[18] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, Prague, Czech Republic, June 2007.

[19] M. Cafarella, E. Chang, A. Fikes, A. Halevy, W. Hsieh, A. Lerner, J. Madhavan, and S. Muthukrishnan. Data management projects at Google. *SIGMOD Record*, 37(1):34–38, Mar. 2008.

[20] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2004.

[21] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[22] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of ACM SIGCOMM 2013*, pages 231–242, Hong Kong, China, 2013.

[23] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[24] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 98–109, Toronto, Ontario, Canada, 2011. ACM.

[25] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[26] G. Cornuejols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. Technical report, DTIC Document, 1983.

[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1232–1240, 2012.

[28] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS, 2014.

[29] Facebook Presto. Retrieved 9/21/2013, URL: `http://gigaom.com/2013/06/06/facebook-unveils-presto-engine-for-querying-250-pb-data-warehouse/`.

[30] T. Feder and D. Greene. Optimal algorithms for approximate clustering. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 434–444, Chicago, Illinois, USA, 1988. ACM.

[31] J. Gantz and D. Reinsel. Digital universe study: Extracting value from chaos, 2011.

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

[33] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th European conference on Computer Systems*. ACM, 2013.

[34] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation*, 2012.

[35] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM 2009*, Barcelona, Spain.

[36] B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou. Wave computing in the cloud. In *HotOS*, 2009.

[37] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM, 1997.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Proc. of the 2nd European Conference on Computer Systems*, 41(3), 2007.

[39] 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*, 2009.

[40] P. Kolari, A. Java, T. Finin, T. Oates, and A. Joshi. Detecting spam blogs: A machine learning approach. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1351, 2006.

[41] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A distributed machine-learning system. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[42] J. Langford. The Ideal Large-Scale Machine Learning Class. `http://hunch.net/?p=1729`.

[43] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1028–1039, 2012.

[44] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 2012.

[45] J. McCalpin. STREAM update for Intel Xeon Phi SE10P. `http://www.cs.virginia.edu/stream/stream_mail/2013/0015.html`.

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

[47] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[48] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. HotOS, 2013.

[49] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[50] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 2013.

[51] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for Large Mapreduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.

[52] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[53] N. Schraudolph, J. Yu, and S. Günter. A stochastic quasi-newton method for online convex optimization. *Journal of Machine Learning Research*, 2:428–435, 2007.

[54] L. Sidirourgos, M. Kersten, and P. Boncz. Sciborq: Scientific Data Management with Bounds on Runtime and Quality. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[55] P. Sukhatme and B. Sukhatme. *Sampling theory of surveys: with applications*. Asia Publishing House, 1970.

[56] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-Out Networking in the Data Center. *IEEE Micro*, 30(4):29–41, July 2010.

[57] J. Vygen. Approximation algorithms facility location problems. Technical Report 05950, Research Institute for Discrete Mathematics, University of Bonn, 2005.

[58] 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.

[59] 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*, pages 265–278, 2010.

[60] 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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[61] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pages 29–42, 2008.