A Comprehensive Review of Speculation for Hadoop Ecosystem

Ahmed Hussein
Jon Eagles, Nathan Roberts, Kuhu Shukla
{ahussein,jeagles,nroberts,kshukla}@verizonmedia.com
Verizon Media Big Data Organization: Hadoop

ABSTRACT

Hadoop is a collection of open source programs and utilities that provide a framework for distributed storage and processing of big data using the MapReduce programming model.

Hadoop aims at reducing the completion time of each individual job by splitting it into tasks that run in parallel. It is evident that such an execution model is sensitive to slow tasks, dubbed "*stragglers*", which impede the overall execution time of a job. This leads to a degradation of resource allocations; hence the loss of value of data and revenue. To reduce the impact of stragglers, Hadoop deploys a submodule, "*Speculator*", that periodically detects a straggler and launches a new "*speculative*" task that competes with the straggler to reach completion. The speculator relies on predefined thresholds to identify tasks that are exhibiting low progression compared to the average successful tasks.

In this paper, we evaluate the current Hadoop Speculator and the success rate of speculative tasks. Additionally, we present an adaptive task runtime estimator that overcomes some of the shortcomings in Hadoop speculator. Finally, we propose a roadmap toward unlocking full potential of dynamic detection and speculation in Hadoop.

1 INTRODUCTION

Verizon Media Group's Big Data stack powers the most demanding *Big Data* applications in the industry. These applications run on some 50K nodes and several N-thousand node clusters, making it one of the largest Hadoop clusters ever built. The robust infrastructure brings scalable computing to a whole new level. Verizon Media Group pioneered this level of scale with Hadoop ecosystem and continues to be a key leader in this space. The building blocks of the distributed system are constituted of various products built atop of Hadoop.

Based on the MapReduce programming model, Hadoop is a framework that simplifies writing applications to process large amounts of data [5, 12]. Hadoop splits the input data into smaller sub-problems that can run in parallel on large clusters. The simplicity to write complex computation and tolerance to failure played an important role in the rise of cloud systems. Hadoop hides fault-tolerance from the programmer by rerunning the tasks of a crashed node on a different machine. Not only crashed nodes, if a node performs poorly, Hadoop runs a speculative task on another machine to finish the computation faster.

Hadoop sets as the core of Verizon infrastructure providing the Distributed File System (HDFS) to store large files across multiple machines. On top of HDFS, *Spark* provides analytics engine for big data and machine learning with fast in-memory approaches that are easy to write. *Tez* also builds atop of Hadoop-Yarn [11, 12] to build high performance batch and interactive data processing application. *Hive* puts SQL on Hadoop making it easy to summarize, query, and analyze large sets of structured data. Similar to Spark, *Presto* is another distributed SQL query engine for running interactive analytic queries against data sources of all sizes. Finally, *HBase* which is a NoSQL database, a distributed, scalable, big data store supporting random read and writes.

1.1 The Straggler Problem

A MapReduce job is not completed until map and reduce tasks are all done. Slow running tasks (*stragglers*) can have significant impact in deteriorating the execution time of the job. The straggler phenomenon became the norm in large scale distributed systems. The straggler tasks add a tailing shape that exceeds 10 times longer compared to average task duration [4, 10]. According to Ouyang et al. [10], many production systems suffer from the fact that approximately 5% of tasks exhibit straggler behavior, with the longest execution time being 10 times slower. Notice that straggler tasks impact is not limited to the job lifetime. However, due to resource allocations, all queued jobs are affected in a cascading effect [6]. Therefore, it is important to detect stragglers early and reschedule them.

There are several key challenges in that process:

- How to detect stragglers?
- How to decide the priority of scheduling the speculative tasks since they consume resources on their own?
- Which node should should run the speculative task?

Since stragglers are common events on every node and they are not limited to hardware or networking problems,

it is not feasible to *blacklist* all nodes that host stragglers. Several works have attempted to address stragglers problem by speculative execution starting like Longest Approximate Time to End (LATE) [14], Mantri [2], and Dolly [1].

1.2 Static Speculation

LATE takes into account node heterogeneity while deciding where to speculate tasks. This has addressed many of the shortcomings in the Hadoop scheduler that assumed that tasks within the same classification perform the same measure of work.

Hadoop follows LATE, collecting the progress of each task. The progress is used to calculate the completion time for tasks to predict the stragglers. A straggler is identified if the estimated completion time (ECT) exceeds certain percentage compared to the average value within the same job.

Our evaluation revealed several issues with the speculative tasks:

- Early tasks have more chance to be tagged as a candidate to be speculated resulting in inefficient scheduling decisions.
- The assessment time needed by LATE is not proportional to the task lifetime leading to premature speculative decision for long running tasks.
- The estimated completion time is measured as the ratio between the progress rate and the execution time.
- long running tasks do not get speculated because the static threshold decides that it is too late to speculate
- LATE is sensitive to activity bursts. This is more evident in reduce tasks as we further elaborate in Section 2.1.

In this paper, we make the following contributions:

- Establish a survey as a base for a better understanding of the speculation process in Hadoop and Tez.
- Implement task runtime estimator based on simple exponential smoothing factor that showed to be more effective by predicting stragglers more accurately at 50% less cost to the job.
- Discuss the limitations and the roadmap toward improving dynamic speculation in Hadoop.

2 BACKGROUND

A straggler task occurs for multiple reasons [4]: (*i*) hardware, (*ii*) networking, (*iii*) resource contention, or (*iv*) skewed data input.

Hadoop has no built-in capabilities to analyze the bottlenecks. Instead, Hadoop detects slow tasks and launches speculative task that compete with the original task. When either task completes, it kills the other one. The process of running speculative tasks is an optimistic approach based on the intuition that a new speculative task will likely run on a different node that does not exhibit hardware failure or network congestion.

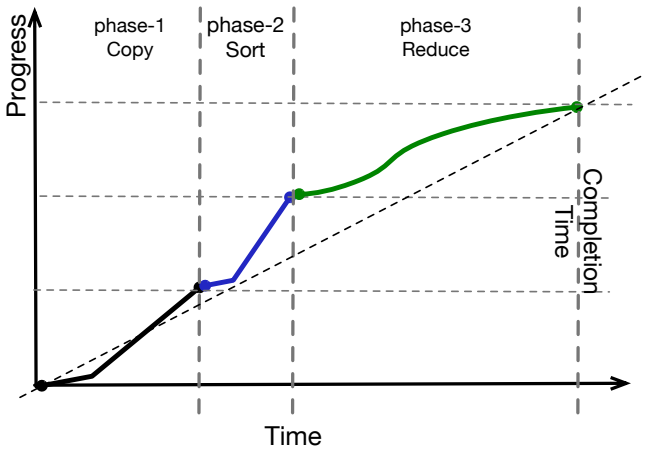

Figure 1: Relation between the progress rate of a reduce task (y-axis) in each different phase vs. the execution time (x-axis)

2.1 Task Progress Calculation

Hadoop makes several assumptions about the progress rate of a job including [14]: (*i*) cluster nodes are homogenous and perform at same rate, (*ii*) tasks within the same category (i.e., map or reduce) require the same amount of work, and (*iii*) tasks have constant progress rate.

Each task has "*progress score* (PS)" between 0 and 1. For map tasks, *PS* is the fraction of the input processed (*L*). For reduce tasks, the progress is divided equally between three phases: (*i*) *copy*, when map outputs are fetched; (*ii*) *sort*, when map outputs are sorted by key; and (*iii*) *reduce*, when reduce function is applied. Hence, *PS* in Equation (1) is calculated as the number of finished phases (*P*) and the fraction of processed input in the current phase. With each phase characterized by different workload (CPU intensity, I/O and memory), it is unlikely that copying and sorting invest the same time compared to applying the reduce function. Figure 1 shows how the progress rate of a reduce task typically changes throughout the execution time based on the workload of the relevant phase. Assuming equal weighting of different phases reduces the accuracy of the speculative decision as mentioned in Section 1.2.

$$PS = \begin{cases} L/R & \textit{for map tasks} \\ (P + \frac{L}{R}) * \frac{1}{3} & \textit{for reduce tasks} \end{cases} \quad (1)$$

Progress rate of each task is $\frac{PS}{T}$, where T is the amount of time the task has been running for, and then estimate the

time to completion. Note that this calculation is based on the assumption that all tasks progress at a constant rate.

The speculator looks at tasks that run for more than a minute and their estimated finishing time exceeds the average of their category (map/reduce). Based on LATE scheduler (Longest Approximate Time to End) [14], the task with farthest finish time into the future is labeled *straggler* and a new task is speculated. The intuition is that the straggler task provides greater opportunity for a speculative backup to overtake the original task.

2.2 Tez Speculation

Tez allows users to have more fine grained control of the data plan by modeling computation into a structure of a data processing workflow (Directed-Acyclic-Graph, or simply *DAG*) [11]. Each *vertex* represents a logical step of transforming and processing the data. In distributed processing, the work represented by a single vertex is physically executed as a set of *tasks* running on potentially multiple machines of the cluster. The direction of an *edge* represents movement of data between producers and consumers.

Tez deploys a similar technique to mitigate the impact of stragglers. Unlike MapReduce, Tez monitors task progress and tries to detect straggler tasks that may be running much slower than other tasks within the *same vertex* [11]. Once a vertex starts execution, it launches its own speculator service. The speculator scans only the tasks within its parent vertex looking for tasks with ECT exceeding the average completion time of the vertex's tasks. Upon detecting such a task, a speculative attempt may be launched that runs in parallel with the original task and races it to completion. If the speculative attempt finishes first then it is successful in improving the completion time.

This model has its own challenge in speculation as it adds another dimension to the task progress. It is unclear how Dependency between vertex is accounted for in the task progress calculation. This is very similar to the way reduce tasks in Hadoop are inaccurate due to the fetch phase. Since Tez DAGs are deeper than MapReduce, this effect is even more pronounced, and even has cascading effects.

In Tez, following a status update, each task scans for the straggler. We found out that the scanning loop was not thread safe causing redundancy and concurrency bugs [7, 8]. We have fixed this implementation to launch the speculator as a service for each vertex. The service runs a daemon that periodically scans vertex's tasks looking for stragglers. Once the vertex is completed, the service shuts down releasing all memory resources.

Another major difference between Tez and Hadoop is that Tez avoids scheduling the speculative task on the node running the original task. This improves the success rate of the speculative tasks as the new launched tasks avoid running on nodes that may be exhibiting some performance issues. We found that many speculative tasks share the same nodes as the original tasks which defies the purpose of speculative tasks. This clearly needs to be addressed in the future in MapReduce.

3 ADAPTIVE ESTIMATOR

In order to address the issues caused by using the average progress rate, we revisit the task runtime estimator to adjust to variation in progress rate throughout the execution time. The intuition is based on the fact that software applications exhibit different behavior throughout the runtime composed of a sequence of phases of execution. The change in execution pattern is not random; it falls into repeating patterns, called *phases*. For distributed systems, tasks exhibit execution phases requiring exhausting samples in order to characterize the progress rate.

Considering that PS of each task as a sequence of observations which are ordered in time. We evaluated different forecasting techniques such as moving averages and exponential smoothing to automatically adjust the ECT according to the progress rate of the task.

3.1 Exponential Smoothing

Simple moving average is a technique widely used to filter and reduce the effect of outliers. It averages past observations of data based on equal weights. The output generated requires the existence of at least k readings and takes only into account the last k readings. This works fine for homogenous systems that produce regular data in periodic intervals and does not exhibit stragglers or skewed data. *Exponential smoothing* is a technique that takes into considerations all past readings and smoothes time series data using the exponential window function assigning decreasing weights to previous data over time [3, 13]. Unlike the simple moving average, this technique does not require minimum number of observations to be made before producing an output.

The smoothed statistic s_t is the weighted average of the previous smoothed statistic s_{t-1} and the current reading x_t:

$$s_t = \alpha \cdot x_t + (1 - \alpha) \cdot s_{t-1} = s_{t-1} + \alpha \cdot (x_t - s_{t-1}) \quad (2)$$

where α is the smoothing factor $0 < \alpha < 1$. Larger values of α gives higher weight to recent changes in data which in turn reduces the smoothing level, eventually giving the full weight to the current reading with $\alpha = 1$. However, a desirable accuracy of the output will not be achieved until several readings have been sampled.

Picking the Forecast Parameters. The initial forecast is equal to the initial value of demand; however, this approach has a serious drawback. Exponential smoothing puts substantial

weight on past observations, so the initial value of demand will have an unreasonably large effect on early forecasts. This problem can be overcome by allowing the process to evolve for a reasonable number of periods (10 or more) and using the average of the demand during those periods as the initial forecast. However, it is important to note that the smaller the value of α, the more sensitive your forecast will be on the selection of this initial smoother value s_1 [3, 13].

The time constant of an exponential moving average is the amount of time for the smoothed response of a unit set function to reach 63.2% of the original signal.

$$\alpha = 1 - e^{\frac{-\Delta T}{\tau}} \tag{3}$$

Where ΔT is the sampling time interval of the discrete time implementation. If the sampling time is fast compared to the time constant ($\Delta T << \tau$) then $\alpha \approx \frac{\Delta T}{\tau}$.

The unknown parameters and the initial values for any exponential smoothing method can be estimated by minimizing the sum of squared errors (SSE). The errors are specified as $e_t = y_t - \hat{y}_{t|t-1}$ for $t = 1, .., T$. Hence we find the unknown parameters and the initial values that minimize.

$$SSE = \sum_{t=1}^{T}(y_t - \hat{y}_{t|t-1})^2 = \sum_{t=1}^{T} e_t^2 \tag{4}$$

3.2 Implementation

Running a job on MapReduce launches the speculator service that periodically scans the running tasks looking for stragglers. Note that each job has its own speculator that runs within the same VM. The frequency of the scanning depends on two main configuration parameters: `retry-after--speculate`, the waiting time to do run next cycle if there are tasks speculated in the current cycle; and `retry-after--no-speculate`, the waiting time to do run next cycle if there no task speculated in the current cycle. Only one task can be speculated in a given cycle. For each running job, the time constant can be tuned independently. In addition, the smooth factor is adjusted during runtime based on the data sampled in the first N readings of the tasks, where N is configurable parameter. This helps in adjusting the smoothing exponential calculation based on the job workload.

When a task attempt is completed, the statistics of the job are updated and a new average is re-calculated based on the completion time of all successful task attempts. Each task attempt periodically sends its PS. The PS is then incorporated to get the new forecast based on the previous readings. Calculating the progress rate to get an estimate of the next data reading is given in Algorithm 1. Finally, The ECT is calculated as a function of the forecast, progress, and the start time of the task as elaborated in Algorithm 2.

Algorithm 1: Forecasting the progress rate using the current task attempt PS

INPUT:
task attempt ID: aID
new progress: newP
current time stamp: newT
OUTPUT: estimate record

```
1:  procedure IncorporateReading
2:      est ← estimates.get(aID)
3:      if (est = NULL) then
4:          est ← new Estimate(-1, 0, MIN_VALUE)
5:          estimates.putIfAbsent(aID, est)
6:          return IncorporateReading(aID, newP, newT)
7:      if (est.atTime > newT || est.data > newP) then
8:          return est
                        /* calculate forecast Equations (2) and (3) */
9:      rate ← (newP − est.data)/(newT − est.atTime)
10:     exp ← (newT − est.atTime)/λ
11:     smoothF ← 1 − e^exp
12:     forecast ← smoothF ∗ rate + (1.0 − smoothF) ∗ est.forecast
13:     est.forcecast ← forecast
14:     est.data ← newP
15:     est.atTime ← newT
16:     est.index ← est.index + 1
17:     return est
```

Algorithm 2: Calculating ECT for Task Attempt

INPUT:
task attempt: taskAt
OUTPUT: estimate runtime to completion

```
1:  procedure estimatedRuntime
2:      rec ← estimators.get(taskAt.ID)
3:      if (rec = NULL) then
4:          return −1                    /* No information available */
5:      remWork ← Math.min(1.0, 1.0 − rec.data)
6:      forecast ← rec.forecast
7:      if (forecast <= 0.0) then
8:          return −1                    /* No information available */
9:      remTime ← (remWork/forecast)
10:     estTime ← remTime + rec.atTime − taskAt.startTime
11:     return estTime
```

Note that we use an index to mark the age of the estimate record (Algorithm 1, line 16). This helps in discarding estimates before we sample enough number of records for each task attempt. Otherwise, we find that the output of the runtime estimator is inaccurate for the first few readings causing inaccurate speculative decisions for task attempts.

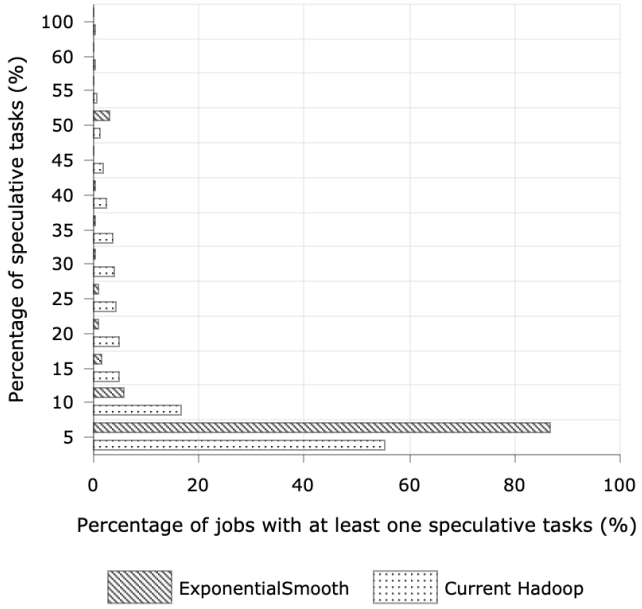

Figure 2: Stragglers statistics on Grid using two different speculators: speculative tasks as percentage of job's tasks (y-axis) vs percentage of jobs with at least one speculative task (x-axis)

3.3 Evaluation

Figure 2 shows the straggler percentage within jobs running on the Grid with the current Hadoop. Up to 55% of jobs launched 5% of speculative tasks while the maximum percentage recorded is 85% for a couple of jobs. The average success rate, in which a speculative task beats the original task is just 9%. In other words, more than 90% of speculative tasks added to the cost of computation and storage without speeding up the overall execution time. After deploying our new speculator "*Exponential Smooth*", the number of speculative jobs dropped to 50% saving more resources. Furthermore, the percentage of speculative tasks per job has dropped significantly (more than 85% of jobs launched at most 5% speculative tasks).

The new exponentially smooth estimator was more effective by protecting job latencies from stragglers by predicting stragglers more accurately and at a lower cost to the job.

3.4 Limitations

Although the time constant is configurable, finding a value that keeps the speculation decision under control is not a trivial task. In worst case when the estimator becomes responsive to recent reading, the platform will over speculate toward the end of execution. This is due to the fact that the nodes become available to execute speculative tasks. Tuning frequent jobs has huge benefit on the long run since a generic configuration will never be optimum for all jobs. Therefore, it is worth investing the effort in tuning the constants for such daily jobs until the speculation meets the target.

While the exponential smoothing forecast adapts to to change of progress rate throughout execution, the time constant does not. Once the parameters are initially set, all estimates are going to use those values until the job is completed. While this is how an estimator should work, it remains a research question whether the calculation should be dynamically adjusted according to the progress scores [10].

4 DISCUSSIONS

In this paper we made an implicit assumptions that reduce tasks are not scheduled before all map tasks finish execution. This forces a barrier separating the two phases making sure that the reducer has all the relevant input ready before it starts execution. This environment setting has the following implications on the speculator:

- the PS reported by a reducer is more accurate because it does not include idle time waiting for map tasks;
- the reducers are not scheduled leaving more resources to speculate map tasks identified as stragglers.

By removing the barrier, the PS of the reducers becomes inconsistent and flaky adding more complication to the estimation.

The PS function detailed in Section 2.1 depends solely on the number of the input records processed. This does not count for the actual workload performed by a single task. For instance, some initialization, finalization, or data processing may not involve a change in processed input. In that case, the task reports no progress, causing the estimator to speculate a backup task. Similarly, background services including runtime services like Garbage Collector (GC) compete on CPU, disk, and memory resources. Those runtime services cause burst of computation on each machine affecting the progress of the tasks running concurrently. For Java VM running on large heaps, a full GC can pause the application threads for several minutes. It is conceivable that the zero progress reported by the task is irrelevant from the task execution itself [4, 9].

For future work, we are considering different approaches to represent the PS as a function of workload rather than processed input. Such approaches require some reinforcement learning to characterize the running task and assign a weight based on the expected workload. In addition, we are investing into developing robust methodology to set a dynamic threshold to identify stragglers.

REFERENCES

[1] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 185–198. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=1924943.1924962

[3] Robert Goodell Brown. 1963. *Smoothing Forecasting and Prediction of Discrete Time Series*. Prentice-Hall.

[4] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[5] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[6] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. 2019. Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. *IEEE Transactions on Services Computing* 12, 1 (Jan 2019), 91–104. https://doi.org/10.1109/TSC.2016.2611578

[7] Tez Jira. 2018. *LegacySpeculator sometime issues wrong number of speculative attempts*. Retrieved July, 2019 from https://issues.apache.org/jira/browse/TEZ-3934

[8] Tez Jira. 2019. *Tez Speculation decision is calculated on each update by the dispatcher*. Retrieved July, 2019 from https://issues.apache.org/jira/browse/TEZ-4067

[9] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 349–365. http://dl.acm.org/citation.cfm?id=3026877.3026905

[10] Xue Ouyang, Peter Garraghan, Bernhard Primas, David Mckee, Paul Townend, and Jie Xu. 2018. Adaptive Speculation for Efficient Internetware Application Execution in Clouds. *ACM Trans. Internet Technol.* 18, 2, Article 15 (Jan. 2018), 22 pages. https://doi.org/10.1145/3093896

[11] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1357–1369. https://doi.org/10.1145/2723372.2742790

[12] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633

[13] Wikipedia. 2019. *Exponential smoothing*. Retrieved July, 2019 from https://en.wikipedia.org/wiki/Exponential_smoothing

[14] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744