Towards Optimizing Hadoop Provisioning in the Cloud

Karthik Kambatla*, Abhinav Pathak*, Himabindu Pucha‡
*Purdue University, ‡IBM Research Almaden

Abstract

Data analytics is becoming increasingly prominent in a variety of application areas ranging from extracting business intelligence to processing data from scientific studies. MapReduce programming paradigm lends itself well to these data-intensive analytics jobs, given its ability to scale-out and leverage several machines to parallely process data. In this work we argue that such MapReduce-based analytics are particularly synergistic with the pay-as-you-go model of a cloud platform. However, a key challenge facing end-users in this environment is the ability to provision MapReduce applications to minimize the incurred cost, while obtaining the best performance. This paper first motivates the importance of optimally provisioning a MapReduce job, and demonstrates that existing approaches can result in far from optimal provisioning. We then present a preliminary approach that improves MapReduce provisioning by analyzing and comparing resource consumption of the application at hand with a database of similar resource consumption signatures of other applications.

1 Introduction

The rising number of web applications serving millions of Internet users and dealing with petabytes of data, the advent of cheap storage capacity resulting in a tremendous growth in data retention, and the availability of cheap resources to process that data, have all reinstated the need for large-scale data processing. Extracting information and intelligence from these data sets, commonly referred to as **data analytics**, is an important data-intensive application stemming from these huge corpus of data. Data analytics is shown to be useful in several scenarios—analytics enable web data mining (e.g., web indexing and search), enable extracting business intelligence (e.g., click stream analysis for increasing ad revenue), enable processing data sets from scientific studies and simulations (e.g., research in natural language processing, seismic simulation, scene completion).

Analyzing these large volumes of data demands a highly scalable solution. MapReduce [6] is one popular approach that enables data analytics by parallely processing data, partitioned among large number of commodity machines. MapReduce can easily scale out to clusters of hundreds of commodity nodes as the data and processing demands scale, and can automatically handle failures in these large settings. Hadoop [1] is an open-source implementation of MapReduce, primarily supported by Yahoo, and also in use at Facebook, Amazon, Baidu etc.

This paper argues that data analytics enabled by MapReduce are particularly synergistic with the *utility computing* or pay-as-you-go model enabled by the cloud platform. An interesting artifact of this utility nature of the cloud paradigm is that the cost of using 1000 machines for 1 hour is the same as using 1 machine for 1000 hours. Thus, a MapReduce job can potentially improve its performance while incurring the same currency cost by acquiring several machines and executing in parallel. A physical cluster, on the other hand, places a hard upper bound on the number of machines available, and hence the achievable performance. Further, data analytics operations that are initiated periodically or sporadically as background batch jobs can allocate resources on-demand, and relinquish them after the job is done, thereby paying only for the used resources.

In fact, analytics enabled by the cloud platform was successfully leveraged on several occasions [3]. For example, 200 servers on Amazon EC2 (equivalent of 1407 hours of virtual machine time) were used by The Washington Post to convert 17481 pages of data in non-searchable PDF format of Hillary Clinton's official white house schedule into friendly WWW format within a total time of 9 hours. The New York Times converted 11 million scanned articles to PDFs using 100 virtual machines in a single day. A positive reinforcement for analytics in the cloud comes from the recent Amazon Web Services announcement to provide Elastic MapReduce [2] built using their compute cloud, EC2, and their storage cloud, S3.

Given the advantages of performing Hadoop jobs atop a cloud platform, a key challenge facing end-users is efficiently provisioning such Hadoop jobs. Provisioning a Hadoop job entails requesting optimum number of resource sets (a $resource\ set$ (RS) is a set of resources sold as a single unit. e.g., standard and high-cpu instances sold by Amazon web services), and configuring Hadoop parameters such that each resource set is maximally utilized. An optimization objective that we believe to be very relevant to end-users is to minimize currency cost incurred to perform a job, while maximizing performance (lower execution time). We argue that an end-user provisioning a Hadoop job should choose the resource set parameters to achieve this objective.

Existing solutions such as [2, 5, 4] simply automate the deployment of Hadoop tasks by setting up the required software, preloading data, and generating default configuration files. They, however, expect end-users to provide appropriate resource-set parameters. Even end-users have no tools to-date at their disposal to determine the optimum configuration, and hence these parameters

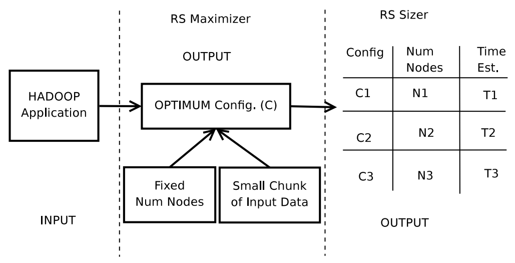

Figure 1: Overview of the optimization process to provision Hadoop jobs in the cloud.

are typically chosen using best practices. Moreover, we believe that existing approaches to provisioning other applications in the cloud are not immediately relevant to Hadoop-based applications; existing applications incorporate a dynamic component to adapt to workload changes and adjust their provisioning, while the workload from Hadoop applications could be known completely apriori.

This paper argues that simply automating Hadoop deployment is not good enough, and choosing parameters based on best practices is not robust enough. Our contributions are twofold: (1) We first demonstrate that best practices for configuring resource set parameters are not robust across different applications, and can cause far from optimal provisioning. (2) We then present our preliminary technique that improves provisioning of a hadoop job by taking resource consumption statistics of the job into account.

An overview of our approach to optimize provisioning of Hadoop jobs in the cloud is presented in Figure 1. Our approach consists of two components: (1) RS Maximizer: This component is responsible for calculating the optimum parameters for the Hadoop job such that each resource set is fully utilized. (2) RS Sizer: Once each resource set is being fully utilized, this component determines the number of resource sets required such that the cost is minimized while performance is maximized. Underutilized resource sets can potentially cause RS Sizer to pick larger number of resource sets, thereby increasing the incurred cost. Note that we focus on RS Maximizer for the scope of this paper, and present preliminary results. We acknowledge that RS Sizer is still a work in progress.

2 Hadoop Background

Hadoop [1] is an open source implementation of the MapReduce [6] programming model. A MapReduce job usually[1] consists of three phases—map, copy and reduce. The input data is split into chunks of 64MB size (by default). In the map phase, a user defined function operates on every chunk of input data producing intermediate key-value pairs which are stored on local disk. One map process is invoked to process one chunk of input data. In the copy phase, the intermediate key-value pairs are transferred to the location where a reduce process would operate on the intermediate data. In reduce phase, a user defined reduce function operates on the intermediate key-value pairs and generates the output. One reduce process is invoked to process a range of keys.

[1]A MapReduce task could potentially have several MapReduce stages.

Hadoop has over 180 configuration parameters. Examples include number of replicas of input data, number of parallel map/reduce tasks to run, number of parallel connections for transferring data etc. Of these several parameters, this paper specifically focuses on two that influence the resource utilization in a resource set.

mapred.tasktracker.map.tasks.maximum and *mapred.tasktracker.reduce.tasks.maximum* respectively set the maximum number of parallel mappers and reducers that can run on a Hadoop slave. Each map/reduce task runs as a separate process and hence a higher number for these parameters translates into higher parallelization. But too high a value can potentially cause resource contention and degrade overall performance. For example, setting a very high value for this parameter results in large number of simultaneous disk reads results in disk contention. Setting a low value, on the other hand, might under-utilize the resources, and once again reduce performance. Thus, the number of map and reduce tasks per resource set must be chosen such that the resources are maximally utilized, resulting in optimum performance.

3 RS Maximizer

3.1 Motivation: one size does not fit all!

This section demonstrates that static, default parameters cannot maximize resource set utilization across all applications, thereby motivating the need for the RS Maximizer component of our provisioning algorithm.

Hadoop installation comes with a default set of values for all the parameters in its configuration. The default values of these parameters are based on typical configuration of machines in clusters and requirements of a typical application. The optimum parameters that maximize resource utilization, however, are dependent on the resource consumption profile of an application. For example, a map task in `sort` application using Hadoop (implemented as merge-sort) reads each chunk of data, generates <key, value>, and outputs equal amount of intermediate data, making it quite io-intensive. On the other hand, a map task in `grep` needs to search for a regular expression which is limited by the CPU resources. Thus, each application has a different bottleneck resource (the resource with the highest utilization fraction compared to other required resources), and different bottleneck resource utilization, and thus needs to pick a different (num_maps, num_reduces) combination such that the bottleneck resource is maximally utilized. Note, however, that statically chosen large values for these parameters may cause resource contention and lower overall performance. Similarly, statically chosen small values may result in under-utilization of resources.

We now experimentally demonstrate the optimum (num_maps, num_reduces) for a given application and resource set, and the impact of sub-optimum configuration parameters. To measure the impact of these parameters on application performance, we ran two experiments for various combinations of (num_maps, num_reduces): (1) `grep` (example available with Hadoop source code) on Hadoop on a cluster of 8 nodes, and an input of 80 GB. Each node in the cluster consists of 8 cores 2.2 GHz CPUs and two SATA drives. All nodes are connected using a Gigabit switch. (2) `wordcount` on Hadoop on a cluster of 4 nodes, and an input of 1 GB. Each node consists of a 3 GHz dual core CPU, 2 GB RAM, 146 GB 10K SCSI drive, and a 1 Gbps NIC. All nodes are connected using a Gigabit switch. A replication factor of two was used in both experiments.

Figure 2 shows the time taken by `grep` to search for a simple regex string in 80 GB of randomly generated data as the number of maps and reduces are varied. Figure 2 shows that: (1) Time taken for `grep` varies with number of maps, but is independent of number of reduces, (2) The configuration with 8 maps yields the best performance and runs 4$\times$ faster when compared to the configuration with 1 map and roughly 1.5$\times$ faster when compared to configuration with 24 Maps. Performing regex on input stream of data is inherently computationally intensive and hence `grep` is CPU dependent. Since each node in the cluster has 8 cores, 8 maps potentially achieve close to optimum CPU utilization. (3) Finally, increasing number of maps from 4 to 8 only marginally improves the performance, whereas changing maps from 1 to 4 almost quadruples the performance. Here, we observe that the disk bandwidth begins to saturate before all the CPUs are fully utilized, giving rise to a small improvement from 4 to 8 maps. Increasing the number of maps further causes disk thrashing as a result of which the performance decreases. Similar results were observed in the 4 node cluster.

We also ran `wordcount` on Hadoop to count the number of occurrences of each word in a randomly generated input data of size 1 GB. The optimum configuration for `wordcount` is 2 maps and 4 reduces for the 4 node cluster (hard to explain). For both `grep` and `wordcount`, the default configuration of (2 maps, 2 reduces) consumed significantly more time than the optimum configurations.

In summary, our experiments demonstrate that: (1) Hadoop parameters (number of mappers and reducers) affect the utilization of resources for a given resource set, and hence the overall performance. Too few mappers/reducers per resource set result in under-utilization of resources. Too many will induce contention and lower system throughput. (2) Parameter values that fully utilize a bottleneck resource are shown to be optimum. (3) Since different applications have different bottleneck resources, their optimum parameter values depend on the size of resources in the resource set and it is non-trivial to find them.

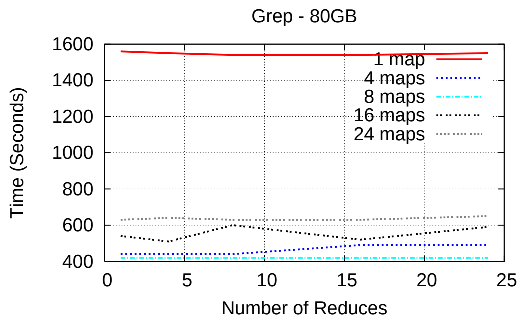

Figure 2: Time taken to `grep` a simple string from 80 GB of input data. Each line corresponds to a fixed number of maps.

3.2 Signature-based Approach: Design and Preliminary Evaluation

Our technique for determining the optimum configuration parameters for a new Hadoop application, given a resource set, is as follows: we first generate resource consumption signature (described later) for the new application by running the application on a small fraction of input data (few of the many chunks of the actual input), using a small number of resource sets (nodes). We then match the resource consumption signature with the signatures of other applications for which we have already computed the optimum configuration. We maintain a database of resource consumption signatures for a few applications for which we know the optimum configuration parameters. The new application is then assigned the optimum configuration of the application based on the closest signature match. Intuitively, two applications having similar resource consumptions would face similar bottlenecks and have similar ratio of resource requirements (CPU:Disk:Network). As a result they would exhibit optimal performance at similar configurations.

Figure 3 details our system. Hadoop applications usually operate on massive volumes of data split into chunks of 64 MB (by default). Our system takes entire input data and generates a copy of a small part (say 1GB) of it. It then runs the application on the smaller data on a subset of nodes. It monitors the usage of three basic resources on all the nodes in the cluster: CPU, disk and network. We use vmstat and ifstat tools to measure CPU usage (in terms of User, System, Idle and Wait percentages), disk usage (# of blocks in and out) and network usage (bytes/sec into and out of network card) every second. All the usage measurements are normalized using their respective maximum values.

Signature Generation: Usually, a Hadoop task consists of a map phase, a copy phase and a reduce phase [6]. Ideally, a resource consumption signature should be generated for each phase and be compared to signatures of the same phase from other applications, to independently determine num_maps, and num_reduces. Unfortunately such an approach has practical limitations: First, the three phases need not be disjoint in time. For example, a copy phase could begin before all the maps are completed. Second, an application may contain multiple MapReduce phases (`grep` for example). Our design, thus, favors phase agnostic signatures, generated as follows: We split the entire job run into n (a pre-choosen number) intervals with each interval having the same duration. For the i^{th} interval we compute the average resource consumption for each, r^{th}, resource. For each node, m, in the cluster we generate a resource consumption signature set, S^r_m, for every r^{th} resource as

$$S^r_m = \{S^r_{m1}, S^r_{m2}, ..., S^r_{mn}\}$$

where S^r_{mi} = mean of normalized r^{th} resource consumption for i^{th} interval on m^{th} node during the job run. We store all such signature sets for an application run.

Signature Comparison: To measure similarity between two normalized resource consumption signatures S^{r1}_m and S^{r2}_m for a particular resource r for a node m for Hadoop applications 1 and 2 respectively, a χ^2 [7] is calculated as:

$$\chi^2{}_{S^{r1}_m, S^{r2}_m} = \sum_{i=1}^{n} \frac{(S^{r1}_{mi} - S^{r2}_{mi})^2}{(S^{r1}_{mi} + S^{r2}_{mi})}$$

χ^2 represents the vector distance between two signatures for a particular resource r in time-interval vector space. We compute scalar addition of χ^2 for all the resource types [2]. Lower value of sum of χ^2 indicates more similar signatures. We choose the configuration of the application that has the closest signature distance sum to the new application.

Database Bootstrapping: To create a database of signatures, we generated signatures for a few applications with default configuration. For these applications, we also found out the optimum configuration (in terms of MapReduce numbers) by running the chosen applications at a variety of combinations of (num_maps, num_reduces). We store the signatures and the corresponding optimum configurations for these applications in our database. We chose the example applications (from the ones provided with Hadoop source)—`grep` (map intensive), `sort` (copy intensive), `wordcount`.

[2]Though a simple scalar addition for different resources is naive, we will investigate other methods to match signatures for each resource separately.

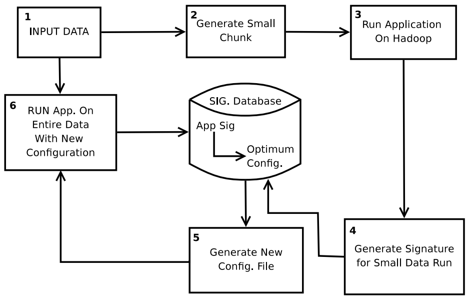
1 INPUT DATA
2 Generate Small Chunk
3 Run Application On Hadoop
6 RUN App. On Entire Data With New Configuration
SIG. Database
App Sig
Optimum Config.
5 Generate New Config. File
4 Generate Signature for Small Data Run

Figure 3: RS Maximizer: Block diagram

We also plan to incorporate signatures from `gridmix` with various ratios of input:intermediate size:output data.

3.3 Preliminary Results

Our technique is implemented by scripting in Perl to run the application over Hadoop as per our system diagram (Fig. 3). We start the measurement tools (vmstat and ifstat to record the cpu, disk and network usage of the Hadoop application) and run the application on small chunk of data (1 GB) using default configuration and then compute the signature for the application.

CPU and disk signatures for a given application are stable. To observe the noise in our signature mechanism we ran `sort` on 1 GB of data twice on our 8 node cluster with default configurations . We computed the signature distances between the generated signatures of these two runs. We plot the signature distances between these two runs for all the resource types (us, sy, wa, id, bi, bo, ni, no = % of CPU in user time, system time, waiting time, ideal time, disk block in, disk block out, network in and network out respectively) in Figure 4. We observe that noise (signature distance) is relatively higher in network resources than others.

Signatures are independent of input data size. We next plot signature distances for `sort` run on 1 GB data vs. `sort` run on 10 GB data with default configuration. Figure 4 plots the signature distances. We observe that different runs of the same application on different data sizes results in very close signatures. The signature distances between `sort` 1 GB vs. 10 GB are about 0.1.

Different applications have widely separated signatures if their bottleneck resources are different. Runs of different applications result in quite distant signatures if the applications' behaviors are different. `sort` and `grep` have very different requirements. `grep` consumes most of the time in the map phase (mostly CPU computation) where as `sort` consumes the maximum time in copy phase. This difference manifests in the signature distance—the signature distance is about 1 in most cases.

Signature based technique successfully predicts optimum configuration for different applications. For testing our system with a couple of unknown

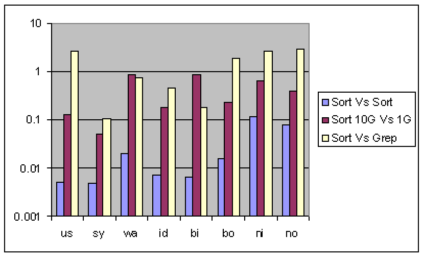

Figure 4: Signature Distances (on y-axis) for various resources on X-axis for*(i)* Sort 1G vs Sort 1G;*(ii)* Sort 1G vs Sort 10G;*(iii)* Sort 1G vs Grep 1G.

applications, we chose `matrix-addition` and `multifile-wordcount`. `matrix-addition` takes in two matrices; the map outputs (row, column) as the key and the value as the value; the reduce adds the values with the same key (row, column) and outputs the result. `multifile-wordcount` implements `wordcount` with the only difference that it takes input as multiple files instead of a single file. We ran these two applications with default (2 maps, 2 reduces) configuration on smaller data (1 GB) and compare the signatures.

Both applications closely matched the signature of `wordcount` from our database. Both applications obtain peak performance with the optimum configuration of `wordcount`. `Multifile-wordcount` is very similar to `wordcount` with just difference in data formats. It is therefore intuitive for them to have similar optimum configuration. Interestingly `sort`, `wordcount` and `multifile-wordcount` had near similar signatures and they observe peak performance with same configurations. This is because all the three applications have similar map phase (just read the input data and output intermediate key-value pairs with word count and `multifile-wordcount` appending a “1” for each word), and a similar reduce phase (with the difference that wordcount just adds up the number of occurrences of each word). `matrix-addition` exhibits similar characteristics as `wordcount`.

Improving Database Diversity: The accuracy of our signature match critically depends on the diversity of sample applications in our signature database. However, adding a new signatures into the database is a heavy-weight process; our technique relies on a brute-force approach to calculate the optimum configuration. We are currently investigating a closed loop approach that will improve both the accuracy of our configuration estimates, as well as add new samples to the database. For every new application run on the cluster with our technique, we predict the optimum configuration and run the application with optimum configuration (as noted above). Simultaneously, we also monitor the resource consumption during this run to identify a bottleneck resource(s) for the application, and ensure that it is saturated. If resource(s) are not saturated, we then search for a optimum configuration by intelligently increasing (num_maps, num_reduces) such that the resource with the highest utilization saturate. After this feedback-driven choice of optimum configuration, we add the new application signature to our database.

4 Discussion and Future Work

An interesting artifact of the cloud paradigm is that the cost of using 1000 machines for 1 hour, is the same as using 1 machine for 1000 hours. This observation implies that a Hadoop job’s performance can potentially be improved, while incurring the same cost, since Hadoop is built to exploit parallelism. Thus, a key goal of the RS Sizer is to leverage this observation while searching for the optimum size of resource sets. RS Sizer scales out a Hadoop job if the gain in performance due to parallelism outweighs the corresponding loss in performance from distributing the processing. We are currently exploring approaches to estimate both the loss and gain in performance of a Hadoop job. Intuitively, one downside of increasing the number of resource sets is the network bandwidth required to coordinate the map and the reduce phase (i.e., copy phase). We are currently exploring techniques to estimate this loss in performance as well as the overall scaling of execution time with increasing number of nodes to design appropriate RS sizing algorithms.

This work proposes an approach to minimize cost, and maximize performance in a cloud setting. We, however, argue that if our resources were not virtualized, our approach is still beneficial since maximizing the utilization of each resource set optimizes the performance of the Hadoop job at hand, and identifying unused resources can lead to potential energy savings.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback on this work; Prasenjit Sarkar, Prashant Pandey, and Karan Gupta for helpful discussions about Hadoop performance. This work was supported in part by the National Science Foundation under grant IIS-0844500.

References

[1] Hadoop. http://hadoop.apache.org.

[2] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[3] Amazon Web Services: Case Studies. http://aws.amazon.com/solutions/case-studies/.

[4] Cloudera. http://www.cloudera.com/.

[5] Hadoop on demand. http://hadoop.apache.org/core/docs/current/hod.html.

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

[7] R. O. Duda, P. E. Hart, and D. G. Stork. Chapter 8. pattern classication. 2nd edition. In *A Wiley-Interscience Publication*, 2001.