Towards Automatic Optimization of MapReduce Programs

Shivnath Babu*
Duke University
Durham, North Carolina, USA
shivnath@cs.duke.edu

ABSTRACT

Timely and cost-effective processing of large datasets has become a critical ingredient for the success of many academic, government, and industrial organizations. The combination of MapReduce frameworks and cloud computing is an attractive proposition for these organizations. However, even to run a single program in a MapReduce framework, a number of tuning parameters have to be set by users or system administrators. Users often run into performance problems because they don't know how to set these parameters, or because they don't even know that these parameters exist. With MapReduce being a relatively new technology, it is not easy to find qualified administrators. In this position paper, we make a case for techniques to automate the setting of tuning parameters for MapReduce programs. The objective is to provide good out-of-the-box performance for ad hoc MapReduce programs run on large datasets. This feature can go a long way towards improving the productivity of users who lack the skills to optimize programs themselves due to lack of familiarity with MapReduce or with the data being processed.

Categories and Subject Descriptors: K.6.4 [Management of Computing and Information Systems]: System Management

General Terms: Experimentation, Performance

Keywords: MapReduce, Hadoop, Cost-based Optimization

1. INTRODUCTION

We are in the "big data" era. Many enterprises continuously collect large datasets that record customer interactions, product sales, results from advertising campaigns on the Web, and other types of information. Facebook collects 15 TeraBytes of data each day into its PetaByte-scale data warehouse [17]. Powerful telescopes in astronomy, particle accelerators in physics, and genome sequencers in biology are putting massive volumes of data into the hands of scientists. The ability to perform scalable and timely analytical processing of these datasets to extract useful information is now a critical ingredient for success. For example, a number of products that appear on the Facebook site are based on large-scale data analysis. Ad hoc analysis and business-intelligence applications are used by data analysts throughout Facebook [17].

Cost-effective processing of large datasets is a nontrivial undertaking. Fortunately, MapReduce frameworks and cloud computing have made it easier than ever for academic, government, and industrial organizations to step into the world of big data. MapReduce frameworks, introduced in [6], provide a programming model and run-time system that are amenable to a variety of real-world tasks. For programs written in this model, the run-time system automatically parallelizes the processing across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and storage.

Hadoop [8] is an implementation of a MapReduce framework that follows the design laid out in the original paper. A number of enterprises use Hadoop in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, financial analysis, scientific simulation, and bioinformatics research. MapReduce frameworks are well suited to run on cloud computing platforms. Cloud-based services are now available that make it easy to set up and run MapReduce programs using Hadoop. *Amazon Elastic MapReduce* is a hosted MapReduce framework that runs dynamically-provisioned Hadoop clusters using Amazon Elastic Compute Cloud (EC2) and Simple Storage Service (S3).

```
// The map function defined in a class named Map
public void map(LongWritable key, Text value,
 OutputCollector<Text, DoubleWritable> output) {
   // We need to parse the input line of text
   //   to extract the product & sales information
   StringTokenizer tok =
      new StringTokenizer(value.toString(), "|");
   Text product = new Text();
   product.set(tok.nextToken());
   DoubleWritable sales = new DoubleWritable();
   sales.set(Double.parseDouble(tok.nextToken()));
   // Output extracted product & sales information
   //   to be processed by the reduce function
   output.collect(product, sales);
}
public static void main(String[] args) {
   JobConf conf = new JobConf();
   // Set some parameters explicitly
   conf.setOutputKeyClass(Text.class);
   conf.setOutputValueClass(DoubleWritable.class);
   conf.setMapperClass(Map.class);
   conf.setReducerClass(Reduce.class);
   conf.setNumReduceTasks(10);
   // Submit the job for execution
   JobClient.runJob(conf);
}
```

Figure 1: Example (edited) MapReduce program in Hadoop

*Supported by NSF grants IIS 0644106 and IIS 0917062

SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.

Parameter Name	Brief Description and Use	Default Value	Values Considered
mapred.reduce.tasks	Number of reducer tasks	1	[5,300]
io.sort.mb	Size in MegaBytes of map-side buffer for sorting key/value pairs	100	[100,200]
io.sort.record.percent	Fraction of io.sort.mb dedicated to metadata storage	0.05	[0.05,0.15]
io.sort.factor	Number of sorted streams to merge at once during sorting	10	[10,500]
io.file.buffer.size	Buffer size used to read/write (intermediate) sequence files	4K	32K
mapred.child.java.opts	Java control options for all mapper and reducer tasks	-Xmx200m	-Xmx[200m,300m]
mapred.job.shuffle.input.buffer.percent	% of reducer task's heap used to buffer map outputs during shuffle	0.7	0.7,0.8
mapred.job.shuffle.merge.percent	Usage threshold of mapred.job.shuffle.input.buffer.percent to trigger reduce-side merge in parallel with the copying of map outputs	0.66	0.66,0.8
mapred.inmem.merge.threshold	Another reduce-side trigger for in-memory merging; off when 0	1000	0
mapred.job.reduce.input.buffer.percent	% of reducer task's heap to buffer map outputs while applying reduce	0	0,0.8
dfs.replication	Block replication factor in Hadoop's HDFS filesystem	3	2
dfs.block.size	HDFS block size (data size processed per mapper task in our setting)	64MB	128MB

Table 1: A subset of job configuration parameters in Hadoop ($[a, b]$ indicates the range of values from a to b)

Running a MapReduce Program: A task is expressed in the MapReduce programming model through a *map* function and a *reduce* function. As an illustration, Figure 1 shows the map function from an example MapReduce program written to run in Hadoop.[1] This program groups the records in a large input text file by product name, and computes the total sum of sales per product. The map function shown in Figure 1 parses each line of text in the input to extract the corresponding product and sales information.

To run the program as a job in Hadoop, a *job configuration* object is created; and the *parameters* of the job are specified. The example Java main function in Figure 1 specifies values for the following parameters explicitly: (i) the respective classes implementing the map and reduce functions, (ii) the data types of the key and value output by the map function (the complete map output is grouped on the key and input to the reduce function), and (iii) the number of distinct reducer tasks to use to process the reduce function.

Apart from the job configuration parameters whose values are specified explicitly like in Figure 1, there are a large number of other parameters whose values have to be specified before the job can be run in a MapReduce framework like Hadoop. The settings of these parameters control various aspects of job behavior during execution such as memory allocation and usage, concurrency, I/O optimization, and network bandwidth usage. The submitter of a Hadoop job has the option to set these parameters either using a program-level interface like in Figure 1 or through XML configuration files. Higher-level languages for MapReduce frameworks like HiveQL and Pig have developed their own *hinting* syntax for parameter specification. For any parameter whose value is not specified explicitly during job submission, default values—either shipped along with the system or specified by the system administrator—are used.

More than 190 parameters are specified to control the behavior of a MapReduce job in Hadoop. As a conservative estimate, the settings of more than 25 of these parameters can have significant impact on job performance. (We will provide empirical evidence in Section 2.) A fairly large subset of these parameters display strong performance *interactions* with one or more other parameters. An interaction exists between parameters p_1 and p_2 when the magnitude of impact that varying p_1 has on job performance depends on the specific setting of p_2. Stated otherwise, the performance impact of varying p_1 is different across different settings of p_2.

Hadoop developers may well be repeating a mistake that relational database developers made in the past, which is one of the reasons why database systems have gained notoriety as systems that are hard to manage. When they were developed originally, relational database systems had a small fraction of the features found in commercial databases today. With the addition of more and more new features at a fast pace, database system developers followed a "tuning-knob-for-everything" philosophy [5]. Consequently, over time, relational database systems became bloated with features; many of them unnecessary for a vast majority of users. What was worse was that setting the tuning knobs became a nightmare for most users. In addition, the learning curve for new users became increasingly steep over time. Highly paid database administrators became essential to manage these systems. Browsing through the Hadoop, Hive, and Pig mailing lists reveals that users often run into performance problems caused by lack of knowledge of the configuration parameters. With MapReduce being a relatively new technology, it is not easy to find qualified administrators.

In this position paper, we make a case for techniques to automate the setting of job configuration parameters for MapReduce programs in general; focusing in particular on Hadoop. The objective is to provide good out-of-the-box performance for ad hoc MapReduce programs run on large datasets. This feature can go a long way towards improving the productivity of MapReduce users who lack the skills to optimize programs themselves due to lack of familiarity with MapReduce or with the data being processed.

Roadmap: The roadmap for the rest of the paper is as follows:

- Section 2 discusses job configuration parameters in Hadoop. It also presents empirical evidence of the performance impact and interactions of some of the parameters in order to motivate the need for automated techniques for setting parameters.
- Section 3 gives a concrete definition of the problem and discusses various solution approaches.
- Section 4 attempts to lay out a research agenda.

2. IMPACT OF MAPREDUCE JOB CONFIGURATION PARAMETERS IN HADOOP

We begin by presenting some empirical evidence to demonstrate differences in job running times between good and bad parameter settings in Hadoop. The purpose is to illustrate trends rather than to present a comprehensive performance study. We used a Hadoop cluster running on 17 nodes, with 1 master and 16 worker nodes. Each node has a dual core 2GHz AMD processor, 1.8GB RAM, 30 GB of local space, and runs Linux in a Xen virtual machine. Each worker node was set to run at most 4 mapper tasks and 2 reducer tasks concurrently. Thus, the cluster can run at most 64 mapper tasks in a concurrent *map wave*, and at most 32 reducer tasks in a concurrent *reduce wave*. Table 1 lists the subset of job configuration parameters that we considered in our experiments.

The MapReduce program that we consider is *TeraSort* which was used on one of Yahoo!'s Hadoop clusters to win the TeraByte sort benchmark in 2008. Figures 2(a) and 3(a) show two response surfaces that were generated by measuring TeraSort's end-to-end

[1]Some edits have been made for simplicity while retaining aspects needed for illustrative purposes in the paper.

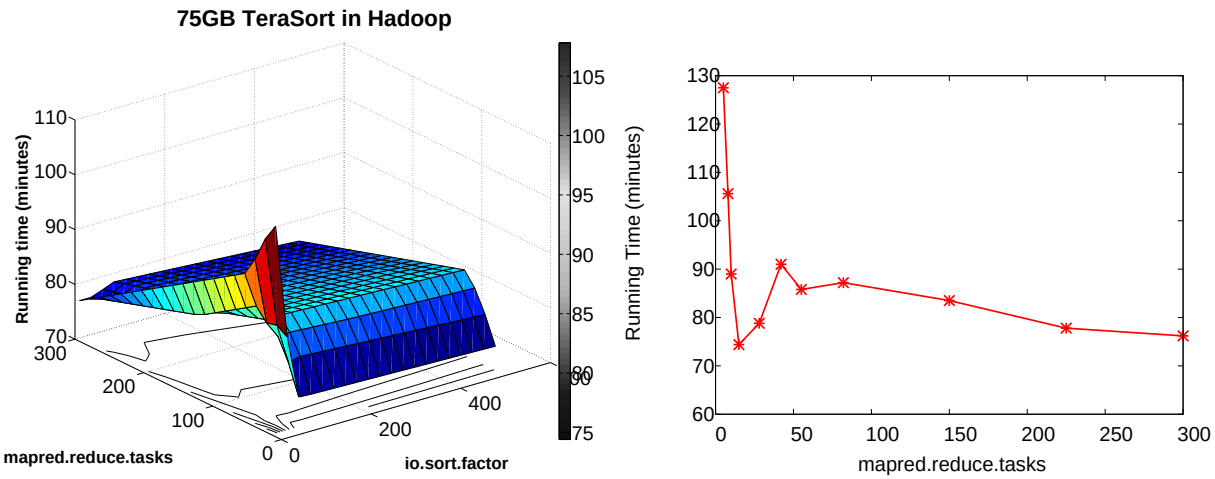

Figure 2: (a) 2D response surface of the TeraSort MapReduce program in Hadoop over a 75GB dataset, with mapred.reduce.tasks $\in [15, 300]$ and io.sort.factor $\in [10, 500]$; (b) a 1D projection of the surface for io.sort.factor=500 for mapred.reduce.tasks $\in [5, 300]$

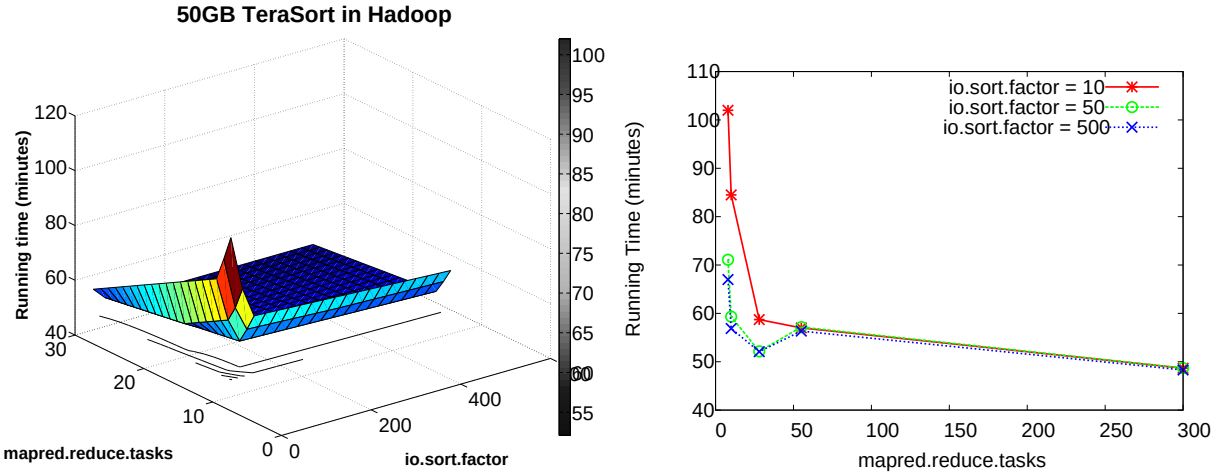

Figure 3: (a) 2D response surface of the TeraSort MapReduce program in Hadoop over a 50GB dataset, with mapred.reduce.tasks $\in [8, 28]$ and io.sort.factor $\in [10, 500]$; (b) 1D projections of the surface for different io.sort.factor for mapred.reduce.tasks $\in [8, 300]$

running time on our cluster for two different datasets (75GB and 50GB) generated using the TeraGen program that comes with the TeraSort package. The two parameters, mapred.reduce.tasks and io.sort.factor, are varied in these figures while all other job configuration parameters are kept constant.

Effect of Parallelism: The companion Figures 2(b) and 3(b) are projections of the respective response surfaces on to a single dimension (namely, mapred.reduce.tasks) to illustrate the predominant trend as the number of reducer tasks used to process the job is increased. Note that these figures show both improvements and drops in performance, with some interesting relationships between the two. The improvements come because of the increase in effective concurrency by utilizing more of the *reduce slots* in the cluster—recall that our cluster has 32 reducer slots across 16 worker nodes—as well as by having each reducer task process less data (which in turn can reduce I/O in nonlinear ways)—because the overall data size processed is fixed. The drops come because of the bound on effective concurrency per wave as well as task setup overheads.

Interactions among Parameters: A number of instances of inter-parameter interactions were seen in our experiments. For example, note from Figure 3(b) that changing io.sort.factor has significant impact on performance at lower values of mapred.reduce.tasks, but zero impact for mapred.reduce.tasks = 300.

Table 2 and Figure 4 provide more instances of inter-parameter interactions.[2] The settings of three parameters are varied in this case. Compared to Figures 2 and 3, the values of the mapred.job.* parameters (from Table 1) were all increased to 0.8 from their respective defaults, io.sort.mb was set to 200, mapred.child.java.opts to -Xmx300m, and mapred.inmem.merge.threshold to 0. Some interesting observations can be made from Table 2 and Figure 4:

- The (relatively obscure) io.sort.record.percent parameter has significant impact on performance. Furthermore, the impact comes over a variation of less than 5% in the parameter's value.
- Across different values of mapred.reduce.tasks, the pattern of change in performance as io.sort.record.percent is varied remains the same, but the magnitude of change differs.
- Performance for 10 reduce tasks is better than that for 28 reduce tasks. One of the main rules of thumb for improving the performance of Hadoop jobs recommends setting mapred.reduce.tasks to 28 for this scenario; which is highly suboptimal. (28 is around 0.9 times the total number of reduce slots in the cluster; the intuition is that effective concurrency is maximized while leaving some slots free for rerunning failed or slow tasks.)

Implications for Shared Clusters: As a follow up to the previous

[2]Row 10 in Table 2 shows the best performance achieved for 50GB TeraSort on our relatively ill-provisioned Hadoop cluster.

Row#	mapred. reduce.tasks	io.sort. factor	io.sort.record. percent	Job Running Time
1	10	10	0.10	1hr, 25mins, 25sec
2	10	10	0.15	1hr, 14mins, 54sec
3	10	500	0.10	1hr, 7mins, 11sec
4	10	500	0.15	1hr, 1mins, 1sec
5	28	10	0.10	1hr, 22mins, 54sec
6	28	10	0.15	1hr, 4mins, 57sec
7	28	500	0.10	1hr, 22mins, 24sec
8	28	500	0.15	1hr, 3mins, 46sec
9	300	10	0.10	45mins, 22sec
10	300	10	0.15	35mins, 9sec
11	300	500	0.10	44mins, 38sec
12	300	500	0.15	35mins, 56sec

Table 2: Performance of Hadoop 50GB TeraSort when the mapred.reduce.tasks, io.sort.factor, and io.sort.record.percent parameters are varied

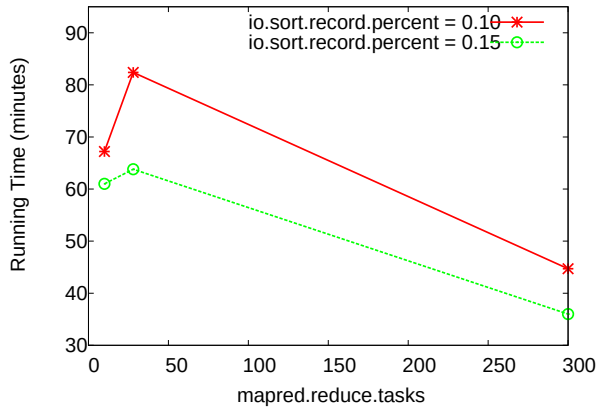

Figure 4: Visualization of the results for io.sort.factor=500 from Table 2

point, it is important to realize that getting better or equal performance with 10 reduce tasks means that the remaining 22 reduce slots in the cluster can be used for running jobs from other users in a shared cluster (shared Hadoop clusters are the norm rather than the exception). Furthermore, such knowledge can be useful for a cloud-based service like Amazon Elastic MapReduce to provision the right amount of capacity automatically for user-submitted MapReduce jobs while (i) reducing the cost incurred by the user in this pay-as-you-go environment, and (ii) being able to handle more users and flash crowds.

3. PROBLEM FORMULATION AND SOLUTION APPROACHES

The empirical evidence from Section 2 suggests that the performance of a MapReduce job J is some complex function of the job configuration parameter settings. In addition, the properties of the data as well as the resources allocated to the job in the cluster will impact its performance. There exists some function F_J such that:

$$y = F_J(\vec{p} \in \vec{P}, \vec{r} \in \vec{R}, \vec{d} \in \vec{D}) \quad (1)$$

Here, y represents a performance metric of interest for J (e.g., J's running time). $\vec{p}$ represents the setting of the job configuration parameters for a run of J. $\vec{r}$ represents the resources allocated for the run, and $\vec{d}$ represents the statistical properties of the data processed by J during the run. The response surfaces shown in Figures 2(a) and 3(a) are partial projections of Equation 1 for the TeraSort MapReduce program when run on our cluster.

Optimization Problem: Given an input dataset and resource allocation for running a MapReduce job, we can think of $\vec{p}$—which represents the setting of the job configuration parameters for a run of J—as specifying an *execution plan* for J. Different choices of the execution plan give rise to potentially different values for the performance metric of interest for J.[3] The problem we highlight in this paper is to automatically and efficiently choose a good execution plan for J given an input dataset and resource allocation.[4] The rest of this section discusses some potential approaches to address this problem.

3.1 Getting Rid of the Tuning Knobs

An initial reaction and a reasonable approach to this problem is to eliminate the "tuning knobs." In recent years, commercial database vendors have tried to eliminate database tuning knobs through code rewrites or by implementing self-tuning controllers (e.g., [16]). The Hadoop community is making similar efforts, e.g., for the high-impact io.sort.record.percent parameter considered in Table 2 and Figure 4 [9].

However, the majority of parameters have been created to expose some fundamental decision in the underlying code that cannot be made optimally without knowing the properties of the actual MapReduce job being processed, its input data, and resource allocation. We cannot truly eliminate these parameters. There are multiple implementation choices (as discussed in the following subsections) that differ in how to make the corresponding decision. Also bear in mind that eliminating an exposed knob can have serious software-engineering implications. Pushing the responsibility for setting the parameter into the underlying code can make the code complex, and consequently, hard to maintain and prone to bugs.

3.2 Database Query-Optimizer-Style Approach

Query optimizers in database systems deal with a similar optimization problem, namely, finding a good execution plan for a declarative SQL query. The approach used by query optimizers is to maintain statistics (e.g., histograms) about the input data; and given a query $\vec{q}$, to use these statistics in conjunction with a *cost model* to estimate the cost of various execution plans for $\vec{q}$. The execution plans are composed of operators—e.g., index scan, sort, and hash join—from a known and fixed set. Intuitively, the cost model has the form:

$$y = F(\vec{q} \in \vec{Q}, \vec{p} \in \vec{P}, \vec{r} \in \vec{R}, \vec{d} \in \vec{D}) \quad (2)$$

Here, y represents a performance metric of interest for the SQL query $\vec{q}$. $\vec{p}$ represents an execution plan for $\vec{q}$. $\vec{r}$ represents the resources allocated for running the plan, and $\vec{d}$ represents the statistical properties of the data processed by $\vec{q}$. Given $\vec{q}$, $\vec{r}$, and $\vec{d}$, a search algorithm (e.g., dynamic programming [12]) is used to find a good execution plan from the space of plans $\vec{P}$ supported by the database system's execution engine.

Provided this overall approach can be extended to MapReduce programs, efficient search algorithms can find good plans quickly. There is more than thirty years of work on database query optimization technology that can be leveraged. However, some stumbling blocks need to be overcome to make this approach work for ad hoc MapReduce programs:

- *Black-box map and reduce functions:* The cost model will now have to account for the fact that map and reduce functions are usually written in programming languages like Java, Python, and C++ that are not restrictive or declarative like SQL.

[3]The problem becomes more complex if multiple performance metrics are involved, e.g., suppose we want to minimize running time subject to a maximum bound on per-node memory usage.

[4]The resource allocation can also be considered a choice that has to be made automatically [10].

- *Lack of statistics about the input data:* Little knowledge about the input data may be available before the job is submitted. Keys and values are often extracted dynamically from the input data by the map function, so schema and statistics about the data may be unknown (like in Figure 1).
- *Differences in plan spaces:* The execution plan space for SQL queries is very different from the plan space of job configuration parameters outlined in Section 2 for MapReduce programs; so algorithms from SQL query optimizers may not translate directly for optimizing MapReduce programs.

Nevertheless, for at least two reasons, it will be a serious mistake to label the work on centralized and parallel database query optimization as inapplicable for MapReduce job configuration parameter optimization. First, many MapReduce programs are written once and run many times over their lifetime (usually on different datasets). Programs for *extract-transform-load (ETL)* and report generation are good examples. Properties of such programs as well as good configuration settings for them can be learned over time in the spirit of *learning optimizers* like Leo [15]. Second, higher-level languages for MapReduce frameworks like HiveQL and Pig allow schema to be associated with base and intermediate data.

3.3 Use of Dynamic Profiling

One approach to overcome some of the stumbling blocks from Section 3.2 is to invest some resources upfront—i.e., before actual job execution starts—to collect any missing information needed to optimize the actual job's configuration parameters. We will illustrate this approach with an example. One of the main considerations in optimizing the TeraSort program (and, in general, in optimizing most parallel programs) is to balance the work equally across all reducer tasks.[5] For achieving such balance—assuming fairly homogeneous resource allocation to all reducer tasks—the overall output data generated by all the mapper tasks has to be partitioned equally across all the reducer tasks.

The input data distribution may not be known ahead of time, so the TeraSort implementation first collects a sample of the input data. If there are k reducer tasks, then the quantiles in the sample are computed that split the sample data into k equal-sized partitions. These k-1 *split points* are then used during actual job execution to split the data into approximately equal-sized partitions. As one would expect, the actual load-balancing achieved would depend on how well the sample captures the actual data distribution. The literature on this topic is vast (e.g., *sample sort* [3] and *probabilistic splitting* [7]).

The basic idea of *dynamic profiling* through sampling applies to parameter configuration for MapReduce jobs, but some challenges arise:

- A strong assumption in the sampling-based partitioning technique outlined above is that the data type and the data distribution of the key in the input data (before applying the map function) are the same as those in the data generated after the map function has been applied. This assumption holds for sorting, but it does not hold for the majority of MapReduce programs (e.g., see the map function in Figure 1).
- Most (if not all) previous implementations of sample sort and probabilistic splitting assume that the input data sample can be assembled by sampling randomly either at the record level or at the block level. However, given an arbitrary MapReduce program, there are restrictions on how the input data can be accessed. Usually, the data can only be accessed in a first-to-last sequential fashion (based on an iterator interface) over predetermined partitions of the input data (called *splits* in Hadoop).

In this setting, designing an efficient *sampling harness* for dynamic profiling in MapReduce frameworks is a nontrivial research and engineering challenge which, to our knowledge, has not been considered before. There are some desiderata for such a sampling harness. Extra run-time overhead must be kept low, ideally some bounded fraction of the job's overall running time. There is the danger of unrepresentative samples generating incorrect estimates that lead to performance degradation rather than improvement. Thus, developing or adopting sampling techniques that give probabilistic error guarantees is highly desirable [13].

The sampling harness should not require major changes or additions to the MapReduce code base; higher code complexity lowers the chances of the harness being adopted by the core developers. From this perspective, it is desirable to leverage features of the current MapReduce framework whenever possible rather than rolling out our own. Next, we discuss some specific questions that the design of the sampling harness will have to account for.

Where is the sampling run from? TeraSort does *client-side* sampling, i.e., it runs the sampling from the client node where the job is submitted. An alternative is to run an extra MapReduce job, called the *sampling job*, to do the sampling as well as to use the collected sample to determine good parameter settings for the actual job which will be run subsequently. Both techniques have their pros and cons. Client-side sampling is centralized and can cause significant overhead on the client node (which often is a worker or master node), especially if the map function has to be run on the collected samples. However, client-side sampling avoids the fairly high job setup overheads of the sampling job.

When is the sampling done? Even with dynamic profiling, the samples can be precomputed before the job is submitted. This point may seem contrary to what we said earlier. To collect samples from an input file, we only need to know the record boundaries (e.g., the newline character in text files); the actual schema need not be known at this point. Such "precomputed" samples can be collected efficiently and nonintrusively, e.g., through coin-toss random sampling when the file is loaded into the system. Precomputed samples can be processed much more efficiently by the sampling job compared to starting mapper tasks for the entire data while sampling at job submission time.

What type of sampling is used? TeraSort uses a simple sampling strategy. It accesses a predetermined number s of input data splits (usually, each HDFS block in the input forms a split), and grabs the first n keys and values from each sampled split. (Both s and n are specified statically.) While this strategy is efficient—recall the restrictions on data access—it is unclear whether this strategy can give strong probabilistic error guarantees. It is possible to do coin-toss sampling or reservoir sampling [18] (which bounds memory requirements under an unknown number of total keys) on the sampled splits. These strategies may give better probabilistic error guarantees but at higher cost since the sampled split will have to be processed completely. Note from Table 1 that the default block/split size is 64MB, which is much larger than typical database or filesystem blocks.

3.4 Reacting Through Late Binding

While some of the concerns of the optimizer-style approach are potentially addressed by dynamic profiling, it also has to rely on a cost model to estimate the performance of different settings of configuration parameters like those in Table 1. It is unclear whether such models can be developed to work at scale, and needs further research. It is therefore important to design and evaluate ap-

[5]The *Partitioner* class user for this purpose is a configuration parameter to a MapReduce job, with hash partitioning as the default.

proaches that either rely less or do not rely at all on the existence of cost models. This section and the next discuss such approaches.

The dependence on cost models can be reduced through "late-binding" approaches that delay the setting of one or more parameters until after some part of the execution has been observed. This approach can be applied for parameters whose value can either (i) be changed during job execution, or (ii) does not have to be set until part of the execution is complete. Hadoop uses a late-binding approach to decide whether to partially aggregate the output of a mapper task. Partial aggregation—which can be done in a MapReduce job by specifying a *Combiner* class as a job configuration parameter—essentially trades compute cycles for potential savings in disk I/O and network data transfer [14]; so the choice of whether to apply a specified Combiner or not has to be done carefully.

The concept of *speculative execution* can be useful here as well as in Section 3.5. If some part of the execution can be done and then undone efficiently (if needed), then reactive setting can be applied to a larger class of parameters while keeping the performance impact low. (However, the risk of thrashing has to be dealt with.) Currently, MapReduce frameworks have support for speculative execution only at the granularity of full mapper and reducer tasks (which is aimed at lowering the chances of some slow or failed tasks increasing the overall job completion time).

3.5 Competition-based Approaches

The idea here is (i) to start multiple instances of the same task concurrently, with each instance having a different setting of the job configuration parameters; (ii) to quickly identify the best instance; and (iii) to kill all the other instances. Similar ideas have been used for adaptive processing of queries in database systems (e.g., [1]). The speculative execution feature of MapReduce frameworks is potentially useful here. Currently, the speculative instances are run with the same parameter configuration as the main instance.

While the reactive and competitive approaches reduce or eliminate the need for cost models, the reality is more subtle. The reactive and competitive approaches have to predict the performance of an entire job run by observing only an initial part of the execution. If mapper and reducer tasks run in multiple waves—which may be a common occurrence for good performance in small MapReduce clusters (say, with ≤ 20 nodes)—then the reactive and competitive approaches can adjust the configuration parameter settings of the current wave based on observations from the previous waves.

3.6 Hybrid Approaches

It is possible that no single approach is good enough to set all high-impact job configuration parameters; so a hybrid approach that combines two or more of the above approaches may be needed.

4. RESEARCH AGENDA

We conclude this position paper by listing a five-point research agenda for automatic setting of job configuration parameters for MapReduce programs:

1. A necessary first step is to conduct a comprehensive empirical study with a representative class of MapReduce programs and different cluster configurations to understand (and potentially model) parameter impacts, interactions, and response surfaces. This study can inform the remaining steps. Monitoring data can be collected and analyzed using tools like Chukwa [4]. Since job running times are high, ideas from experiment-driven management (e.g., quickly generating approximations of response surfaces like Figure 2(a)) can also be beneficial here [2].
2. Is it possible to develop a cost model that is useful to recommend good parameter settings for MapReduce job configuration parameters? It seems feasible to develop a cost model that can deal with common classes of map and reduce functions by providing mechanisms to learn and plug in *profiles* for these functions. At the same time, it is possible that the best cost model that works at scale is a collection of parameterized rules.
3. Designing an efficient sampling harness will possibly become relevant once we have a good understanding of cost models as well as parameter impacts and interactions as a function of the properties of the job, input data, and allocated resources.
4. This paper focused on optimizing job configuration parameters for ad hoc MapReduce programs. A separate but closely-related problem is to tune the performance of a MapReduce program that is run repeatedly (e.g., for daily report generation) and whose current performance is unsatisfactory.
5. Insights from automatic optimization of single MapReduce programs will be crucial while addressing other optimization problems in the MapReduce framework like: (i) generating an execution plan composed of one or more MapReduce jobs for a higher-level operation like join [11]; (ii) optimizing a given graph (e.g., a chain or a DAG) of MapReduce programs that are related through input-output relationships; and (iii) tuning the layout of input data accessed by a MapReduce program.

5. REFERENCES

[1] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of SIGMOD Conf.*, May 2000.

[2] S. Babu, N. Borisov, S. Duan, H. Herodotou, and V. Thummala. Automated Experiment-Driven Management of (Database) Systems. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, May 2009.

[3] G. Blelloch, C. Leiserson, B. Maggs, C. G. Plaxton, S. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proc. of SPAA*, 1991.

[4] J. Boulon et al. Chukwa: A Large-scale Monitoring System. In *Cloud Computing and its Applications*, 2008.

[5] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proc. of VLDB Conf.*, Sept. 2000.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, 2004.

[7] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. In *Proc. of PDIS*, 1991.

[8] Apache Hadoop. http://hadoop.apache.org/.

[9] Map-side sort is hampered by io.sort.record.percent. issues.apache.org/jira/browse/MAPREDUCE-64.

[10] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *Proc. of the First Workshop on Hot Topics in Cloud Computing*, June 2009.

[11] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *Proc. of USENIX Annual Technical Conf.*, 2008.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of SIGMOD Conf.*, June 1979.

[13] S. Seshadri and J. F. Naughton. Sampling Issues in Parallel Database Systems. In *Proc. of EDBT Conf.*, 1992.

[14] A. Shatdal and J. F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proc. of SIGMOD Conf.*, 1995.

[15] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of VLDB Conf.*, Sept. 2001.

[16] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive Self-tuning Memory in DB2. In *Proc. of VLDB Conf.*, 2006.

[17] A. Thusoo et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[18] J. S. Vitter. Random Sampling with a Reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, Mar. 1985.