MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs

Juwei Shi[†⋆], Jia Zou[†], Jiaheng Lu[⋆], Zhao Cao[†], Shiqiang Li[†] and Chen Wang[†]
[†]IBM Research - China, Beijing, China, {jwshi, jiazou, caozhao, shiqli, wangcwc}@cn.ibm.com
[⋆]Renmin University of China, Beijing, China, jiahenglu@ruc.edu.cn

ABSTRACT

MapReduce based data-intensive computing solutions are increasingly deployed as production systems. Unlike Internet companies who invent and adopt the technology from the very beginning, traditional enterprises demand easy-to-use software due to the limited capabilities of administrators. Automatic job optimization software for MapReduce is a promising technique to satisfy such requirements. In this paper, we introduce a toolkit from IBM, called *MRTuner*, to enable holistic optimization for MapReduce jobs. In particular, we propose a novel *Producer-Transporter-Consumer* (PTC) model, which characterizes the tradeoffs in the parallel execution among tasks. We also carefully investigate the complicated relations among about twenty parameters, which have significant impact on the job performance. We design an efficient search algorithm to find the optimal execution plan. Finally, we conduct a thorough experimental evaluation on two different types of clusters using the HiBench suite which covers various Hadoop workloads from GB to TB size levels. The results show that the search latency of *MRTuner* is a few orders of magnitude faster than that of the state-of-the-art cost-based optimizer, and the effectiveness of the optimized execution plan is also significantly improved.

1. INTRODUCTION

Nowadays MapReduce based data-intensive computing solutions are increasingly deployed as production systems. These systems become popular in traditional industries such as banking and telecommunications, due to demands on processing fast-growing volumes of data [10]. Enterprises usually demand easy-to-use and manageable softwares. However, MapReduce-based systems such as Hadoop from the open source community hold a high learning curve to IT professionals, especially on system performance management to better utilize the system resources. The parameter configuration in Hadoop requires the understanding of the characteristics of the job, data and system resources, which is beyond the knowledge of traditional enterprise IT people. Another interesting scenario about MapReduce job tuning comes from analytic services (e.g. Elastic MapReduce[1]) on the cloud. The users, such as data scientists, do not know how to correctly choose the MapReduce parameters to accelerate the job execution. Therefore, motivated by above scenarios, this paper addresses the challenge to build an automatic toolkit for MapReduce job optimization.

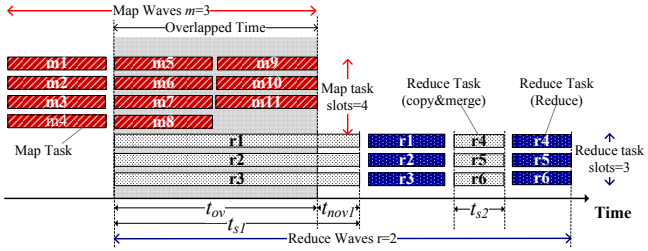


Figure 1: The Pipelined Execution of a MapReduce Job

Job optimization (i.e. query optimization) [1] technologies are widely used in relational database management systems (RDMBS) over the past few decades. Traditional query optimizers build a cost model to estimate query processing costs, and design search algorithms like dynamic programming to find the best execution plan. However, neither cost models nor search algorithms from RDBMS work for MapReduce because of the intrinsical system difference.

Cost-based optimization for MapReduce has been studied in [6, 7], which models the execution of individual Map or Reduce tasks, and simulates all the MapReduce execution plans to find the best one. This solution, while pioneering, has some drawbacks. For instance, existing MapReduce cost models [6, 11] focus on predicting the cost of individual Map or Reduce tasks, but rarely address the parallel execution among tasks. In the MapReduce programming model, the overall execution time may not be equal to the sum of the cost of each individual task, because of the potential saving from the overlapped time window among (Map and Reduce) tasks in the parallel execution. The overlaps among tasks should be considered in a more holistic optimizer to maximally utilize the limited hardware resources in enterprises. Further, Hadoop has more than 190 configuration parameters, out of which 10-20 parameters have significant impact on job performance. To address the issue of the high dimensionality, the existing algorithm [6] uses a random search algorithm, which may lead to sub-optimal solutions without deviation bounds.

To overcome the above two limitations, we have an in-depth study about MapReduce job optimization. To model the inter-task

 Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 13

[1]http://aws.amazon.com/elasticmapreduce/

parallelism of a MapReduce job, we use a pipelined execution model to describe the relations among tasks. A key property of the pipelined model is that part of the *Shuffle* stage can be overlapped with the *Map* stage. An example of the pipelined execution is shown in Figure 1 (We will elaborate task slots, Map and Reduce waves and other notions in Section 3). For this job execution, parts of the Shuffle tasks (r1 to r3) are overlapped with the Map tasks (m5 to m11) in the time duration t_{ov}. The overlap is affected by some critical parameters like the compression option, the number of Map and Reduce tasks, and the number of copy threads. The overlapped time window makes the design of MapReduce cost models challenging, and we thereby identify a few fundamental tradeoffs (See Section 3.1 for details) to guide the design of a new MapReduce cost model. As the foundation of this study, we propose the *Producer-Transporter-Consumer (PTC)* cost model to characterize the tradeoffs in the MapReduce parallel execution. The key intuition of the PTC model is that, for a MapReduce job execution plan, the generation of Map outputs (i.e. by the Producer), the transportation of Map outputs (i.e. by the Transporter) and the consumption of Map outputs (i.e. by the Consumer) should keep pace with each other so that utmost utilization of the system resources (CPU, disk, memory and network) is achieved to minimize the overall running time of the parallel execution.

To address the challenge of the high dimensionality, we focus on the search space reduction without losing the accuracy of the optimization. By investigating the complicated relations among parameters, we have two important findings. Firstly, given the identified tradeoffs, some parameters should be optimized by a holistic cost model. For example, considering the overlapped Shuffle duration, the running time of the Map stage affects that of Reduce tasks, meaning that the Map and Reduce stages should not be optimized separately. Secondly, we figure out the dependencies among parameters, and find that some parameters can be represented by existing variables. For example, given the estimated Map selectivity (i.e. the ratio of output to input) and the average record size of Map outputs, the size of the Map output buffer can be calculated from the input split size. Thus this parameter is not an independent variable in our cost model. As a result, the careful investigation of relations among different parameters facilitates the reduction of the search space and enables the design of a fast search method.

In this paper, we introduce the design and implementation of a toolkit, namely *MRTuner*, to enable holistic optimization for MapReduce jobs. The toolkit has been used in IBM and with several customers for pilots. The key contributions of this paper are as follows.

- We design and implement a toolkit to enable holistic optimization for MapReduce jobs, which covers parameters of MapReduce and HDFS.
- We identify four key factors to model the pipelined execution plan of a MapReduce job, and propose a *Producer-Transporter-Consumer (PTC)* cost model to estimate the running time of a MapReduce job.
- We figure out the relations among the performance sensitive parameters, and design a fast search method to find the optimal execution plan.
- We conduct a thorough experimental evaluation on two different types of clusters using HiBench [8], covering various workloads from GB to TB levels. The search time of the MRTuner job optimizer outperforms that of the state of the art cost-based MapReduce optimizer by a few orders of magnitude. More importantly, MRTuner can find much better execution plans compared with existing MapReduce optimizers.

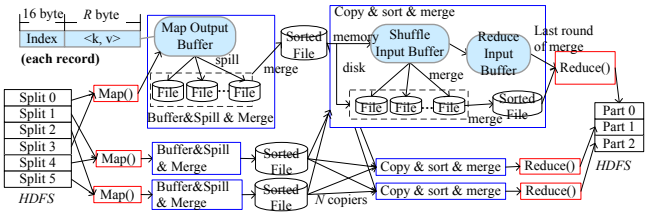


Figure 2: Hadoop MapReduce Internals

As an in-depth study of MapReduce performance, we believe the insights in designing and evaluating MRTuner can also benefit the Hadoop system administrators and IT professionals who are interested in the performance tuning of Hadoop MapReduce.

The rest of the paper is organized as follows. We begin with Section 2 to introduce MapReduce. We dive into our new PTC model in Section 3. In Section 4, we describe the architecture of MRTuner. Section 5 is devoted to the implementation of MRTuner. Then the related work is showed in Section 6. Finally, we present the experimental results and conclude in Section 7 and 8 respectively.

2. PRELIMINARIES

In this section, we introduce some preliminary knowledge about MapReduce in Hadoop, which will be used in the rest of this paper.

Job Execution. MapReduce is a parallel computation framework that executes user defined Map and Reduce functions. The task-slot capacity (i.e. the maximum number of Map or Reduce tasks that can run simultaneously) is fixed when the cluster instance gets started. When a job is submitted, tasks will be assigned to slave nodes for the execution when there are available task slots on these nodes. When the first round of Map tasks finish, Reduce tasks are able to start to transfer Map outputs. As shown in Figure 1, for example, the number of Map and Reduce slots is 4 and 3, respectively. Given 11 Map tasks and 6 Reduce tasks, there are 3 waves (rounds) of Map tasks and 2 waves of Reduce tasks. When the first wave of Map tasks (i.e. m1 to m4) completes, the first wave of Reduce tasks (i.e. r1 to r3) would start to copy Map outputs.

Map Task. Map tasks read input records, and execute the user defined Map function. The output records of Map tasks are collected to the Map output buffer whose structure is shown in the top left of Figure 2. For each record, there are 16 bytes meta-data for sorting, partitioning and indexing. There are parameters to control spill thresholds for both the meta-data and data buffers. When the buffer exceeds the configured threshold, the buffered data will be spilled to disk. When all the Map outputs are generated, the Map task process will merge all the spilled blocks to one sorted file. In other words, if the size of Map outputs exceeds the threshold of the configured buffer, there is extra disk I/O for spill and merging. Moreover, the Map output data can be compressed to save disk I/O and network I/O.

Reduce Task. When a Reduce task starts, concurrent threads are used to copy Map outputs. The number of concurrent copy threads is determined by a parameter. This parameter impacts the parallel execution of a job, and there is a tradeoff between the context switch overhead and the Shuffle throughput. When Map outputs are copied to the Reduce side, multiple rounds of on-disk and in-memory combined sort & merge are performed to prepare Reduce inputs. There is also a buffer in the Reduce side to keep Reduce inputs in memory without being spilled to disk. Finally, the outputs of Reduce tasks are written to HDFS.

3. A NEW COST MODEL FOR MAPREDUCE

In this section, we first elaborate the key findings in the parallel execution of MapReduce jobs, and then present a new cost model.

3.1 MapReduce Inter-task Tradeoffs

The key idea of inter-task optimizations is to overlap the execution of Map and Reduce stages. To reduce the total running time of a job, the Shuffle stage, which is network I/O bounded, is optimized to be executed in parallel with the Map stage (without network I/O). Specifically, when the first wave of Map tasks completes, the first wave of Reduce tasks would start to copy Map outputs while executing the second wave of Map tasks. Based on the pipelined execution model, we identify four key factors to characterize the MapReduce execution path and important tradeoffs.

- **The number of Map task waves m.** The factor m is determined by the Map input split size (or the number of Map tasks) and the Map task slot capacity.
- **The Map output compression option c.** The factor c is the parameter of the Map output compression option.
- **The copy speed in the Shuffle phase v.** The factor v is determined by the number of parallel copiers, the network bandwidth and the Reduce task slot capacity.
- **The number of Reduce task waves r.** The factor r is determined by the number of Reduce tasks and Reduce task slot capacity.

We analyze the impact of the key factors on job performance in the pipelined execution shown in Figure 1. t_{ov} is the overlapped Shuffle time, and we assume Reduce tasks copy Map outputs with the amount of D_{ov} during t_{ov}. If m increases (i.e. the number of Map tasks increases), D_{ov} increases (i.e. more Map outputs can be copied in the overlapped manner), with the cost in task creation, scheduling and termination. If r increases, D_{ov} decreases (only the first wave of Reduce tasks can perform the overlapped copy). But there is a benefit that more Reduce inputs can be hold in memory without being spilled to the disk, since the maximum available Reduce input buffer is fixed for each wave of Reduce tasks. When the number of copy threads increases, v increases, at the cost of more context switch overhead. If the Map stage is slowed down by the context switch overhead, the time window t_{ov} increases. Therefore, we summarize the following fundamental tradeoffs that should be addressed by the MapReduce cost model.

- (T1) Selecting m is a tradeoff between the time window for the overlapped copying and task scheduling overhead. When m increases, more Map outputs can be transferred in the overlapped manner, but the cost to schedule the increased wave of Map tasks should not be neglected. Further, the overlapped time duration affects the copy speed v.
- (T2) The compression option c is beneficial if the cost to compress and decompress the *total* Map outputs is less than the cost of transferring the *additional* amount of Map outputs without compression, excluding D_{ov}. It means that we should exclude the overlapped Shuffle time t_{ov} to estimate the running time of a job.
- (T3) Selecting the copy speed v is a tradeoff between context switch overhead and the amount of overlapped shuffled data. Because of the context switch overhead of copy threads, the speedup of transferring Map outputs may not lead to the reduction of the overall execution time. In other words, we should minimize the number of Reduce copy threads while trying to catch the throughput of Map outputs generation.
- (T4) Selecting the number of Reduce waves r is a tradeoff between the amount of overlapped shuffled Map outputs and the saved I/O overhead from buffered Reduce inputs. Since the buffer allocated for each Reduce task is fixed, we need to have more Reduce tasks to keep more Reduce inputs in memory. But it comes at the cost of the potential increase of the number of Reduce waves, which leads to the fewer Map outputs that can be copied in the overlapped manner.

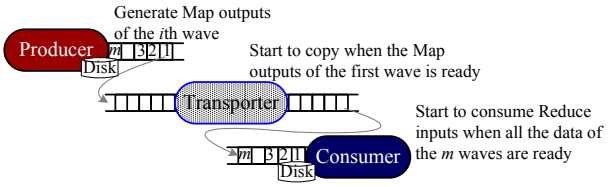


Figure 3: The Components of the PTC Model

3.2 Producer-Transporter-Consumer Model

We propose a cost model, namely *Producer-Transporter-Consumer (PTC)*, to model the key tradeoffs. As shown in Figure 3, the PTC model consists of three components: Producer, Transporter and Consumer. The *Producer* is responsible for reading Map inputs from HDFS, processing the user defined Map function, and generating Map outputs on the local disk. The *Transporter* is in charge of reading Map outputs, copying it to the Reduce side, and merging it to the disk. The *Consumer* needs to read Reduce inputs for the user defined Reduce function.

The assumptions of the PTC model are given as follows.

- The Transporter starts to transfer Map outputs when the first wave of Map tasks finishes.
- The Transporter can only pre-fetch Map outputs for the first wave of Reduce tasks, meaning that at most $1/r$ of total Map outputs can be copied in the overlapped manner.
- Only the network I/O cost may be eliminated in estimating the running time. The disk I/O cost can not be excluded since both Map and Reduce tasks preempt this resource.
- We assume that there is no skewness among tasks. We leave it as future work to consider skew tasks in the PTC model (See the discussion in Section 7.3).
- For simplicity of presentation, we assume that the allocated system resources of each task slot are the same. The PTC model can be easily extended to handle heterogeneous slots.

Cost Estimation of the PTC Model. The Producer models the process of generating Map outputs in m waves. The main result of the Producer model is summarized as the following proposition.

PROPOSITION 3.1. *Given the inputs $\mathcal{D}$, Map and Reduce slots $\{N_{ms}, N_{rs}\}$ and four factors $\{m, r, v, c\}$, the running time of the Producer to process all the Map tasks is*

$$\mathcal{T}_{producer} = t_{map}(\mathcal{D}, c) + t_{schedule}(m) + t_{cs}(\mathcal{D}, c, v, r) \quad (1)$$

where $t_{map}(\cdot)$ is the running time of Map tasks. $t_{cs}(\cdot)$ is the context switch time. $t_{schedule}(\cdot)$ is the time spent on task scheduling.

PROOF. Suppose that Map tasks perform only one-pass sort-merge (See optimizations in section 5.2.1), the running time $t_{map}(\cdot)$ is determined by the inputs $\mathcal{D}$ and the compression option c. $t_{cs}(\cdot)$ is the context switch time led by parallel Reduce copy threads, which is determined by the inputs $\mathcal{D}$, the compression option c, the copy speed v and the number of Reduce waves r. Finally, for each wave of Map tasks, there is a penalty factor $t_{schedule}(\cdot)$ that represents the overhead to create, schedule and terminate tasks. Therefore, Eq 1 holds in the proposition 3.1 □

The implementation of $t_{map}(\cdot)$, $t_{cs}(\cdot)$ and $t_{schedule}(\cdot)$ is given in Section 5.1. We do not have N_{ms} and N_{rs} as arguments of these functions since they are fixed after the cluster is started. The key design of the Producer model is to consider the context switch and task scheduling overhead in the pipelined execution.

The Transporter models the process of transferring Map outputs. The main result of the Transporter model is summarized as follows.

PROPOSITION 3.2. *Given the inputs $\mathcal{D}$, Map and Reduce slots $\{N_{ms}, N_{rs}\}$ and four factors $\{m, r, v, c\}$, the running time of the Transporter to transfer the Map outputs is*

$$\begin{aligned}\mathcal{T}_{trasporter} =& \min((\frac{m-1}{mrv} \cdot D_s - \frac{m-1}{m} \cdot \mathcal{T}_{producer} \\ & - t_{lrw}(\frac{m-1}{mr} \cdot D_s)), 0) \\ & + \frac{(2mr - m - r + 1)}{mrv} \cdot D_s + t_{lrw}(D_s))\end{aligned} \quad (2)$$

where $\mathcal{T}_{procuder}$ is given in Proposition 3.1. D_s is the amount of transferred data which is determined by $\mathcal{D}$ and c. $t_{lrw}(x)$ is the time to read and write data with the amount of x on local disk.

PROOF. As shown in Figure 1, since only the first wave of Reduce tasks can perform copy in parallel with the Map stage, the maximum amount of Map outputs copied in the overlap manner is $D_{o\max} = D_s \cdot \frac{m-1}{m} \cdot \frac{1}{r}$. If all the $D_{o\max}$ can be transferred before Map tasks finish (i.e. within the time window $t_{ov} = \frac{m-1}{m} \cdot \mathcal{T}_{producer} + t_{lrw}(\frac{m-1}{mr} \cdot D_s)$, where $\frac{m-1}{m} \cdot \mathcal{T}_{producer}$ is the time to process the first $m-1$ waves of Map tasks, and $t_{lrw}(\frac{m-1}{mr} \cdot D_s)$ is the time to read (in the Map side) and write (in the Reduce side) shuffled data), the non-overlapped time to transfer $D_{o\max}$ is $t_{nov1} = 0$. Otherwise, there is the additional time t_{nov1} to transfer Map outputs with the amount of $D_{o\max} - t_{ov} \cdot v$ after all the Map tasks finish. Thus the running time (excluded the overlapped time) to transfer Map outputs for the first wave of Reduce tasks is

$$\begin{aligned}t_{s1} = t_{nov1} =& \min((D_s \cdot \frac{m-1}{m} \cdot \frac{1}{r} \cdot \frac{1}{v} - \frac{m-1}{m} \cdot \mathcal{T}_{producer} \\ & - t_{lrw}(\frac{m-1}{mr} \cdot D_s)), 0)\end{aligned}$$

Next we derive the time to copy the rest of Map outputs D_n. The amount of outputs generated by the last wave of Map tasks is $D_s \cdot \frac{1}{m}$. The amount of the rest of the Map outputs generated for the second to the rth wave of Reduce tasks is $D_s \cdot \frac{m-1}{m} \cdot \frac{r-1}{r}$. Thus the time to transfer D_n is

$$\begin{aligned}t_{s2} =& D_s \cdot (\frac{1}{m} + \frac{m-1}{m} \cdot \frac{r-1}{r}) \cdot \frac{1}{v} \\ =& \frac{(2mr - m - r + 1)}{mrv} \cdot D_s\end{aligned}$$

Finally, the running time of reading Map outputs (in the Map side) and writing Reduce inputs (in the Reduce side) is $t_{srw} = t_{lrw}(D_s)$. Based on the assumption that Map and Reduce tasks preempt the disk I/O resource, the time t_{srw} can not be excluded in estimating the running time for the Transporter.

Therefore, Proposition 3.2 shows the sum of t_{s1}, t_{s2} and t_{srw}, as desired. □

The Consumer models the process of consuming Reduce inputs. The main result of the Consumer cost model is summarized as the following proposition.

PROPOSITION 3.3. *Given the inputs $\mathcal{D}$, Map and Reduce slots $\{N_{ms}, N_{rs}\}$ and four factors $\{m, r, v, c\}$, the running time of the Consumer to consume the Reduce inputs is*

$$\mathcal{T}_{consumer} = t_{reduce}(\mathcal{D}, c) + t_{schedule}(r) - t_{lrw}(B_r) \cdot r \quad (3)$$

where $t_{reduce}(\cdot)$ is the running time to process Reduce inputs without enabling the Reduce side buffer. B_r is the allocated Reduce side buffer for each Reduce task.

PROOF. $t_{reduce}(\cdot)$ is determined by the shuffled Map outputs size D_s. D_s is determined by the Map inputs $\mathcal{D}$ and the compression option c. Thus $t_{reduce}(\cdot)$ is determined by $\mathcal{D}$ and c. For a Reduce task, the amount of Map outputs that can be hold in the Reduce side buffer is B_r (i.e. the amount of allocated Reduce side buffer for each Reduce task). Thus the total time can be saved by the Reduce side buffer is $t_{lrw}(B_r) \cdot r$. Moreover, there is a penalty $t_{schedule}(\cdot)$ to schedule Reduce tasks. Therefore, we have Eq 3 in the proposition 3.3. □

Based on the results in Proposition 3.1, 3.2 and 3.3, we have the global cost function of the PTC model as follows.

$$\mathcal{T}_{PTC} = \mathcal{T}_{producer} + \mathcal{T}_{transporter} + \mathcal{T}_{consumer} \quad (4)$$

Implication of the PTC Model. We use a serial of illustrative examples with different settings of the factors to demonstrate how to address the tradeoffs with the PTC model. All the tasks in the same wave are shown by a single line. Since the scheduling overhead $t_{schedule}$ is constant, we do not show it in examples for simplicity.

Figure 4 (a) shows the running time break-down when $\{m = 3, r = 1, c = true, v = 2\}$. Under this setting of the factors, we assume that: 1) The running time of each wave of Map tasks is $t_{map} = 20\ seconds$ (The unit is omitted in the following examples for simplicity). 2) The context switch time is $t_{cs} = 5$ for the second and the third waves of Map tasks respectively. 3) The network copy time of each Map wave is $t_s = 15$. 4) For each wave of Shuffle tasks, the local read and write time is $t_{lrw} = t_{lr} + t_{lw} = 5 + 5 = 10$. Note that t_{lrw} can not be overlapped with the Map stage. So we illustratively break Map tasks in the duration of t_{lrw}. 5) The running time of the wave of Reduce tasks is $t_{reduce} = 35$.

Since there is only one wave of Reduce tasks, $2/3$ of the Map outputs can be copied in parallel with the Map stage, and the non-overlapped copy time of the first Reduce wave is $t_{nov1} = 15$. Thus the running time to transfer Map outputs is $t_{s1} = 15$. The total running time of this job is 150.

Figure 4 (b) illustrates the tradeoff T1. The factors of this job is set to be $\{m = 1, r = 1, c = true, v = 2\}$. Compared with the job in Figure 4 (a), there is only one wave of Map tasks. Although the context switch time t_{cs} is eliminated, the copy of all the Map outputs can not be overlapped. This leads to additional 30 for the Shuffle stage. The total running time of this job is 170.

Figure 4 (c) illustrates the tradeoff T2. The factors of this job is set to be $\{m = 3, r = 1, c = false, v = 2\}$. In comparison to the job in Figure 4 (a), the compression option c is disabled. The gray dotted line points to the wave of Map outputs that the copy task is responsible for. Since there is no compression/de-compression overhead, t_{map} and t_{reduce} are reduced to 5 and 15 respectively, but at the cost that both t_{lr} and t_{lw} are increased to 10, and t_s

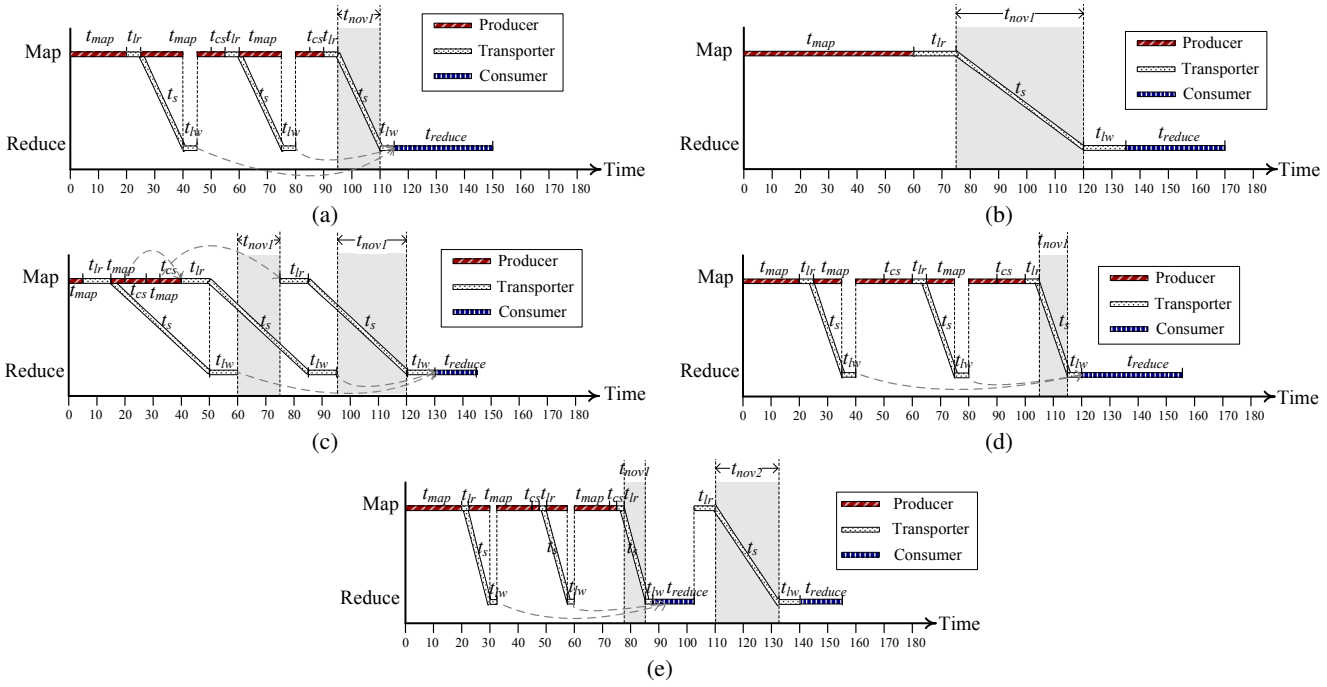


Figure 4: Examples to illustrate the PTC model

is increased to 35. The context switch time for each wave t_{cs} is increased to 7.5 due to the increased Map outputs. Then, t_{nov1} is increased to 40. The running time of this job is 145.

Figure 4 (d) illustrates the tradeoff T3. The factors of this job is set to be $\{m = 3, r = 1, c = true, v = 3\}$. Compared with the job in Figure 4 (a), the copy speed is increased to reduce t_s to 10, which is obtained at the cost that the context switch time t_{cs} is increased to 10. Although the non-overlapped copy time t_{nov1} is reduced to 10, the total running time is increased to 155 due to the increased context switch overhead. Actually, only the last wave of Reduce copy benefits from the increased copy speed.

Figure 4 (e) illustrates the tradeoff T4. The factors of this job is set to be $\{m = 3, r = 2, c = true, v = 2\}$. The number of Reduce waves is increased to 2 comparing to Figure 4 (a). Each wave of Reduce tasks can use the Reduce side buffer to reduce disk I/O, which saves 2.5 seconds for each Reduce wave ($t_{reduce} = 15$). Because of the reduction of the overlapped copy amount, t_{cs} is reduced to 2.5 for each wave. Since only the first wave of Reduce tasks can copy Map outputs in the overlapped manner, the total non-overlapped copy time is increased as $t_{nov} = t_{nov1} + t_{nov2} = 7.5 + 22.5 = 30$. The running time of this job is 155.

4. SYSTEM ARCHITECTURE

Before discussing the specifics of our system, we first provide the design goals of MRTuner.

- *Integrated Self-tuning Solution.* MRTuner is designed to optimize MapReduce jobs from the whole Hadoop stack including MapReduce and HDFS.
- *Low-latency Optimizer.* The MRTuner job optimizer is designed to respond in sub-second.
- *Optimization for Job Running Time.* MRTuner is designed to optimize the job running time. Note that the running time is used to measure the execution of parallel tasks. The optimization of the running time does not conflict with that of system resource utilization.
- *Loosely Coupled with Hadoop.* MRTuner is designed to be loosely coupled with Hadoop. We choose to build job profiles from the standard Hadoop log.

We provide the architecture of MRTuner in Figure 5. MRTuner consists of three components: the Catalog Manager (CM), the Job Optimizer (JBO) and the Hadoop Automatic Configuration (HAC).

The CM is responsible for building and managing the catalog for historical jobs, data and system resources. We incrementally extract the catalog in a batch manner, and respond to the upper layer in real time. The statistics in the catalog are collected by the job profiler, the data profiler and the system profiler. The catalog query engine provides API for querying these statistics.

To optimize a MapReduce job, the JBO calls the Catalog Matcher to find the most similar job profile as well as related data and system information, and estimates the new job's profile with the Job Feature Estimation component. Then, the JBO estimates the running time of potential execution plans to find the optimal one.

The HAC is designed to advise the configuration time parameters (i.e. parameters that are fixed after the Hadoop instance gets started). The HAC covers MapReduce and HDFS.

5. IMPLEMENTATION

We implemented MRTuner on Hadoop 1.0.3 and 1.1.1. We first describe the implementation of the catalog, and then present how to use the PTC model to build the MapReduce optimizer with a fast search capability. Finally, we present the HAC.

5.1 MRTuner Catalog

The MRTuner catalog is defined as a set of statistics which can be extracted from Hadoop built-in counters [16], Hadoop configuration files and system profiling results.

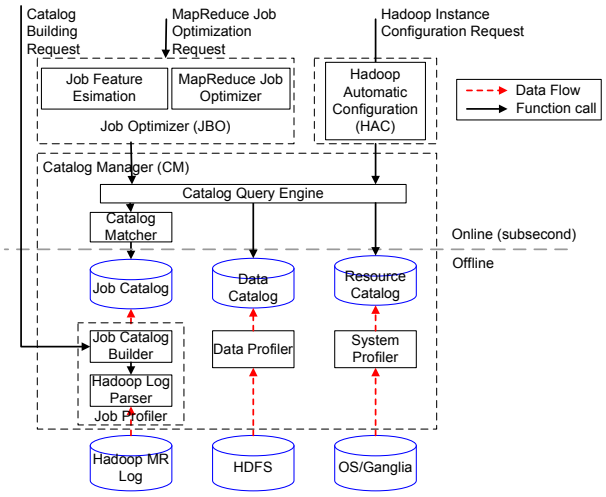


Figure 5: MRTuner Architecture

Table 1: The selected MRTuner catalog fields

Symbol	Description
Job Fields for a Job $\mathcal{J}$	
N_{mi}	Number of Map input records
N_{mo}	Number of Map output records
N_{ri}	Number of Reduce input records
N_{ro}	Number of Reduce output records
S_{mi}	Map input bytes
S_{mo}	Map output bytes
S_{ri}	Reduce input bytes
S_{ro}	Reduce output bytes
R_{mo}	Ratio of Map output compression
Data Fields for Data $\mathcal{D}$	
$\|\mathcal{D}\|$	Size of the data
S_B	Size of DFS blocks of the Data
E_{ik}	Length distribution of the input key
E_{iv}	Length distribution of the input value
System Resources Fields for a Cluster $\mathcal{C}$	
N_n	Number of machines
C_{lw}	Throughput of local disk write for each Map or Reduce slot
C_{lr}	Throughput of local disk read for each Map or Reduce slot
C_{cmpr}	Compression throughput of each Map or Reduce task slot
C_{decmpr}	Decompression throughput of each Map or Reduce task slot
C_{net}	Network throughput of the cluster
$C_{schedule}$	Overhead to create, schedule and destroy a Map or Reduce task.

Job, Data and Resource Profiles. Table 1 shows the fields of the catalog for jobs, data and system resources. We only list the selected fields that are required by the HAC and the JBO. As mentioned before, we assume that the allocated system resources for each Map or Reduce slot are the same. Thus $\{C_{lw}, C_{lr}, C_{cmpr}, C_{decmpr}\}$ in Table 1 are the average cost for each slot.

The job profile is built based on standard Hadoop logs to be compatible for the future upgrade of systems. To build the data profile, we detect the input format and data information of the MapReduce job from the historical job configuration files. The system resource profile is built based on `dd` like micro-benchmarks. We implement RRDtool[2] parser to collect Ganglia[3] data to compute the fields of system resources from the micro-benchmarking results.

The existing logs are processed in batch to build the catalog which will be incrementally updated as new logs coming in. We use the catalog meta-data to track Hadoop logs, data paths and system resources that have been processed. A daemon periodically collects new catalog information along the three dimensions.

Job Feature Estimation. Given a job $\mathcal{J}$ with input data $\mathcal{D}$, the Catalog Matcher is responsible for finding the most similar job profile from the catalog. The similarity of $\mathcal{D}$ and $\mathcal{D}'$ is measured by $Sim(\mathcal{D}, \mathcal{D}') = \frac{\mathcal{D} \cdot \mathcal{D}'}{|\mathcal{D}| \times |\mathcal{D}'|}$. The attributes of $\mathcal{D}$ are the data fields in Table 1.

Given a historical job profile $\mathcal{J}'$ and the input data $\mathcal{D}$, we estimate the features $\mathcal{F}$ of the newly submitted job $\mathcal{J}$. The feature set of the new job is defined in Table 2.We use a linear model to calculate the data properties in $\mathcal{F}$. For example, we have $f^m_{sel}(\mathcal{D}) = |\mathcal{D}| \cdot S'_{mo}/S'_{mi}$, where S'_{mo} and S'_{mi} are fields in the matched historical job profile $\mathcal{J}'$. $f_{cs}(\cdot)$ and $f_{copy}(\cdot)$ are obtained by the system profiling tool, which measures the context switch overhead and copy speed with varied number of concurrent copy threads. $f^m_{maxm}(\cdot)$ and $f^r_{maxr}(\cdot)$ are obtained based on the historical job configuration and the memory usage from Ganglia.

Implementation of the PTC Model. Given the catalog in Table 1 and job features in Table 2, we provide the implementation of functions in the PTC model as follows.

[2]http://oss.oetiker.ch/rrdtool/
[3]http://ganglia.info/

Table 2: The features $\mathcal{F}$ for a job $\mathcal{J}$ with input data $\mathcal{D}$

Symbol	Description
$f^m_{sel}(\cdot)$	Function for the size of Map output records
$f^m_{num}(\cdot)$	Function for the number of Map output records
$f^m_{maxm}(\cdot)$	Function for the maximum memory used by the Map Function
$f^r_{maxr}(\cdot)$	Function for the maximum memory used by the Reduce Function
$f^m_{rmc}(\cdot)$	Function for the running time of Map inputs read, user defined Map function and in-memory collect
$f^m_{cr}(\cdot)$	Function for the ratio of Map output compression
$f_{cs}(\cdot)$	Function for the context switch overhead
$f_{copy}(\cdot)$	Function for the copy speed of Reduce tasks

The running time of Map tasks is $t_{map}(\mathcal{D}, c) = f^m_{rmc}(\mathcal{D}) + f^m_{sel}(\mathcal{D}) \cdot (f^m_{cr}(\mathcal{D}) \cdot C_{lw} + C_{cmpr})$ if c is true. Otherwise, $t_{map}(\mathcal{D}, c) = f^m_{rmc}(\mathcal{D}) + f^m_{sel}(\mathcal{D}) \cdot C_{lw}$. Note that we should consider the overlapped Map tasks (equal to N_{ms}) in each wave. For example, the size of the data for $f^m_{rmc}(\mathcal{D})$ should be $\frac{|D|}{N_{ms}}$.

The context switch time is $t_{cs}(\mathcal{D}, c, v, r) = f_{cs}(f^m_{sel}(\mathcal{D}) \cdot f^m_{cr}(\mathcal{D}) \cdot \gamma, v, N_{rs})$ if c is true. Otherwise, $t_{cs}(\mathcal{D}, c, v, r) = f_{cs}(f^m_{sel}(\mathcal{D}) \cdot \gamma, v, N_{rs})$. The fraction γ is the percent of shuffled Map outputs which will lead to the context switch overhead. When the Shuffle phase is maximally overlapped, we have $\gamma = \frac{1}{r} \cdot \frac{m-1}{m}$.

The job scheduling time is $t_{schedule}(m) = m \cdot C_{schedule}$.

The local read and write time is $t_{lrw}(x) = \frac{|x|}{N_{ms}} \cdot (C_{lr} + C_{lw})$ and $t_{lrw}(x) = \frac{|x|}{N_{rs}} \cdot (C_{lr} + C_{lw})$, for Map and Reduce tasks respectively, where x is the amount of data to read and write.

The running time to process Reduce inputs without the Reduce side buffer is $t_{reduce}(\mathcal{D}, c) = \frac{D_s}{N_{rs}} \cdot C_{decmpr} + \frac{D_s}{N_{rs}} \cdot C_{lr}$ if c is true. Otherwise, $t_{reduce}(\mathcal{D}, c) = \frac{D_s}{N_{rs}} \cdot C_{lr}$.

5.2 MapReduce Job Optimizer

Given a MapReduce job with inputs and the running cluster, the goal of the job optimizer is to find the optimal setting of the performance sensitive parameters in Table 3. The description of each parameter can be found in [16].

5.2.1 *From the PTC Model to Parameters*

Based on the four key factors of the PTC model, we can derive the optimal values for other performance sensitive parameters.

Table 3: The List of Targeted Parameters

Symbol	Parameter	Groups
N_m	mapred.map.tasks	Map input split
N_r	mapred.reduce.tasks	Reduce output
$s_{\min}$	mapred.min.split.size	Map input split
$s_{\max}$	mapred.max.split.size	Map input split
B_m	io.sort.mb	Map output buffer
r_{rec}	io.sort.record.percent	Map output buffer
c	mapred.compress.map.output	Map output compr.
N_{copy}	mapred.reduce.parallel.copies	Reduce copy
N_{sf}	io.sort.factor	Reduce input
B_r	mapred.job.reduce.input.buffer.percent	Reduce input
S_{OB}	dfs.block.size	Reduce output
$N_{ms}(i)$	mapred.tasktracker.map.tasks.maximum	Set by HAC
$N_{rs}(i)$	mapred.tasktracker.reduce.tasks.maximum	Set by HAC

In addition, the relationship among parameters described by the model below can help further narrow down the search space as described in Section 5.2.2.

Map Output Buffer. We first model the impact of parameters related to the Map output buffer. As shown in Table 3, we denote the buffer for sorting files (io.sort.mb) as B_m, the fraction for meta-data (io.sort.record.percent) as r_{rec}, the spilling threshold (io.sort.spill.percent) as r_{spill} and the JVM heap size as B_j. Since $B_m < B_j - f^m_{maxm}(\mathcal{D})$, the available memory buffer B_{ma} to store Map output records and their meta-data should satisfy the following condition.

$$B_{ma} \leq B_m \cdot r_{spill} \leq (B_j - f^m_{maxm}(\mathcal{D})) \cdot r_{spill} \tag{5}$$

Since each Map output record has 16 byte as the meta-data field, we have

$$r_{rec} = \frac{16}{16 + R}, \tag{6}$$

where R is the average length of Map output records. Given the job feature $f^m_{num}(\cdot)$ in Table 2, we have $R = \frac{|\mathcal{D}|}{f^m_{num}(\mathcal{D})}$.

Map Input Split. We denote the size of the Map input split as s, and we have

$$s = \frac{|\mathcal{D}|}{m \cdot N_{ms}} \tag{7}$$

To avoid the I/O overhead caused by additional spills when the Map buffer B_{ma} can not keep all the Map outputs, we adjust s. Then, the size of each Map task output D_{mo} should be less than the allocated buffer for the data field of Map outputs, that is,

$$D_{mo} = \frac{f^m_{sel}(\mathcal{D})}{|\mathcal{D}|} \cdot s \leq B_{ma} \cdot (1 - r_{rec}) \tag{8}$$

Based on Eq 5, Eq 6, Eq 7 and Eq 8, we derive the lower bound of m as follows.

$$m \geq \frac{f^m_{sel}(\mathcal{D}) \cdot (16 + R)}{N_{ms} \cdot (B_j - f^m_{maxm}(\mathcal{D})) \cdot R \cdot r_{spill}} \tag{9}$$

Then, given the number of Map waves m, B_m can be derived as

$$\begin{aligned} B_m \geq \frac{B_{ma}}{r_{spill}} &\geq \frac{f^m_{sel}(\mathcal{D}) \cdot s}{r_{spill} \cdot |\mathcal{D}| \cdot (1 - r_{rec})} \\ &= \frac{f^m_{sel}(\mathcal{D})}{N_{ms} \cdot m \cdot r_{spill} \cdot (1 - r_{rec})} \end{aligned} \tag{10}$$

Reduce Output. When the Map input split size is not equal to the block size of the input file, it may result in the data locality problem that incurs extra network overheads. The data is allocated to nodes in terms of blocks by HDFS, and there is strictly one-to-one mapping between Map tasks and input splits, then if the split size is not equal to or is not a factor of the input file block size (denoted as S_B), some nodes may copy data from the remote to prepare input splits for Map tasks. Then, to eliminate this data movement overhead, the input split should satisfy:

$$S_B \mod (\frac{|\mathcal{D}|}{m \cdot N_{ms}}) = 0 \tag{11}$$

We can further optimize the Reduce output block size S_{OB} to be multiples of the optimal Map split size of the most frequent consumer job (i.e. the most frequent job that processes the Reduce output file of the current job as Map inputs).

Reduce Input. Based on the estimation of the maximum memory usage of the user defined Reduce function $f^r_{maxr}(\mathcal{D})$, the Reduce side buffer B_r is given as follows.

$$B_r = B_j - f^r_{maxr}(\mathcal{D}) \tag{12}$$

When the amount of data in the buffer exceeds the threshold mapred.job.shuffle.merge.percent (denoted as r_{mg}), data will be spilled to disk. Those disk files will be merged progressively whenever the number of such files exceeds the merge factor threshold (i.e. io.sort.factor). To ensure that the sort and merge can be completed in one pass, we set merge factor N_{sf} as below.

$$N_{sf} = \frac{D_s}{N_r \cdot B_S \cdot r_{mg}}, \tag{13}$$

where B_S is the Shuffle buffer determined by the parameter mapred.job.shuffle.input.buffer.percent.

5.2.2 *Finding the Optimal Execution Plan*

In this section, we present the method to find the optimal parameter setting for a MapReduce job. The basic idea is to find the optimal setting of the four key factors based on the cost model in Eq 4, and derive other performance sensitive parameters based on the four factors.

To further reduce the search space, we derive ranges of the factors that may have large space of candidates. Note that c only has two values: true or false, that is, $\mathbb{C} = \{true, false\}$. The number of values for v is also limited. It depends on the parameter N_{copy}. We have $\mathcal{V} = \{v | v = f_{copy}(N_{copy} \cdot N_{ns}), 1 \leq N_{copy} \leq N_{copy\max}\}$. Next we focus on the range of m and r.

Range of m. Based on the Proposition 3.2, the maximum amount of Map outputs that can be transferred in the overlap manner is

$$D'_{o\max} = \lim_{m \to \infty} \frac{m - 1}{m} \cdot f^m_{sel}(\mathcal{D}) = f^m_{sel}(\mathcal{D}) \tag{14}$$

If the saved running time by the overlapped Shuffle phase is less than that to schedule increased tasks, there is no need to increase m. Thus we have the upper bound of m as

$$m \leq \frac{f^m_{sel}(\mathcal{D})}{C_{net} \cdot C_{schedule}} \tag{15}$$

Based on Eq 9 and Eq 15, we obtain the following lemma.

LEMMA 5.1. *Given the inputs $\mathcal{D}$, system resources $\mathcal{C}$ and job feature functions $\mathcal{F}$, the number of Map waves m is bounded by*

$$\begin{aligned} \mathcal{M} = \{m | &\frac{f^m_{sel}(\mathcal{D}) \cdot (16 + R)}{N_{ms} \cdot (B_j - f^m_{maxm}(\mathcal{D})) \cdot R \cdot r_{spill}} \leq m \\ &\leq \frac{f^m_{sel}(\mathcal{D})}{C_{net} \cdot C_{schedule}}\} \end{aligned} \tag{16}$$

Range of r. Based on the Proposition 3.3, the maximum amount of Reduce inputs that can be hold in memory is the total Map outputs $f^m_{sel}(\mathcal{D})$. Since the saving of the disk I/O is at the cost of the additional task scheduling overhead, there is no need to increase r if all the Reduce inputs can be kept in memory. Then we have $r \leq \frac{f^m_{sel}(\mathcal{D})\cdot(C_{lr}+C_{lw})}{C_{schedule}\cdot N_{rs}}$. Thus we obtain the following lemma.

LEMMA 5.2. *Given the inputs $\mathcal{D}$, system resources $\mathcal{C}$ and estimated job feature functions $\mathcal{F}$, the number of Reduce waves r is bounded by*

$$\mathcal{R} = \{r | 1 \leq r \leq \frac{f^m_{sel}(\mathcal{D}) \cdot (C_{lr} + C_{lw})}{C_{schedule} \cdot N_{rs}}\} \tag{17}$$

Algorithm 1 PTC-Search ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{P}$)

1: $C_{\max} \leftarrow \infty$
2: **for** each factor set $\{m, r, v, c\} \in \mathcal{M} \times \mathcal{R} \times \mathcal{V} \times \mathbb{C}$ **do**
3: $\quad C \leftarrow \mathcal{T}_{producer} + \mathcal{T}_{transporter} + \mathcal{T}_{consumer}$ ▷ Eq 4
4: $\quad$ **if** $C < C_{\max}$ **then**
5: $\quad\quad \{m_o, r_o, v_o, c_o\} \leftarrow \{m, r, v, c\}$
6: $r_{rec} \leftarrow 16/(16 + \frac{|\mathcal{D}|}{f^m_{num}(\mathcal{D})})$
7: $B_m \leftarrow \frac{f^m_{sel}(\mathcal{D})}{N_{ms}\cdot m \cdot r_{spill}\cdot(1-r_{rec})}$ ▷ Eq 10
8: $s \leftarrow \frac{|\mathcal{D}|}{m \cdot N_{ms}}$ ▷ Eq 7
9: **if** $s < S_B$ **then** $s_{\max} \leftarrow s$
10: **if** $s > S_B$ **then** $s_{\min} \leftarrow s$
11: $N_r \leftarrow r \cdot N_{rs}$
12: $B_r \leftarrow B_j - f^r_{maxr}(\mathcal{D})$ ▷ Eq 12
13: $N_{sf} \leftarrow \frac{D_s}{N_r \cdot B_S \cdot P_m}$ ▷ Eq 13
14: $N_{copy} \leftarrow f^{-1}_{copy}(v)$
15: $S_{OB} \leftarrow k \cdot s'$ of the most frequent consumer job J'
16: **return** the parameter setting P_o

Finally we give the search method in Algorithm 1. In the first step, according to the cost model in Section 3.2, we find the optimal setting of the four key factors with derived ranges. In the second step, other parameters are figured out based on the models in Section 5.2.1. In the implementation, the candidates and bounds of each parameter are further determined by a rule-based engine. Since there are only four factors with derived ranges need to search, the PTC-Search is very efficient. More importantly, given the relationship in Section 5.2.1, the parameter setting is optimal.

5.3 Hadoop Automatic Configuration

The HAC component is designed to automatically configure static parameters of MapReduce and HDFS (i.e. parameters shared by all jobs, for example, task slots).

MapReduce Automatic Configuration. The goal of MapReduce configuration is to determine the task slot capacity and the available memory for each Map or Reduce task. We model the problem by a non-linear programming problem to maximize memory utilization with constraints on CPU utilization and concurrent write threads.

HDFS Automatic Configuration. The HAC configures HDFS based on the overall historical workload behaviors. The main problem of configuring HDFS is how to allocate local disks for HDFS and MapReduce temp directory. The allocation depends on the workload submission pattern. For the Hadoop cluster that often runs hybrid jobs, we separate the two directories in different disks to avoid competitions. On the contrary, if a significant portion of jobs are running alone, we allocate all the disks to both HDFS and MapReduce temp directory to improve the disk throughput. The detection of the job submission pattern is based on the submission and the completion trace from standard Hadoop logs.

6. RELATED WORK

Hadoop Self-tuning and Models. Starfish is a self-tuning system for MapReduce [6, 7]. The authors propose the *profile-prediction-optimization* approach to optimize parameters for MapReduce jobs. While Starfish is the pioneer work on MapReduce job optimization, it does not well address the issues of inter-task parallelization and parameter reduction. Wang [15] proposes a simulation approach to provide fine-grained simulation of MapReduce at the sub-phase level. However, it only models single node processes such as the task processing time, and lacks the capturing of the cost that occurs between nodes, which is modeled by our PTC model. Scalla [11] proposes an analytical model for Hadoop. However, Scalla focuses on the I/O cost and the task startup cost, and does not model the running time. In addition, Scalla does not address the inter-task parallelization problem.

Parallel Computation Models. Bandwidth-latency models such as the LogP model [3] and the BSP model [14] are a group of performance models for parallel programs that focus on modeling the communication between the processes in terms of network bandwidth and latency, allowing quite precise performance estimations. However, since the MapReduce platform is not a typical message-based system, it is difficult to apply those models to the MapReduce platform directly.

Cloud-based Auto-tuning. Another class of related works comes from the world of cloud infrastructures. A plenty of research exists to monitor the workload and system events to manage and auto-tune the platform, including database multi-tenancy [4], resource orchestration [5], performance modeling [13], etc. For example, Delphi [4] provides a self-managing system controller for multi-tenant DBMS. However, these analytic models do not translate well for Hadoop self-tuning, due to the characteristics of Big Data [2] and the user defined function of the MapReduce workload [7]. In this paper, MRTuner provides an analytical model based on the characteristics of MapReduce and platform internal mechanisms.

7. EVALUATION

In this section, we conduct two groups of experiments to compare MRTuner with a commercial enterprise Hadoop and with Starfish under a variety of input data, jobs and clusters. In the first group, we evaluate the response time of MapReduce optimizers. In the second group, we demonstrate the effectiveness of MRTuner from the perspectives of both job elapsed time and system resource utilization, by using workloads with different characteristics. More importantly, we dissect the performance improvements through parameter optimizations on selected workloads to elaborate why the tuning of MRTuner works.

7.1 Experiment Setup

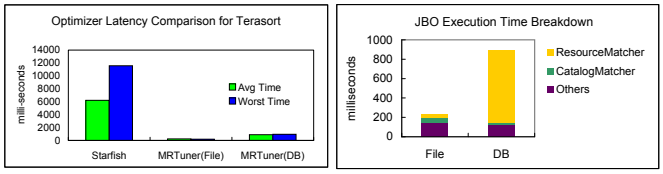
We deploy two clusters to evaluate MRTuner in scenarios where the system resource configurations are different.

- Cluster $\mathcal{A}$: The Hadoop cluster with ten machines is deployed. For the Name Node and Job Tracker, we use the HS21 blade with 4×3.66GHz Processors and 4 GB memory. For other nine nodes, we use the HS20 blade with 4×1.66GHz Processors and 4 GB memory.
- Cluster $\mathcal{B}$: The Hadoop cluster of x3650 servers is deployed. There are totally 128 2.9GHz CPU cores, 880 GB memory and 36 TB local disk.

Both clusters are deployed upon the 1 $Gbps$ Ethernet switch. The Hadoop version is 1.0.3. The JVM is IBM J9 VM (build 2.4, JRE 1.6.0). We use Ganglia to monitor the cluster.

Table 4: The Parameter Settings

Parameters	Hadoop-X on $\mathcal{A}$	Hadoop-X on $\mathcal{B}$	TS-1	TS-2	TS-3	TS-4	NG-1	NG-3	NG-5	NG-6	PR-1	PR-3	PR-4	PR-6
N_{ms}	30	128	29	29	118	118	29	29	118	118	29	29	118	118
N_{rs}	15	64	29	29	98	98	29	29	98	98	29	29	98	98
N_r	10	60	29	29	90	90	29	29	90	90	29	29	90	90
$s_{\min}$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
$s_{\max}$	N/A	N/A	N/A	N/A	N/A	N/A	4m	4m	8m	8m	N/A	N/A	N/A	N/A
B_m	256	256	194	322	819	819	322	322	480	480	322	322	161	195
r_{rec}	0.05	0.05	0.138	0.138	0.138	0.138	0.296	0.296	0.296	0.296	0.091	0.076	0.091	0.075
c	false	false	true	true	true	true	true	true	true	true	true	true	true	true
N_{copy}	5	5	1	1	1	1	1	1	1	1	1	1	1	1
N_{sf}	10	10	10	10	30	10	10	20	10	20	10	10	10	10
B_r	0	0	0.7	0.7	0.7	0.7	0.5	0.7	0.5	0.7	0.4	0.3	0.7	0.7
S_{OB}	128m	128m	64m	64m	128m	128m	64m	128m	128m	128m	128m	64m	128m	128m



(a) JBO Time Comparison (b) JBO Time Breakdown

Figure 7: The Execution Time of Optimizers

Our benchmarking suite is derived from HiBench [8], which includes both micro-benchmarks and real-world Hadoop applications. We present the results of Terasort for micro-benchmark, Text Classification for Text Mining and Pagerank for Social Network Analysis. MRTuner has been validated in some customer cases and internal product integration. MRTuner achieves similar performance improvements in these cases. For the purpose of results reproducing, we focus on the publicly available HiBench in this paper. However, we would like to emphasize some memory intensive workloads in real production. For these memory intensive workloads, MRTuner is able to measure the memory usage of the user defined Map and Reduce functions (covered by the $f^m_{mmax}(\cdot)$ and $f^r_{rmax}(\cdot)$ estimations) to avoid out of memory issues.

In our evaluation, the following related work is compared with MRTuner:

- *Default Hadoop.* The execution time of jobs under the default Hadoop configuration is an order of magnitude slower than that of MRTuner optimized for almost all the workloads, so the results are not shown.

- *Hadoop-X.* Hadoop-X is a commercial enterprise Hadoop offering with built-in automatic configuration capabilities. Hadoop-X is able to automatically configure the task slot capacity based on the system resources. Hadoop-X also has a set of pre-configured performance sensitive parameters (See Table 4 for details). Besides these parameters, we also tune the number of Map and Reduce tasks to reasonable settings for each job for Hadoop-X.

- *Starfish.* We compare MRTuner with the cost-based MapReduce optimizer of Starfish 0.3.0[4]. However, we can only run a few micro-workloads like Terasort. Other workloads such as Pagerank and text classification are not supported by the Starfish that we can download.

[4]http://www.cs.duke.edu/starfish/

Table 5: The Comparison between Hadoop-X and MRTuner

JobName	ID	Cluster	Input (GB)	Hadoop -X(sec)	MRTuner (sec)	Speed -up
Terasort	TS-1	$\mathcal{A}$	10	469	278	1.7
Terasort	TS-2	$\mathcal{A}$	50	2109	1122	1.87
Terasort	TS-3	$\mathcal{B}$	200	767	295	2.60
Terasort	TS-4	$\mathcal{B}$	1000	6274	2192	2.86
N-Gram	NG-1	$\mathcal{A}$	0.18	4364	192	22.7
N-Gram	NG-2	$\mathcal{A}$	0.7	N/A	661	∞
N-Gram	NG-3	$\mathcal{A}$	1.4	N/A	1064	∞
N-Gram	NG-4	$\mathcal{B}$	1.4	1100	249	4.41
N-Gram	NG-5	$\mathcal{B}$	2.8	1292	452	2.86
N-Gram	NG-6	$\mathcal{B}$	5.6	1630	930	1.75
PR(Trans.)	PR-1	$\mathcal{A}$	3.23	962	446	2.2
PR(Deg.)	PR-2	$\mathcal{A}$	Inter	49	41	1.2
PR(Iter.)	PR-3	$\mathcal{A}$	Inter	933	639	1.5
PR(Trans.)	PR-4	$\mathcal{B}$	3.23	148	65	2.28
PR(Deg.)	PR-5	$\mathcal{B}$	Inter	24	22	1.09
PR(Iter.)	PR-6	$\mathcal{B}$	Inter	190	82	2.32

7.2 Latency of MRTuner JBO

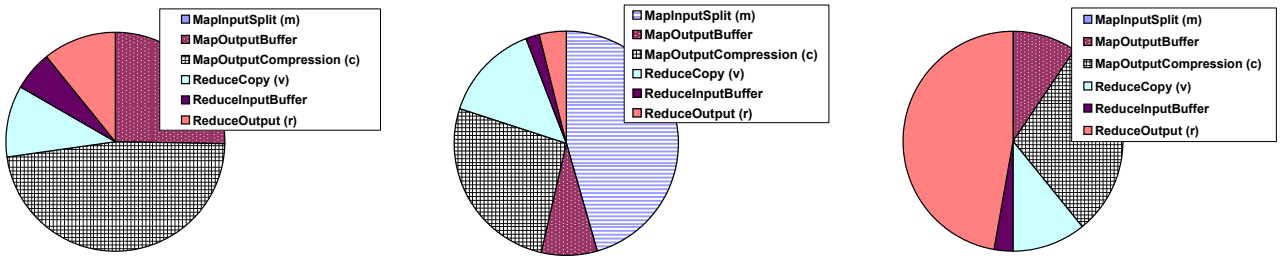
We implement both file-based and DB-based catalogs. Figure 7 shows the execution time of JBO. Note that the latency does not include the catalog building time, which is the off-line overhead.

Catalog Matcher. The file-based catalog makes use of an index file (pre-generated by the Catalog Builder) to retrieve similar jobs for the Catalog Matcher. In the DB-based catalog, the latency of this part is lower due to the indexing and caching of the database. The average latency of the file-based catalog matching is 57 milliseconds, and that of the DB-based catalog is 19 milliseconds.

Resource Matcher. For the file-based catalog, the query of system resources is based on an XML file. The DB-based catalog uses multiple tables loaded from Ganglia RRDtools to get such information by join operations, which brings higher latency. The average latency for resource matching is 37 milliseconds for the file-based catalog and 747 milliseconds for the DB-based catalog.

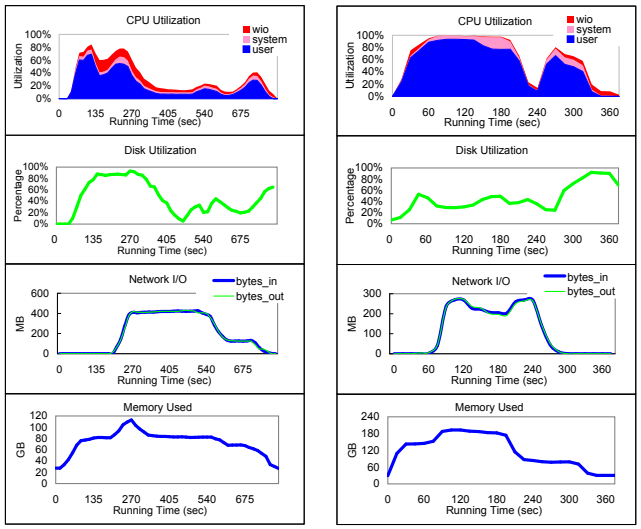
Cost Model Optimization Searching. Since there are only four factors in the PTC model, the number of candidates is as small as $1000 - 10000$ for the evaluated workloads, which reduces the search time by four orders of magnitude of reduction compared to Starfish optimizers [6]. The response time of the PTC search is within 1 millisecond for all the evaluated cases. As shown in Figure 7, the end-to-end response time of MRTuner JBO achieve 25 times faster than that of Starfish, mainly because of the reduction of the search space from 20+ dimensions to 4 dimensions.

For jobs whose input data is not pre-existing, we need to get



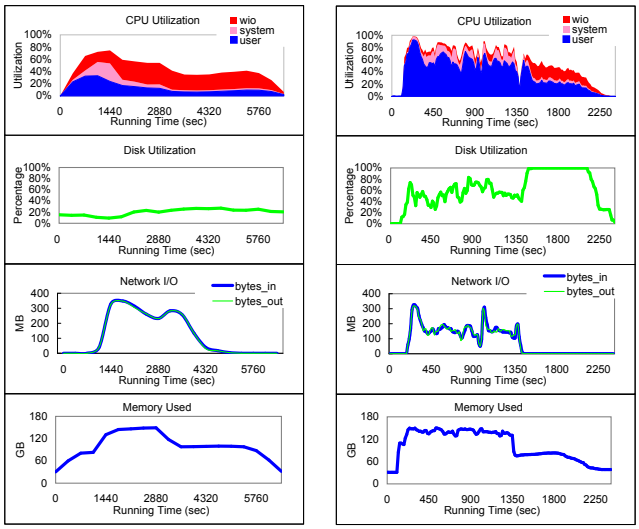
(a) TS-3 (b) NG-5 (c) PR-4

Figure 6: Impact of Parameter Groups on Selected Jobs



(a) Hadoop-X (b) MRTuner

Figure 8: Resource Utilization of TS-3



(a) Hadoop-X (b) MRTuner

Figure 9: Resource Utilization of TS-4

its data catalog on the fly, which leads to about 650 milliseconds latency. In such cases, we only use the data size and block size properties of the data, and MRTuner still performs a factor of 6 faster than Starfish on the job optimizer response time.

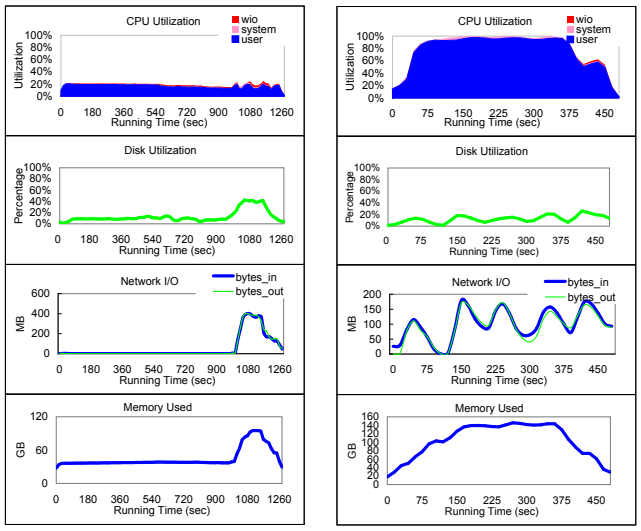
7.3 MRTuner Effectiveness

To evaluate the effectiveness of MRTuner, we show not only the running time, but also system resource utilization, and elaborate why MRTuner outperforms the state-of-the-art MapReduce optimizer, especially focusing on the inter-task optimizations of the PTC model. For typical workloads, we give the quantitative analysis on how each of the parameter (group) affects the performance. Both Hadoop-X and MRTuner optimized settings are shown in Table 4. The running time of these workloads is listed in Table 5.

Terasort. Terasort is a standard benchmark created by Jim Gray [12]. The input data is generated by the *TeraGen* program. The elapsed time of Terasort is shown in Table 5. MRTuner achieves the job elapsed time of 2 and 3 times faster than Hadoop-X on cluster $\mathcal{A}$ and $\mathcal{B}$, respectively. Figure 8 and Figure 9 shows cluster-wide system resource utilization of TS-3 (200 GB) and TS-4 (1 TB) respectively. The figures indicate that MRTuner achieves much better CPU utilization, less context switch overhead (indicated by the system CPU time), better memory utilization and less network overhead compared with Hadoop-X.

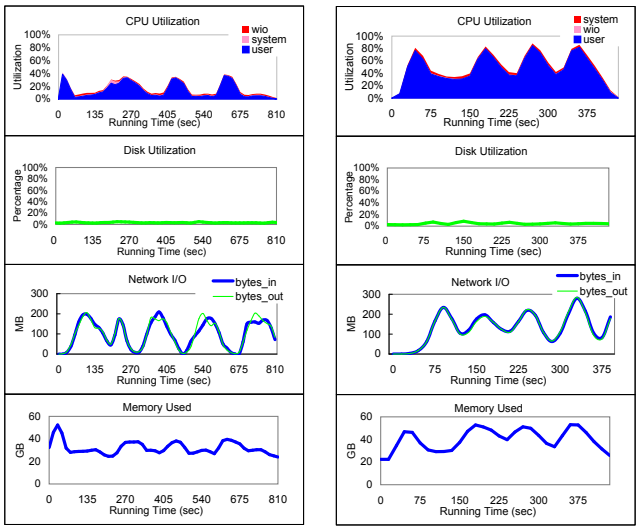
We analyze the impact of each parameter group for TS-3 in Figure 6 (a) (See Table 3 for parameter groups). The ratio in Figure 6 is obtained by setting the related parameters to the Hadoop-X setting. For example, to evaluate the impact of Map output buffer group, we change related parameters from the MRTuner setting to the Hadoop-X setting (i.e. $B_m = 256$ and $r_{rec} = 0.05$) to get the running time t'. Suppose that the job elapsed time of MRTuner is t. The impact of this parameter group is measured as $\frac{t'-t}{t}$. Finally, we normalize all the ratio of impacts in a pie chart. Figure 6 (a) indicates that the Map output compression, the Map output buffer and the Reduce output are the top three influencing factors for the workload TS-3.

Next we dissect optimizations on TS-3 to elaborate why MRTuner outperforms Hadoop-X. First, the task slot optimization of MRTuner considers cluster-wide system resources. The non-linear programming model in Section 5.3 figures out the optimal configuration. The main contribution of this optimization is on the Reduce output. The increased parallelism improves 21.2% of the elapsed time compared with Hadoop-X. Second, based on the estimation of map output properties (i.e. the size and the number of records), MRTuner ensures that there is no additional disk spills before Map tasks finish, which improves the job elapsed time by 26.99% over Hadoop-X. (The disk read and write size of Hadoop-X is a factor of 2.99 more than that of MRTuner, due to the additional spills.) Similarly, based on the estimation of Reduce input properties, MRTuner is able to allocate 70% of the JVM heap as the Reduce input buffer, which improves the job elapsed time by 6.27% over Hadoop-X. Third, the PTC model of MRTuner ensures the maximal overlap-



(a) Hadoop-X (b) MRTuner

Figure 10: Resource Utilization of NG5



(a) Hadoop-X (b) MRTuner

Figure 11: Resource Utilization of Pagerank Workflow on $\mathcal{B}$

ping of the Map and Shuffle phases, while avoiding contention of copy threads. The Reduce copy related optimization improves the job elapsed time by 11.25% over Hadoop-X. In addition, MRTuner decides that the compression should be enabled for Map outputs for the evaluated Terasort workloads. Without compression, 70.3% of Map outputs can not be copied to the Reduce side when all the Map tasks complete, and the non-overlapped Reduce copy time leads to 50.52% of performance reduction although the compression and decompression overhead is saved.

Text Classification. The Mahout Bayes based text classification [8] is a workload with more than ten MapReduce jobs. The second job, which is to tokenize the words and n-grams in a document, accounts for more than half of the total execution time. Thus the analysis below is focused on this job. The job elapsed time of this workload is shown in Table 5. For NG-1, the job elapsed time of MRTuner is a factor of 22.7 faster that of Hadoop-X. For NG-2 and NG-3, Hadoop-X runs for more than three hours and still can not finish, and we observe a lot of disk errors caused by the overwhelming data spills in the Map phase (because of limited disk space of the cluster $\mathcal{A}$ and the expanded Map spills). Such situation would not happen in MRTuner, and jobs of this workload can finish in significantly lower latency in MRTuner. Because MRTuner eliminates the intermediate data spills effectively. NG-4, NG-5 and NG-6 show only 1.75-4.41x speedups. Because Cluster $\mathcal{B}$ has sufficient disk space, and the disk read/write capacity in Cluster $\mathcal{B}$ is significantly higher than that in Cluster $\mathcal{A}$. Figure 10 is cluster-wide system resource utilization of NG-5. It indicates that MRTuner obtains much better CPU and Memory utilization compared with Hadoop-X.

We dissect optimizations on NG-5 to elaborate why MRTuner outperforms Hadoop-X. First, MRTuner optimization eliminates the additional disk spills in the Map stage. It is realized based on the estimation of map output properties. Note that without this optimization, some jobs can not be completed in Cluster $\mathcal{A}$. MRTuner splits inputs based on the Map Output/Input ratio (456 map tasks for NG-5) instead of splitting input based on the block size (22 map tasks for NG-5). A side effect of this splitting is that MRTuner increases the number of simultaneously running Map tasks, which leads to much higher CPU utilization (see Figure 10). Second, the compression of Map outputs also brings significant performance gain (35%). Finally, MRTuner avoids copier thread contention by estimating the throughput of Map output generation. Reducing the number of copier threads to 1 obtains about 20% speed-up compared with the Hadoop-X setting.

The impact of each parameter group settings is shown in Figure 6(b) for the NG-5 workload. We find that it is difficult to perform the thorough parameter impact analysis for NG-2 and NG-3. Because for many parameter settings, the workloads can not run to completion due to their larger intermediate data volume compared with the available disk space in Cluster $\mathcal{A}$. In addition, NG-4 and NG-6 are all similar with NG-5 in the performance improvement breakdown, showing that the Map Input Split and the Map Output Compression are the top two influencing factors. Actually, the impact of Map Input Split for NG-1 is significantly higher (80%), due to the significantly lower disk transfer rate in Cluster $\mathcal{A}$. In addition, different with NG-5, the impact of the Reduce Copy for NG-1 is almost 40%. Note that the major performance sensitive parameters of the text classification are significantly different from that of Terasort, which shows different characteristics of the two workloads.

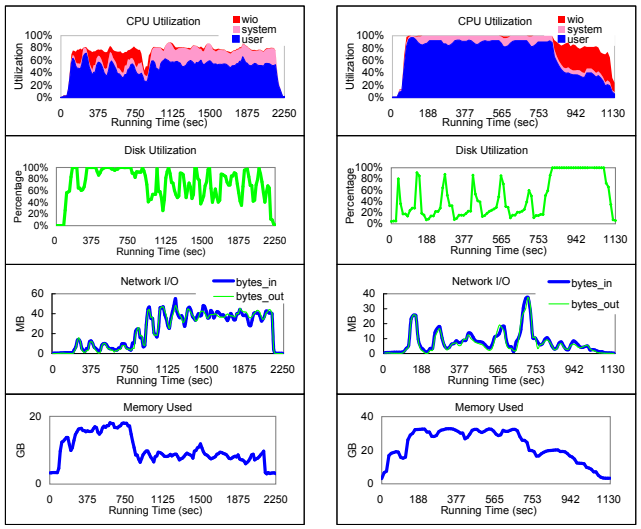
Pagerank. We use X-RIME [17] implementation of Pagerank and publicly available social network data[5]. The directed user interaction graph has $6,034,780$ vertices and $52,614,182$ arcs. We set the number of Pagerank iterations to 3, and use the default Pagerank damping factor 0.85. We use the Java interface of MRTuner to dynamically generate parameters for each iterative MapReduce job. Table 5 shows the elapsed time of MapReduce jobs. The result of Pagerank is the average of iterative steps. For the workloads on the cluster $\mathcal{A}$, we obtain $2.2x$ speedup on the transformation job (transforming raw graph data to the adjacency list), $1.2x$ speedup on the degree calculation job, and averagely 1.5x speedup on the iterative Pagerank step jobs. For the workloads on the cluster $\mathcal{B}$, we obtain 2.3x, 1.1x and 2.3x speedups for the three jobs, respectively. Figure 11 is cluster-wide system resource utilization of all jobs of the Pagerank workflow on the cluster $\mathcal{B}$. It indicates that MRTuner obtains better CPU and memory utilization. The impact of each parameter group on PR-4 in the cluster $\mathcal{B}$ is shown in Figure 6 (c). It

[5]http://current.cs.ucsb.edu/facebook/index.html

indicates that the Reduce output, the Map output compression and the Reduce copy are the top three influencing factors for PR-4.

Figure 11 (b) further indicates that the Reduce phases of both PR-4 and PR-6 have downward trends in terms of CPU utilization. Through profiling Reduce tasks, we find that it is caused by data skew [9]. The replication in HDFS writing further amplifies the impact of data skew.

Although the skew issue is hardly to be addressed by parameter tuning, it will affect the PTC model. For example, skew of Map tasks may increase the overlapped time duration for the Shuffle stage. As future work, we are extending the PTC model to estimate task skew based on the historical task execution time.



(a) Starfish (b) MRTuner

Figure 12: Resource Utilization of TS2

7.4 Comparison with Starfish

We deploy Starfish 0.3.0 in the cluster $\mathcal{A}$, but fail to configure it in the cluster $\mathcal{B}$ due to JVM issues. Since the version which we download only supports a few micro-workloads such as Terasort and Word Count, we only compare these workloads that are available in Starfish. The latency of JBO is compared in Section 7.2.

We present the effectiveness of job optimizers. For TS-2, the job elapsed time of MRTuner is a factor of 1.6 faster than that of Starfish. Figure 12 shows system resource utilization of Starfish and MRTuner on TS-2, which indicates that MRTuner has better system resource utilization. When we compare the suggested execution plans, we find that more network I/O of MRTuner are overlapped with the Map stage. Also, there is no additional disk I/O in Map outputs using the MRTuner generated parameters for TS-2. Finally, there is much less context switch overhead since the PTC model optimizes tasks among Map and Reduce stages.

8. CONCLUSION

In this paper, we proposed the MRTuner toolkit to enable holistic optimization for MapReduce jobs. The Producer-Transporter-Consumer model was proposed to estimate the tradeoffs in MapReduce execution plans. The relationship among parameters was derived to develop a fast search algorithm. The experimental evaluation based on HiBench demonstrated the effectiveness and efficiency of MRTuner. As future work, we are extending the PTC model in YARN to address the multi-tenancy and skew issues.

9. ACKNOWLEDGMENTS

The authors would like to thank Ning Yang, Zeli Liu and Zelin An from RUC, and Zhongxin Guo from BUPT, for implementing the initial prototype of the catalog. The authors also thank anonymous reviewers for their valuable comments and suggestions to improve the paper. Jiaheng Lu is partially supported by NSF China (No. 60903056), 863 National High-tech Research Plan of China (No. 2012AA011001), IBM-RUC research funds and the Fundamental Research Fund for RUC.

10. REFERENCES

[1] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[2] S. Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In *PODS*, pages 1–4, 2012.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[4] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *SIGMOD*, pages 517–528, 2013.

[5] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for ”autonomic” orchestration. In *VTDC*, page 7, 2006.

[6] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[7] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[8] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDEW*, pages 41–51, 2010.

[9] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, pages 25–36, 2012.

[10] J. K. Laurila, D. Gatica-Perez, I. Aad, O. Bornet, T.-M.-T. Do, O. Dousse, J. Eberle, and M. Miettinen. The mobile data challenge: Big data for mobile computing research. In *Proc. of Nokia Mobile Data Challenge Workshop*, 2012.

[11] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996, 2011.

[12] O. OMalley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *Sort Benchmark*, 2009.

[13] N. Park, I. Ahmad, and D. J. Lilja. Romano: autonomous storage management using performance prediction in multi-tenant datacenters. In *SoCC*, pages 1–14, 2012.

[14] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[15] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, pages 1–11, 2009.

[16] T. White. *Hadoop: The Definitive Guide*. O’Reilly, 2012.

[17] W. Xue, J. Shi, and B. Yang. X-rime: cloud-based large scale social network analysis. In *SCC*, pages 506–513, 2010.