PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs

by

Mostafa Ead

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The MapReduce programming model has become widely adopted for large scale analytics on big data. MapReduce systems such as Hadoop have many tuning parameters, many of which have a significant impact on performance. The map and reduce functions that make up a MapReduce job are developed using arbitrary programming constructs, which makes them black-box in nature and prevents users from making good parameter tuning decisions for a submitted MapReduce job. Some research projects, such as the Starfish system, aim to provide automatic tuning decisions for input MapReduce jobs. Starfish and similar systems rely on an execution profile of a MapReduce job being tuned, and this profile is assumed to come from a previous execution of the same job. Managing these execution profiles has not been previously studied. This thesis presents PStorM, a profile store that organizes the collected profiling information in a scalable and extensible data model, and a profile matcher that accurately picks the relevant profiling information even for previously unseen MapReduce jobs. PStorM is currently integrated with the Starfish system, providing the necessary profiles that Starfish needs to tune a job. The thesis presents results that demonstrate the accuracy and efficiency of profile matching. The results also show that the profiles returned by PStorM lead to Starfish tuning decisions that are as good as the decisions made by profiles collected from a previous run of the job.

Acknowledgements

First, I would like to thank Ashraf Aboulnaga, my supervisor, for guiding me through my graduate studies. Thanks Ashraf for your advice at the academic, technical, personal, and career levels.

I would also like to thank Shivnath Babu and Herodotos Herodotou, the co-authors of the Starfish system, for sharing their insights about Hadoop performance optimization approaches, and for their fruitful discussions about PStorM.

I would like to thank the NSERC Business Intelligence Network for providing funding for this work.

A special thanks to my lovely wife, Yassmin Taher, for her support for me being a successful man, for her sharing of my presentation rehearsals, and for being the wonderful mother of my little cute baby, Sandra.

Finally, I owe sincere thankfulness to my parents, for their continual support and encouragement since my early childhood till the current moment.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The MapReduce (*MR*) programming model [6] has become widely adopted for large scale data analytics in many organizations. MapReduce systems, the most popular being Hadoop [11], have many tuning parameters, and these parameters have a significant impact on performance. Setting these tuning parameters is difficult even for expert users. And as Hadoop and its ecosystem get more widely adopted in diverse application domains, more users from diverse backgrounds become involved in the development of MR jobs. These users may be experts in their application domains, but they are likely novices when it comes to Hadoop performance tuning. Thus, it is important to develop automatic tuning techniques for MR jobs in Hadoop, especially since these jobs often run on clusters of hundreds or thousands of machines, so any wasted resources due to poor tuning decisions will be amplified by the size of the cluster.

Tuning MR jobs is difficult due to the complexity and scale of the software and the underlying clusters on which these jobs run. The MapReduce model places no restrictions on the types of data that jobs process, and the *map* and *reduce* functions that process the data are also unrestricted in their complexity. Moreover, data storage and processing happens in a large-scale distributed system with possibly heterogeneous hardware. An effective way to deal with the complexity of MapReduce tuning is to adopt a *feedback-based* approach to tuning. In this approach, the system collects information about the execution of MapReduce jobs in the form of *execution profiles*. The system then uses these execution profiles (which contain feedback from job execution) to set the tuning parameters for future jobs. Any effect of MapReduce complexity on performance is captured in the execution profiles, which makes feedback-based tuning simpler and more robust than approaches that do not use feedback (such as rule-based tuning).

```
<job_profile>
<input>hdfs://namenode:50001/wiki/txt</input>
<counter key="MAP_INPUT_RECORDS" value="7001"/>
<counter key="MAP_INPUT_BYTES" value="19821797"/>
<counter key="HDFS_BYTES_READ" value="19821933"/>
<counter key="REDUCE_INPUT_BYTES" value="3650063320"/>
<statistic key="MAP_SIZE_SEL" value="11.566"/>
<statistic key="MAP_PAIRS_SEL" value="1995.917"/>
<statistic key="COMBINE_SIZE_SEL" value="0.7479"/>
<statistic key="COMBINE_PAIRS_SEL" value="0.6405"/>
<cost_factor key="READ_HDFS_IO_COST" value="60.45"/>
<cost_factor key="MAP_CPU_COST" value="4686280.79"/>
</job_profile>
```

Figure 1.1: An Example Execution Profile from the Starfish System

Feedback-based tuning using execution profiles is used in the Starfish system [17] and also in [32]. Execution profiles are also used in the PerfXplain system [22] to provide automatic explanations for differences between the observed and expected performance of an MR job. The execution profiles in such systems can consist of general purpose execution log information such as CPU utilization, job durations, or memory used. The profiles can also be based on MapReduce-specific information collected from the execution of an instrumented MR job. As an example, Figure 1.1 shows some of the MapReduce-specific information in a profile from the Starfish system (which we use in this thesis). As the figure shows, an execution profile can contain information collected at runtime from an instrumented MapReduce job about different aspects of the execution of this job such as the cost of different operations and the amount of data read and written by these operations.

Profile-driven tuning is an effective way to deal with the complexity of MapReduce tuning since it reduces the need for a-priori expertise by relying heavily on observed information from job execution. This tuning approach has shown its effectiveness for MapReduce [17, 22] and previously for some tuning tasks in relational database systems [1]. One of the major challenges in profile-driven tuning is providing an execution profile for a submitted MapReduce job. Collecting execution profiles imposes overhead on job execution, especially if it requires running an instrumented MR job. And systems that use profile-driven tuning typically use a given execution profile for tuning only the job from which this profile was collected. The typical workflow in such a system is as follows: When an MR

job is submitted for the first time, it is run without profile-driven tuning and an execution profile is collected from this run. Later, if the same job is submitted again, the profile collected from the first run of the job is used for tuning. The execution profile of one MR job is not used for tuning another job, even if the two jobs are similar.

MR jobs submitted to a cluster can be expected to have some similarity since they process the same data and often reuse the same *map* or *reduce* functions. The similarity between MR jobs is likely to be higher if the jobs are generated from high-level query languages such as Pig Latin [30] or Hive [39]. However, all of this similarity is ignored by current systems that use profile-driven tuning; profiles are collected independently and a profile of one job is not reused for tuning other jobs.

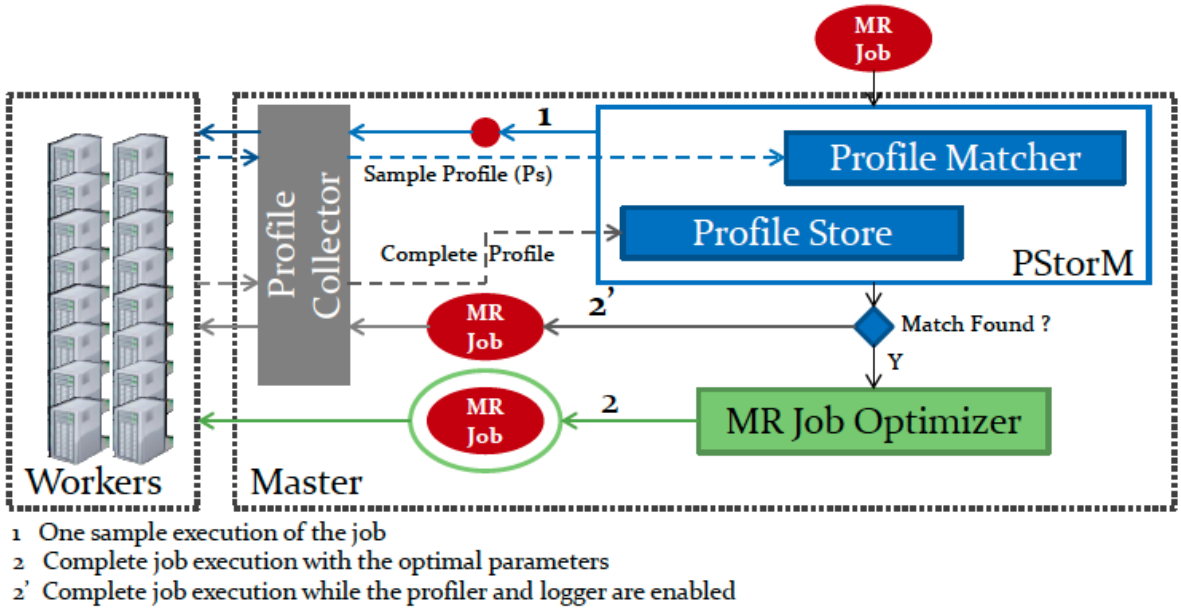


Figure 1.2: PStorM Architecture

This thesis addresses this problem and presents PStorM, a Profile Store and Matcher for feedback-based tuning of MapReduce jobs. Figure 1.2 presents the PStorM architecture. PStorM stores all the execution profiles collected from different runs of MR jobs on the cluster, and uses these stored profiles to provide an accurate profile for a newly submitted MR job. Thus, PStorM consists of two components: a scalable and extensible data store for execution profiles which we call the *profile store*, and a *profile matcher* that can accurately match a submitted job with the stored profiles. The profile matcher can provide accurate profiles *even for previously unseen MapReduce jobs*. The profile matcher can also

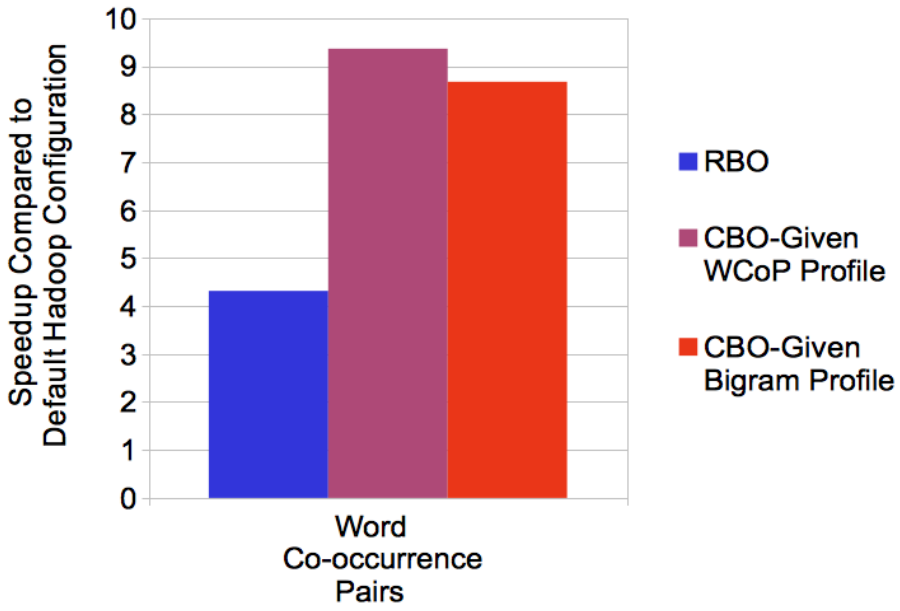


Figure 1.3: Speedups of the Word Co-occurrence Pairs Job Using Different Tuning Approaches

provide a *composite* profile for a MapReduce job using execution feedback from the mapper in one job and the reducer in another job. Thus, PStorM can cheaply and accurately provide execution profiles to a system for feedback-based tuning, such as Starfish. PStorM reuses collected execution profiles for as many jobs as possible, which reduces the need for collecting such profiles, thus reducing the overhead of feedback-based tuning and increasing its applicability. PStorM can be deployed on a Hadoop cluster within an organization in which case it would be used for tuning the jobs of this organization. Alternately, PStorM can be deployed on the cluster of a cloud provider offering Hadoop as a service, in which case it would be used for tuning the jobs of users from potentially different organizations.

As an example to motivate profile reuse, we found that the profile collected during the execution of the bigram relative frequency MR job [26] can be used for tuning the configuration parameters of the word co-occurrence MR job [26] (further details of the MR jobs used in our example and the cluster on which these jobs run are given in Chapter 6). The former job counts the frequency of a pair of subsequent words relative to the frequency of the first word in that pair. The latter job counts the co-occurrences of any pair of words in a sliding window of length n. Therefore, when the length of this sliding window is 2, both jobs exhibit similar behaviour, and consequently similar execution profiles. Figure 1.3

shows the runtime speedups over Hadoop with the default configuration parameter settings achieved for the word co-occurrence pairs job by using different tuning approaches. The first bar in the figure shows the speedup obtained using a Rule-Based Optimizer (RBO) that uses tuning rules based on Hadoop tuning expertise. The second bar shows the speedup obtained using the Cost-Based Optimizer (CBO) of Starfish given the complete profile of the word co-occurrence pairs job itself. The third bar shows the speedup obtained using the Starfish CBO given the profile of the other MR job, the bigram relative frequency job. These different tuning approaches are discussed in more detail later in the thesis. The speedup in the third case is 2x the speedup achieved using the RBO recommendations, and this speedup is only slightly smaller than the speedup achieved in the second case. Therefore, this figure shows that reusing the profiles collected for previously executed MR jobs (the bigram relative frequency job in this example) can result in good runtime speedups for other MR jobs even when they have not been executed before on the cluster (the word co-occurrence pairs job in the example).

The quality of the profiles returned by PStorM is highly dependent on the accuracy of its profile matcher. The accuracy of the matcher is in turn dependent on the features used to distinguish among job profiles and the matching algorithm that uses these features to choose the best profile for a submitted MR job. In this thesis, we use Starfish as the system for generating execution profiles, and the matching algorithm is used to provide Starfish with the profiles it needs for tuning Hadoop configuration parameters (more about Starfish in Section 2.3.1). The PStorM matching algorithm employs a set of *static features* and *dynamic features* to match a submitted MR job to the profiles in the PStorM data store. The static features for a submitted MR job are obtained by analyzing the code of this MR job. The dynamic features of a submitted MR job are obtained by executing *one sample map task* plus the reducers to process the output of this task, and collecting a Starfish profile based on this sampling. The features used by PStorM are selected based on our domain expertise in Hadoop tuning, and we experimentally demonstrate that these features result in higher matching accuracy than features selected using a machine learning feature selection approach. The matching algorithm used by PStorM is also domain specific, and it uses a multi-stage approach to evaluate the distance between different features in the profile. We experimentally demonstrate that this simple matching algorithm performs as well as more complex machine learning algorithms that require expensive training. Our experiments also demonstrate that Starfish tuning of MR jobs based on profiles returned by PStorM results in runtime speedups of up to 9x *even for previously unseen MR jobs*. This speedup is due to PStorM's ability to create profiles for previously unseen MR jobs by leveraging the information collected about other previously executed jobs. The cost paid to obtain this speedup is the consumption of one map slot plus the corresponding reduce

slots for sampling to collect the dynamic features that are used to perform a lookup in the PStorM data store.

Note that when the profile of an MR job in PStorM matches a submitted MR job, this does not mean that the jobs are functionally equivalent. It simply means that the runtime behaviour and resource consumption of the two jobs are similar, and the profile returned by PStorM will result in *correct tuning decisions* for the submitted job. It is reasonable to assume that we will often find jobs that match each other according to this definition, since MapReduce uses a very stylized form of programs and a very stylized form of job execution. For example, every program has a map and a reduce function, and these functions are often reused by different users in an organization; the programs use one of a few input readers to read their data; execution of the map and reduce functions proceeds in *waves* and the output of map tasks is always written to disk; etc. Thus, the jobs on a particular cluster are likely to exhibit similarity, and this similarity is exploited by PStorM.

The contributions of this thesis can be summarized as follows:

- A set of novel features of MR jobs that effectively distinguishes among job profiles.
- A set of similarity measures for use with different types of features.
- A domain specific multi-stage mechanism for matching profiles. The matching algorithm can create an output profile from different stored profiles, which is useful for previously unseen jobs.
- A profile store that organizes the profile information collected from MR jobs.

The rest of the thesis is organized as follows. Chapter 2 provides background and an overview of the related work. Chapter 3 presents an overview of the proposed system. Chapter 4 describes the profile matching technique. Chapter 5 outlines the design of the profile store and discusses other alternative designs. Chapter 6 presents our experimental results. Finally, Chapter 7 provides our conclusions and suggestions for future work.

Chapter 2

Background and Related Work

2.1 MapReduce and Hadoop

MapReduce [6] is a parallel data flow programming model and an execution framework proposed to enable large scale data analysis on massive computing clusters. MapReduce can be viewed as implementing a form of Single Instruction Multiple Data (SIMD) parallelism. The programmer expresses a data analysis task as two special functions: the map and reduce functions. The map function accepts an input key-value pair, and produces a set of intermediate key-value pairs. The reduce function accepts the set of intermediate values associated with the same intermediate key, and produces one or more output key-value pairs. In a MapReduce system, the data is partitioned into multiple *splits*, where each split contains a set of input records (i.e., input key-value pairs). An MR job is composed of independent map tasks such that each task applies the map function to all the input key-value pairs in an input split, and independent reduce tasks where the reduce functions are applied to the assigned partitions of intermediate data. The MapReduce model uses *blocking* execution – the output of a map task is not used until this task finishes its execution. Blocking execution and the fact that the tasks are independent enables the MapReduce execution framework to provide fault tolerance seamlessly to the user. If a map or reduce task fails, this task is simply restarted possibly on a different machine in the cluster.

Hadoop MapReduce [11] is a Java open-source implementation of the MapReduce programming model. The Hadoop framework runs two kinds of daemons: one *JobTracker* runs on one machine in the cluster and one *TaskTracker* runs on each machine of the cluster. Each MR job is composed of map tasks and reduce tasks. An MR job is submitted

to the JobTracker, which is responsible for scheduling tasks on different TaskTrackers (i.e., different machines). Each TaskTracker is responsible for monitoring the execution of the map and reduce tasks assigned to it.

Hadoop has many configuration parameters that affect the performance of a submitted MR job. The Starfish system [17] identified 14 of these parameters that have a major impact on performance (Table 2.1). Some parameters affect the execution of map tasks and other parameters affect the execution of reduce tasks. For example, the size of the map-side buffer, where the output of the map function is serialized, is set to the value of the *io.sort.mb* configuration parameter. Values of *io.sort.mb* that are smaller than necessary lead to frequent IO operations in order to flush the quickly-filled map-side buffer. Also, the parameter *mapred.reduce.tasks* determines the number of reduce tasks to be launched for the current MR job. Sub-optimal setting of *mapred.reduce.tasks* results in overhead on the reduce tasks in terms of the number of intermediate partitions assigned to each reducer. Hence, many stragglers are likely to occur and cause increased job execution time.

Tuning Hadoop configuration parameters is important for any Hadoop cluster, and it will become increasingly important as more users and researchers from diverse domains and backgrounds use MapReduce and Hadoop. These users are experts in their application domains, but they may lack the expertise to tune the Hadoop configuration parameters, which will consequently hurt the performance of their MR jobs.

Good setting of the configuration parameters is dependent on the MR job itself and the input data being processed. Therefore, no single cluster-wide setting can suffice to optimize all Hadoop MR jobs submitted to a cluster. There is a need for tuning tools that take into consideration the characteristics of the MR job and the data. These tools fall into two classes: *rule-based optimizers* and *feedback-based optimizers*. We discuss each of these approaches to tuning in the next two sections.

2.2 Rule-Based Optimizers for Hadoop MR Jobs

Hadoop MapReduce is a new data processing system, so it lacks the sophisticated optimizers that are common in the more mature relational database management systems. Instead, some optimizers exist that make decisions based on predefined rules which are applied when some diagnostic criteria are met. We call such an optimizer a rule-based optimizer (RBO). These rules are heuristics that are developed based on the tuning expertise of Hadoop administrators. Many sources of optimization rules exist [19, 27, 36, 40].

An example rule states that when the intermediate data size is non-negligible or larger

Configuration Parameter	**Description**	**Default Value**
io.sort.mb	Size in MB of the map-side memory buffer	100
io.sort.record.percent	Percentage of the map-side buffer used to store meta-data about the intermediate key-value pairs	0.05
io.sort.spill.percent	Threshold percentage of the map-side buffer that should be reached before a buffer spill to disk is triggered	0.8
io.sort.factor	Number of open streams used during the external merge-sort phase	10
mapreduce.combine.class	Class name of the combiner (Optional)	NULL
min.num.spills.for.combine	Minimum number of disk spills that should exist before the combiner is triggered	3
mapred.compress.map.output	Whether or not to compress intermediate data	false
mapred.reduce.slowstart. completed.maps	Percentage of map tasks that should be completed before the JobTracker can start scheduling the reduce tasks	0.05
mapred.reduce.tasks	Number of reduce tasks spawned during the reduce phase	1
mapred.job.shuffle.input. buffer.percent	Percentage of the reduce-side heap memory used to buffer the shuffled data	0.7
mapred.job.shuffle.merge. percent	Percentage of the reduce-side shuffle-buffer that should be filled before merging is triggered	0.66
mapred.inmem.merge.threshold	Number of map tasks whose intermediate data should be shuffled before the shuffle-buffer is merged	1000
mapred.job.reduce.input. buffer.percent	Percentage of the reduce-side heap memory used to buffer the intermediate data before being fed to the reduce function	0
mapred.output.compress	Whether or not to compress output data	false

Table 2.1: Configuration Parameters for Hadoop MR Jobs

than the input data size, LZO compression should be enabled during the spill phase of the map tasks [27]. Although this tuning action will increase the CPU usage for compression/decompression, the overall job runtime will be decreased because this action will reduce the amount of disk IO at the map side and the amount of data shuffled at the reduce side. Another rule recommends setting the number of reducers to 90% of the available reduce slots on the cluster [19]. The reason supporting this action is that whenever there is one or more failed reduce tasks, there will be other available reduce slots to take over the failed reduce tasks.

The tuning rules used by RBOs typically do not take into consideration cross-parameter interactions [17]. For example, the size of the map-side memory buffer is controlled by the parameter *io.sort.mb*. Another parameter, *io.sort.record.percent*, controls the proportion of this buffer that is used to store the meta-data of the intermediate records serialized in that buffer. For MR jobs with intermediate records of relatively large size, the recommended action is to set *io.sort.mb* to higher values so as to hold more intermediate records. That action should reduce the number of spills from that buffer to disk and the number of merge rounds used to merge those spills into one partition per reducer, and consequently the overall runtime should be reduced. However, this action increases the CPU time used to sort the intermediate records before each spill. Also, setting *io.sort.mb* to higher values consumes more of the heap memory available for the Java virtual machine (JVM) that is running the map task. Therefore, a better action is to set *io.sort.mb* to a slightly higher value and *io.sort.record.percent* to a lower value. Hence, the proportion of the buffer allocated for meta-data is reduced so as to allow more space for the large intermediate records.

There are many sources for tuning rules that can be used in an RBO. Vaidya [40] is a rule-based performance diagnostic tool for Hadoop MR jobs that combines many different rule-action pairs. This tool runs a set of rule-checking tests over the job execution log and the result counters, and it recommends a set of tuning actions for any performance problems that it identifies. The tool comes with default rule-action pairs, and the user is allowed to add new performance diagnostic rules and submit them to the open source rules database.

We have also developed our own RBO as part of this thesis. This RBO combines tuning rules from several sources [19, 27, 36]. Appendix B describes the configuration parameters tuned by these rules, the conditions that cause the rules to be triggered, and the actions taken in each case. We show experimentally that these rules – like any other heuristic approach – are not guaranteed to improve performance, since they are based on certain assumptions that may not hold for a particular MR job. A more robust tuning approach is to use feedback-based tuning, which we describe next.

2.3 Feedback-Based Tuning of Hadoop MR Jobs

Unlike the rule-based optimizers presented in the previous section, feedback-based tuning systems do not rely on global assumptions and do not require a-priori user expertise. These systems simply rely on an execution profile of the submitted job, which captures any effect of the MapReduce complexity on the performance of this job. We will present two examples of such systems: the Starfish and PerXplain systems.

2.3.1 The Starfish System for Hadoop Tuning

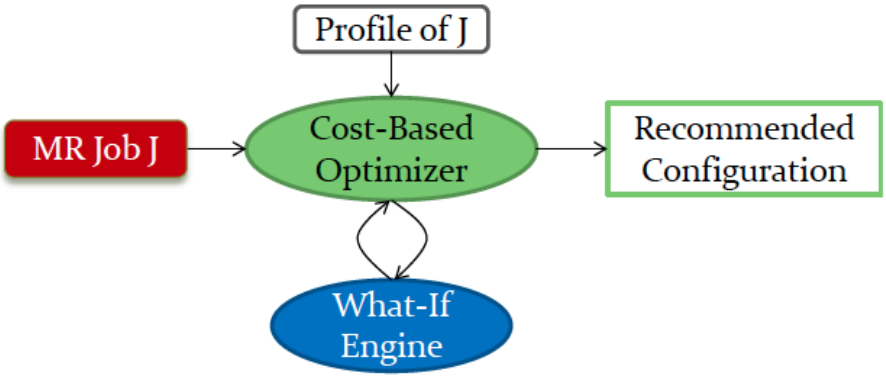
Starfish is a self-tuning system for big data analytics [17]. It adds a new layer to the Hadoop software stack – above the Hadoop MapReduce layer and below the procedural and declarative Hadoop interfaces such as Pig, Hive, and Oozie. Starfish is composed of different components that tackle the tuning problem at three different levels: the job, workflow, and workload levels. The job-level tuning is closely related to the work presented here. This level is composed of four components: MR job profiler, sampler, what-if-engine, and cost-based optimizer.

The profiler collects execution profiles for MR jobs, and these profiles contain fine-grained data flow and cost statistics about the execution of every phase in the map and reduce tasks of the Hadoop MR job. Starfish stores these profiles in a file-based hierarchy with very simple storage and retrieval. The profiler uses dynamic-instrumentation to collect this profiling information at every phase in the map and reduce tasks. Therefore, the profiling code is added to an un-modified MR job, and it can be turned on or off on demand.

The sampler is responsible for selecting random samples of the map tasks and turning on the profiler for only these samples. The collected sample profile represents an estimated profile of the job and is collected with lower overhead than that incurred by complete profiling. The sampler submits for execution only the map tasks selected for profiling in the sample. Since one map task is executed for each input split, the sampler effectively selects a certain number of input splits at random, and eliminates the other input splits. Consequently, the number of executed map tasks will be equal to the number of selected splits.

The What-IF engine (WIF) is responsible for answering what-if questions regarding the expected performance of a given MR job. A job in Starfish is represented as $j = < p, d, r, c >$, where p is the program code of the MR job, d is the input data set, r is the set of cluster characteristics where the job is submitted, and c is the set of configuration

(a) First Job Submission



(b) Subsequent Job Submissions

Figure 2.1: Starfish Optimization Workflow

parameters attached to the job. Given a job profile of the job j, the WIF engine provides predictions regarding the runtime of the job in case any of d, r, or c has been changed. The WIF engine uses analytical cost models for different aspects of MR job execution to predict the runtime of an MR job given its collected profile.

The cost-based optimizer (CBO) is responsible for providing tuning decisions about how to set the configuration parameters. The Starfish tuning workflow for Hadoop MR jobs is shown in Figure 2.1. The first time a job is submitted, it is executed with the default configuration and a profile is collected for that job. In subsequent submissions of the same job, the profile is used by the CBO to find an optimal configuration of the job. To find this configuration, the CBO searches the space of possible values of the configuration parameters, and invokes the WIF engine to predict the runtime of the submitted job at every point selected from that space. The CBO, finally, recommends the configuration settings that resulted in the lowest predicted runtime.

In this thesis, we show that the Starfish CBO using a full profile of an MR job can provide up to 9x runtime speedup for this job as compared to the default Hadoop configuration parameters. The integration between PStorM and Starfish is presented in Chapter 3.

2.3.2 The PerfXplain System for Hadoop Performance Explanation

Another work that uses execution profiles for tuning is PerfXplain [22], which uses profiles for debugging MapReduce job performance and providing appropriate explanations for unexpected job performance. PerfXplain provides a query language to help the user construct a performance question, and generates an explanation answering why the user observed a different value of a certain performance metric than what was expected. The query is composed of an observed performance predicate, an expected performance predicate, and an optional despite-a-fact predicate. The query also contains a pair of jobs, such that one of them shows the expected performance, and the other job shows the observed performance that is different from the expected performance.

First, PerfXplain classifies every job pair in a training set of observed MR jobs based on their execution profiles as either matching the observed relative performance or matching the expected relative performance. PerfXplain then composes an explanation consisting of a set of predicates (performance feature, operator, and value) which have the highest information gain to classify the set of job pairs into the aforementioned two classes.

PerfXplain can make use of the profile store in order to provide more precise explanations. The profile store component of PStorM contains a wealth of information about MR jobs executed previously on the cluster. It can provide not only aggregate CPU and IO statistics about jobs, but also detailed information about every phase in the execution of an MR job, in addition to the code signature of every job represented by the static features collected by PStorM. This wealth of information collected at different levels of abstraction provides a comprehensive history of jobs to be mined by PerfXplain, and thus, PerfXplain can generate more precise and detailed explanations to the user.

2.4 Feedback-Based Tuning in Relational Query Optimizers

The idea of a store for execution feedback information was used in [1] in the context of automatic statistics collection for the query optimizer of the IBM DB2 relational database system. That paper stores execution feedback from query processing in a feedback warehouse and uses this feedback to determine which statistics need to be collected and when to collect them with minimal DBA intervention. Even though that paper uses a feedback

store, the data in that store and the matching algorithm are much simpler than what is required in PStorM.

2.5 MapReduce Tuning using Static Code Analysis

PStorM uses static code analysis to match a submitted MR job against the profile store, relying on the fact that MapReduce programs have a very stylized and restricted form (map and reduce functions, one of a few input readers, etc.). Static code analysis has been previously used for MapReduce tuning in the Manimal system [20]. Manimal is an MR job optimizer that targets the optimization of the MR job at the level of the data flow. Manimal analyzes the MR job looking for relational-style optimization opportunities, i.e., selection and projection operators for which data access can be optimized.

For selection-style optimization, Manimal statically analyzes the compiled code of the submitted job searching for conditional statements applied to the input data such that output data is emitted when a test condition holds true. If such a pattern is found, Manimal suggests building an index over the input data to feed the MR job with only the records that satisfy the test condition. Hence, the execution time used to read the condition-failing records off the underlying distributed file system (DFS) and to check the test condition on those records is avoided.

For projection-style optimization, Manimal modifies the input data stored in the DFS so that this data contains only the fields of the input record that are examined by the MR job. For example, if an MR job analyzes the titles of a given set of web documents and does not examine the content of these documents, the input record that represents one web document can be modified to include only the title of that document. As a consequence, the time spent during the read phase of the map task and the memory allocated for each input record are reduced.

Chapter 3

Overview of the PStorM System

PStorM is composed of two main components: the profile store and the profile matcher. The profile store is an instance of an HBase database [13] that stores execution profiles in an extensible and efficiently accessible schema. The profile matcher is a program that chooses the best execution profile from this HBase database. PStorM runs as a daemon in a normal Hadoop cluster.

In this thesis, PStorM is used to store execution profiles collected by the Starfish system [17], and the profiles returned by PStorM are used for Starfish optimization (Figure 1.2). An overview of Starfish was presented in Section 2.3.1. For each MR job submitted to the cluster, PStorM runs only one map task as a sample of this job using the Starfish sampler, and a sample profile is collected. This sample profile is used by the profile matcher to build a feature vector for the submitted job. This feature vector is used to probe the profile store looking for a matching job profile. If a matching profile is found, it will be provided to the Starfish CBO which in turn will recommend suitable parameter tuning decisions for the submitted job. Then, the job will be submitted for execution on the cluster with the tuned parameters while the profiler is turned off. If no matching profile is found, the job will be submitted for complete execution on the cluster while the profiler is turned on. Then, a complete profile is collected and stored in the profile store. This collected profile will be used for tuning in the next submission of the same job. This profile can also be used for tuning of similar jobs. It can also be combined with other profiles to compose a matching job profile for other submitted MR jobs.

If a complete profile of a job being optimized by Starfish (without using PStorM) is not available, it is possible to collect a profile for use by the CBO by running a sample of the map tasks that make up this job [17]. Sampling exposes a tradeoff between the

cost of profiling and the accuracy of the collected profile. Sampling more map tasks incurs runtime overhead and consumes some of the map slots available on the cluster. These map slots can be used to run the map tasks of other MR jobs, instead of running sample map tasks whose output is discarded. At the same time, sampling more tasks results in a more accurate execution profile. The authors of Starfish propose as a rule of thumb sampling 10% of the map tasks of a job and then running the reduce tasks to process the map output. We will refer to the execution profile collected from such a sample as a *10%-profile*. In our work we have verified that, in most cases, tuning decisions based on a 10%-profile do indeed provide speedups comparable to tuning decisions based on a full profile. However, PStorM has a lower overhead than even the 10%-profile since it requires sampling only one map task for collecting the features used for matching.

The reason that PStorM requires only one sample is that the purpose of sampling in PStorM is fundamentally different from the purpose of sampling in Starfish. Starfish needs to collect enough information from the sample to construct an accurate and representative profile, while PStorM only needs to collect enough information from the sample to probe the profile store and retrieve a matching profile. Thus, PStorM can get away with much lower sampling accuracy (and hence sampling overhead) than Starfish.

Having presented an overview of PStorM, we now turn our attention to its two main components, the profile matcher (Chapter 4) and the profile store (Chapter 5).

Chapter 4

PStorM Profile Matcher

The profile matcher can be considered as a domain-specific pattern recognition problem. Every pattern recognition problem is composed of the following steps: feature selection (Section 4.1), data preprocessing and normalization, finding the appropriate similarity measures (Section 4.2), the pattern matching technique (Section 4.3), and finally adjustment of the matching thresholds.

4.1 Feature Selection

This section answers the following question: What features of an MR job and its profile can distinguish this job from others in the profile store? To answer this question, we first explore the performance models used by the Starfish WIF engine so as to identify the features that play an important role in its runtime predictions. As outlined in [16], these performance models rely on three categories of features:

- Configuration Parameters: The values of the configuration parameters specified by the CBO as it searches the space of possible configurations.
- Data flow Statistics: A set of profile attributes that specify input/output data properties of the map, combine, and reduce tasks (examples in Table 4.1).
- Cost Factors: A set of profile attributes that specify the IO, CPU, and network costs incurred during the course of job execution (examples in Table 4.2).

Feature	Description
MAP_SIZE_SEL	Selectivity of the map function in terms of size
MAP_PAIRS_SEL	Selectivity of the map function in terms of number of records
COMBINE_SIZE_SEL	Selectivity of the combine function in terms of size
COMBINE_PAIRS_SEL	Selectivity of the combine function in terms of number of records
RED_SIZE_SEL	Selectivity of the reduce function in terms of size
RED_PAIRS_SEL	Selectivity of the reduce function in terms of number of records

Table 4.1: Data Flow Statistics

Feature	Description
READ_HDFS_IO_COST	IO cost of reading from HDFS (ns per byte)
WRITE_HDFS_IO_COST	IO cost of writing to HDFS (ns per byte)
READ_LOCAL_IO_COST	IO cost of reading from local disk (ns per byte)
WRITE_LOCAL_IO_COST	IO cost of writing to local disk (ns per byte)
MAP_CPU_COST	CPU cost of executing the mapper (ns per record)
REDUCE_CPU_COST	CPU cost of executing the reducer (ns per record)
COMBINE_CPU_COST	CPU cost of executing the combiner (ns per record)

Table 4.2: Profile Cost Factors

These data flow statistics and cost factors are extracted from the profile that is provided to the CBO (Figure 2.1).

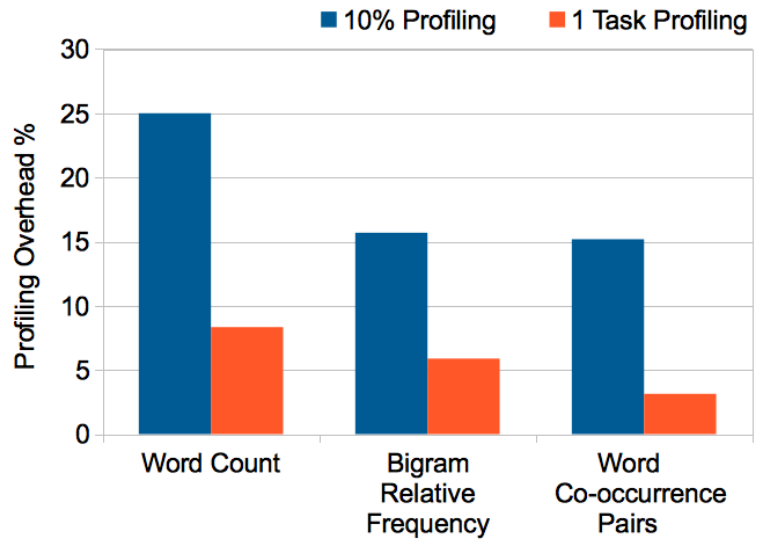
Suppose that a job, J, was executed on the cluster, and its collected profile was P_J. This profile was then provided to the CBO and the recommended configuration parameters were C_J. Now, suppose that the same job, J, is submitted to the cluster and we want to use PStorM to provide the CBO with a profile that will lead to similar recommendations to C_J. The profile matcher should return a profile, P_m, that contain similar data flow statistics and similar cost factors to P_J. The configuration parameters are supplied by the CBO. Hence, the space of features used to represent a profile is narrowed down to the data flow statistics plus the cost factors of the profiled job.

4.1.1 Dynamic Features of MapReduce Jobs

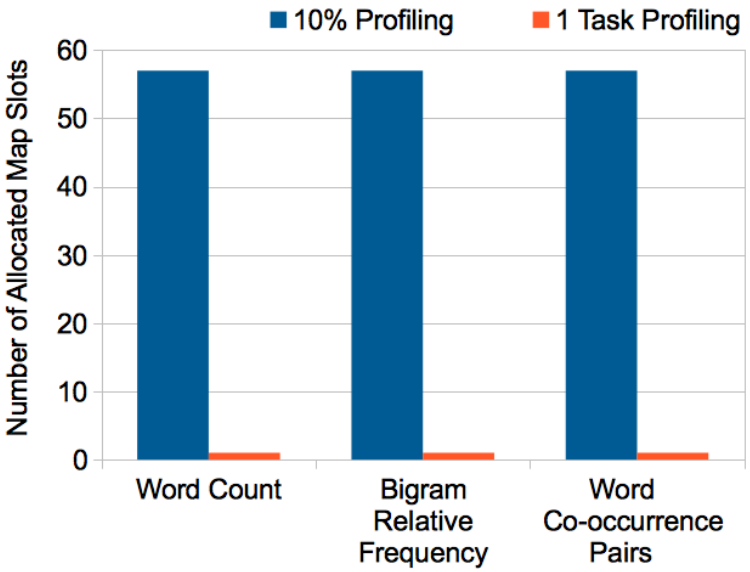
We refer to features extracted from the job profile collected using the Starfish profiler as *dynamic features*, since they are based on the execution of the MR job. A submitted job that is to be matched against the profile store does not initially have an attached profile. In order to create a feature vector for this job to be used by the profile matcher, PStorM executes only one map task of the job. During this execution, the profiler collects the sample profile P_s. The overhead for collecting the profile P_s is low, but the accuracy of this profile is sufficient for use by the profile matcher. To quantify the overhead of 1-task sampling in PStorM, we compare it against the overhead of collecting a 10%-profile in Starfish. Figure 4.1(a) shows the overhead of 10% profiling and 1-task sampling for different MR jobs (details of these jobs are given in Table 6.1, and the jobs are executed on the 35GB Wikipedia data set). The overhead for each job is presented in comparison to the runtime of the job when using the configuration recommended by the RBO while the profiler was turned off. In addition to having lower overhead, the 1-task sampling consumes only one map slot while the 10% profiling consumes 57 map slots, as seen in Figure 4.1(b) (the data set was stored in 571 HDFS splits). Therefore, 1-task sampling does not affect the response time of individual MR jobs and does not reduce the overall cluster throughput.

The features in P_s used for matching against the profile store should have low variance among multiple sample profiles of the same job, and should have high variance among different job profiles stored in PStorM. Drawing on the definition of the data flow statistics, and based on our observations of the values of these features for different job profiles, we conclude that some of the data flow statistics in P_s satisfy these requirements and should be used for matching. The data flow statistics shown in Table 4.1 are the ones that we use for matching. For example, the map size selectivity of an MR job that sorts its input is 1 for all sample profiles collected for that job. On the other hand, the map size selectivity for the word count MR job is larger than 1 for all sample profiles, because the map function emits one intermediate record for every word extracted from an input line. Furthermore, the map size selectivity of the word co-occurrence job is much larger than 1 and also larger than the selectivity of the word count job, because the map function emits one intermediate record for every pair of words extracted from the input line using a sliding window of size n.

On the other hand, the features that make up the profile cost factors do not satisfy the required properties because values of these features suffer from a high variance among sample profiles of the same job. For example, the map CPU cost can differ between two samples of the same job just because the first sample was executed on one input split whose



(a) Profiling Overhead as a Fraction of the Runtime With the RBO Recommendations Without Profiling



(b) Map Slots Consumed

Figure 4.1: Comparison between 10% Profiling and 1 Task Profiling

size is different from the size of the input split of the other sample. As another example, the map CPU cost can differ because one sample was executed on a node that was under-utilized, while the other sample was executed on a node that was over-utilized. The latter example is a normal case in any Hadoop cluster, and is one of the reasons MapReduce has a straggler handling mechanism [6].

Consequently, we need other features that can be extracted from the submitted job, and provide hints about the cost factors. We discuss such features next.

4.1.2 Static Features of MapReduce Jobs

In this section, we explore features that can be extracted statically, i.e., from the byte code of a submitted MR job. We call these *static features*. All MR jobs are developed by implementing certain interfaces, and are executed by a well-defined framework. Every MR job will follow the same course of action: Input data is fed to the mapper in the form of a set of key-value pairs. The map function is invoked to process the designated set of input key-value pairs, and produces a set of intermediate key-value pairs. Intermediate key-value pairs are divided into a number of partitions equals the number of reducers. Each reducer starts by shuffling its designated partition from all mappers to its local machine. Finally, the reduce function is invoked to process the set of values corresponding to the same intermediate key and produces the output key-value pairs.

Therefore, all MR jobs are similar except for certain parts customized by the programmer who wrote the job. These customizable parts are the input formatter, mapper class, intermediate keys partitioner, intermediate keys comparator, reducer class, and output formatter. The class names of most of these customizable parts are among the set of static features that we use for matching (Table 4.3). These customizable parts cause each MR job to have different data flow statistics and different profile cost factors. For example, the input formatter of a MR job that joins two inputs is *CompositeInputFormat* and the input formatter of a word count MR job is *TextInputFormat*. The difference in input formatter leads to different READ_HDFS_IO_COST values in the map-side profiles of these jobs. The same applies to output formatters: different output formatters lead to different WRITE_HDFS_IO_COST values in the reduce-side profiles.

The static features described thus far can be extracted while dealing with the mapper and reducer classes as black boxes. These static features can easily be extracted from the Java byte code without analyzing the logic of this code. However, analyzing the logic of the code can lead to more powerful matching, as we discuss next.

Feature	Description
IN_FORMATTER	Class name of the input formatter
MAPPER	Class name of the mapper
MAP_IN_KEY	Data type of the input key
MAP_IN_VAL	Data type of the input value
MAP_CFG	Control Flow Graph of the map function
MAP_OUT_KEY	Data type of the intermediate key
MAP_OUT_VAL	Data type of the intermediate value
COMBINER	Class name of the combiner
REDUCER	Class name of the reducer
RED_OUT_KEY	Data type of the output key
RED_OUT_VAL	Data type of the output value
RED_CFG	Control Flow Graph of the reduce function
OUT_FORMATTER	Class name of the output formatter

Table 4.3: Static MR Job Features

4.1.3 Static Features Based on the Control Flow Graph

Looking into the execution logic of the map and reduce functions can lead to much better matching, based on deeper analysis and robust to changes in class names. In particular, we have found that analyzing the *control flow graph (CFG)* of the map and reduce functions can significantly contribute to distinguishing MR jobs from each other. The CFG is a graphical representation of all paths and branches that might be traversed by a program during its execution. A vertex in this graph is a branching statement or a block of sequentially executed statements, and an edge represents a goto statement from one branch vertex to another vertex. In PStorM, a CFG is extracted for the map function and another CFG is extracted for the reduce function. We use the Soot tool [41] to extract these CFGs, and we add the CFG of the map and reduce functions to our set of static features.

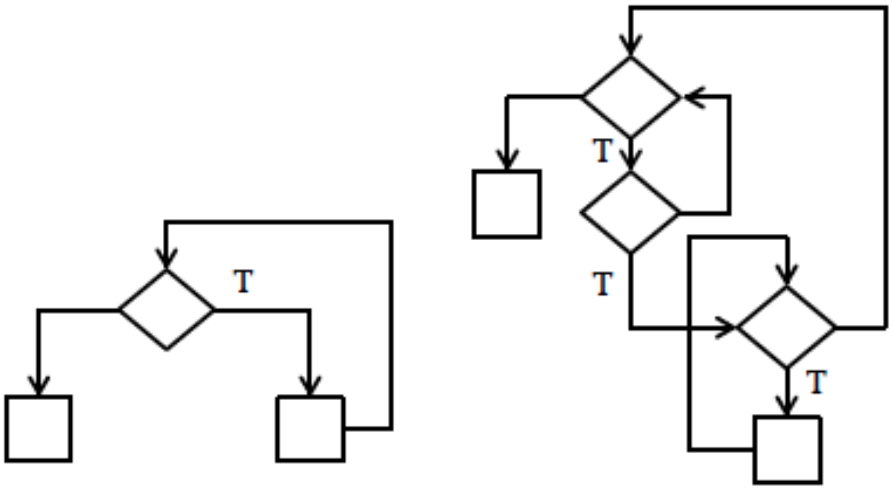
As an example of the use of the CFG in matching, the map functions of the word count and the word co-occurrence MR jobs are shown in Algorithm 1 and Algorithm 2, respectively. These two jobs are part of the workload that we use for evaluating PStorM. More details about these jobs and the settings in which they are executed are provided in Chapter 6. The map function of the word count job contains one loop, which is represented as a cycle in the CFG, shown in Figure 4.2(a). The word co-occurrence map function contains one outer loop, one inner condition, and one inner loop, and has the CFG shown in Figure 4.2(b). The CFGs of the map and reduce functions help to distinguish between

Algorithm 1 Map Function of the Word Count Job

```
function MAP(Object key, Text line, Context context)
    iterator ← line.tokenize()
    while iterator.hasMoreTokens() do
        word ← iterator.currentToken()
        context.write(word, 1)
    end while
end function
```

Algorithm 2 Map Function of the Word Co-occurrence Job

```
function MAP(LongWritable key, Text line, Context context)
    window ← getUserParameter()
    words ← line.extractWords()
    for i = 1 → words.length do
        if isNotEmpty(words[i]) then
            for j = i → i + window do
                pair ← (words[i], words[j])
                context.write(pair, 1)
            end for
        end if
    end for
end function
```



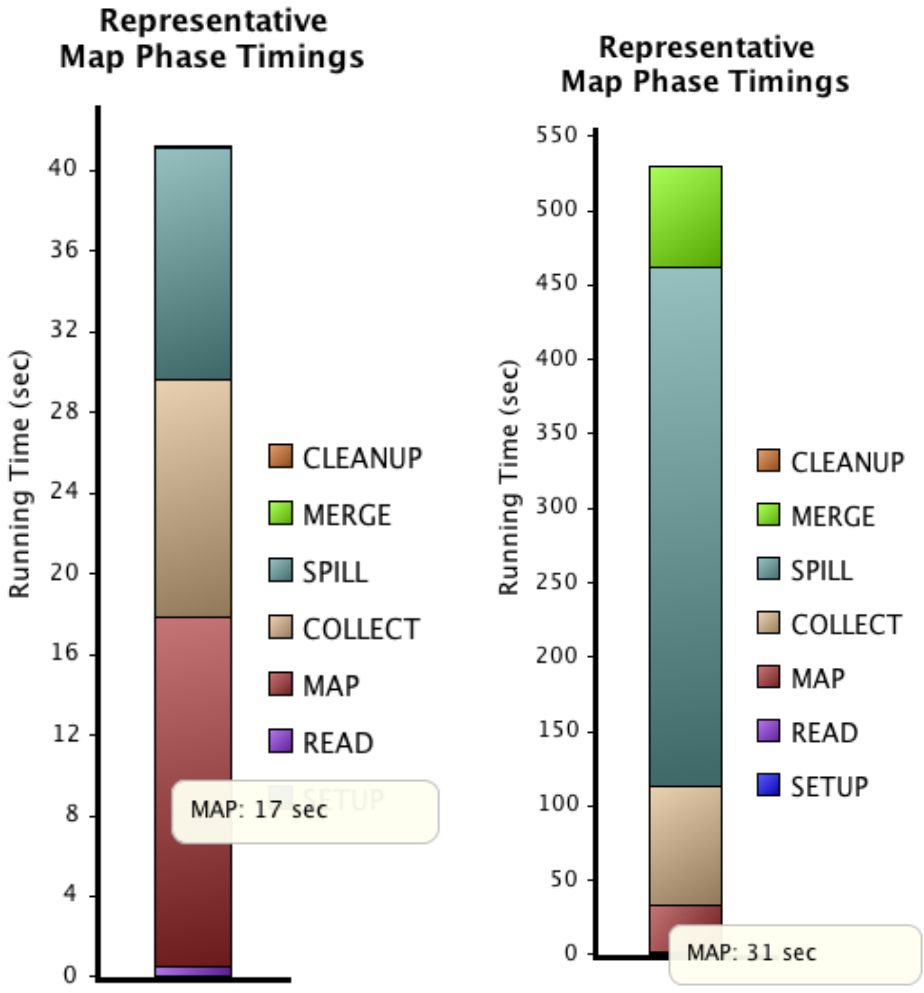
(a) Word Count Job (b) Word Co-occurrence Job

Figure 4.2: CFGs of the Map Functions of the Word Count and Word Co-occurrence MR Jobs

different jobs. Moreover, different CFGs entails different values of the MAP_CPU_COST and REDUCE_CPU_COST features, which are part of the profile cost factors (Table 4.2). To illustrate this point, we show in Figure 4.3 that the map-phase times of these two jobs are different due to the difference between the behaviour of their map functions, which is captured in the CFGs. This behaviour would be captured in MAP_CPU_COST, but we avoid using this feature or any other profile cost factor for matching unless necessary, since profile cost factors have high variance among sample profiles of the same job. The CFG of the map function is a more robust way to distinguish among MR jobs based on their computational requirement.

Comparing the CFG of the map and reduce functions of a job is more robust than comparing hash values of the source or byte codes of these functions. For example, consider two implementations of the word count map function, one that uses a for-loop as in Algorithm 1 and one that uses a while-loop. Both implementations have the same behaviour, and will be matched if comparing the CFGs. However, hashing code will result in a mismatch between these two versions of the word count job.

Matching programs based on their CFGs is, in general, extremely complex or undecidable. However, in our case we use a very conservative similarity metric for matching (discussed in the next section), and if this metric does not result in a match we do not



(a) Word Count Job (b) Word Co-occurrence Job

Figure 4.3: Map-Phase Times of the Word Count and Word Co-occurrence MR Jobs (Captured from the Starfish Visualization System [15])

rely on the CFG but rely on the other features instead. Also, since the jobs being matched are MR jobs which following a restricted programming model, the likelihood of finding a match based on the CFG is increased.

4.2 Similarity Measures for Matching in the PStorM Store

The feature vector constructed for a submitted MR job and the feature vectors stored in PStorM are all composed of two types of features, static and dynamic. The static features are all categorical, and the dynamic features are all numerical. To match jobs based on these features, we need to define similarity measures for both types of features (numerical and categorical).

There are many similarity measures proposed in the literature for matching two pure categorical feature vectors, e.g., Jaccard index, cosine similarity with TF-IDF, and string edit distance. In this thesis we use the Jaccard index to match the static features, since it is a simple similarity measure that incurs low computation cost, and it has been shown to outperform the other similarity measures [12]. The Jaccard index is defined as the fraction of tokens that appear in both of two categorical sets. For our usage, it is defined as follows:

$$Jacc(S_{J_1}, S_{J_2}) = \frac{|S_{J_1} \cap S_{J_2}|}{|S_{J_1} \cup S_{J_2}|}$$

where S_{J_1} and S_{J_2} are the extracted static feature vectors from jobs J_1 and J_2, respectively. The time complexity to calculate the Jaccard index is $O(|S_{J_1}||S_{J_2}|)$. However, in PStorM only corresponding pairs of feature values are tested for equality, which reduces the time complexity to $O(|S_J|)$ (the size of the static feature vector of all jobs is the same).

Jaccard similarity is suitable for all static features except for CFGs. It would be possible to use sophisticated graph matching or graph isomorphism algorithms for matching CFGs, but these algorithms are time consuming. Moreover, we choose to make our CFG matching conservative since a small change in the CFG can lead to a large change in the semantics and resource consumption of the program. Thus, we base our CFG matching on synchronized traversal of the two graphs. We exploit the fact that each CFG has one begin statement, and each statement has either one or two next statements whether it is a normal statement or a branch statement. The following context free grammar describes the structure of the CFGs extracted by the Soot tool that we use in this thesis.

$$
\begin{aligned}
CFG &\models Statement \\
Statement &\models normal\_stmt \mid BranchStatement \\
BranchStatement &\models branch\_cond\ IsLoop\ Successors \\
IsLoop &\models true \mid false \\
Successors &\models Statement\ Statement\ ExpCatchStmt \\
ExpCatchStmt &\models caught\_exp \mid \epsilon
\end{aligned}
$$

Thus, to match two CFGs, we start from the first statement of each CFG, and we move through the CFGs simultaneously using a breadth-first search approach. The range of match score values is not [0, 1] as in the Jaccard index. Instead, it is either 0 or 1, for mismatch or match, respectively.

In addition to matching static features, we also need a similarity measure for dynamic features. The dynamic feature vector is composed of pure numerical features that are defined on different scales. Euclidean distance is a suitable distance measure, but it requires all features to have the same scale. We use Euclidean distance in PStorM but we normalize the features to a common scale. This normalization happens at profile matching time. PStorM stores the minimum and maximum observed values for each feature, and maintains these values when profiles are added to the profile store. At matching time, the minimum and maximum values of each feature are used to normalize the feature value to a number between 0 and 1.

4.3 Multi-stage Profile Matching

The building blocks of the profile matcher have been introduced in the previous sections. In this section, those building blocks are connected together to create a multi-stage profile matching workflow starting from a job that is submitted to the MapReduce cluster, and ending with a matching profile retrieved from the profile store (if a match is found). Such a matching profile can then be used by the CBO.

When a job is submitted to the cluster, one map task is selected randomly to be executed with profiling turned on, along with the reduce tasks to process the output of this map task. This gives us a sample profile P_s. Two feature vectors are constructed, one for map-side matching and the other for reduce-side matching. Each feature vector contains both the dynamic features extracted from P_s and the static features extracted from the byte code of

the submitted job. Hence, each feature vector contains features of mixed data types. The next section describes a generic machine learning approach for computing a distance metric that considers numerical and categorical features in one distance measure. The approach works well, but it incurs a large overhead to build a training data set, learn the model used for matching, and maintain the model as more job profiles are collected. Instead, we propose a multi-stage matching approach based on our domain knowledge, such that the distance between features of different types (numerical or categorical) is calculated in different stages of the matching algorithm.

The profile matching workflow is shown in Figure 4.4, and it is applied twice, one time for map-profile matching and another time for reduce-profile matching. The workflow starts with a set of candidate job profiles, C, consisting of all the profiles stored in the PStorM profile store, and applies three filters to this set until 1 candidate is left. That candidate is the matched profile. First, the Euclidean distance between the dynamic features of each candidate job profile and P_s is calculated, and job profiles with distances larger than a defined threshold θ_{Eucl} are filtered out of C. We refer to this filtered set as C'. The second filter applied is the CFG matcher, and jobs whose CFGs do not match the CFG of the submitted job are filtered out. Third, the Jaccard similarity index between the static features of each job still in the candidate set and the submitted job is calculated, and jobs with similarity index lower than a defined threshold θ_{Jacc} are filtered out. Finally, if more than one job remains in the set, a tie-breaking rule is used that returns the profile of the job whose input data size is closest to the input data size of the submitted job.

The profile matcher declares failure to find a matching profile if the set C becomes empty after the first filter. However, the matcher does not declare failure if the set C becomes empty after the second or the third filters. An empty set after these filters means that the submitted job was never executed before on the cluster. Therefore, an alternative filter is applied. Given the job profiles remaining in the set C', the Euclidean distance between the profile cost factors (shown in Table 4.2) of each job profile and P_s is calculated, and jobs with distances larger than the defined θ_{Eucl} are excluded. Then, the profile of the job whose input data size is closest to the input data size of the submitted job is returned by the matcher. We mentioned in Section 4.1.1 that profile cost factors suffer from high variance among sample profiles of the same job. Therefore, as much as possible we avoid using these features for matching. Instead, we rely on the fact that the static features provide an indirect indication of the cost factors. However, we still use the cost factors for matching if other features fail to narrow the candidate set down to 1 job. This is because the profile cost factors play a crucial role in the runtime prediction of the WIF engine, even though they have a high variance.

The profile matcher returns *No Match Found* when the set of candidate job profiles after

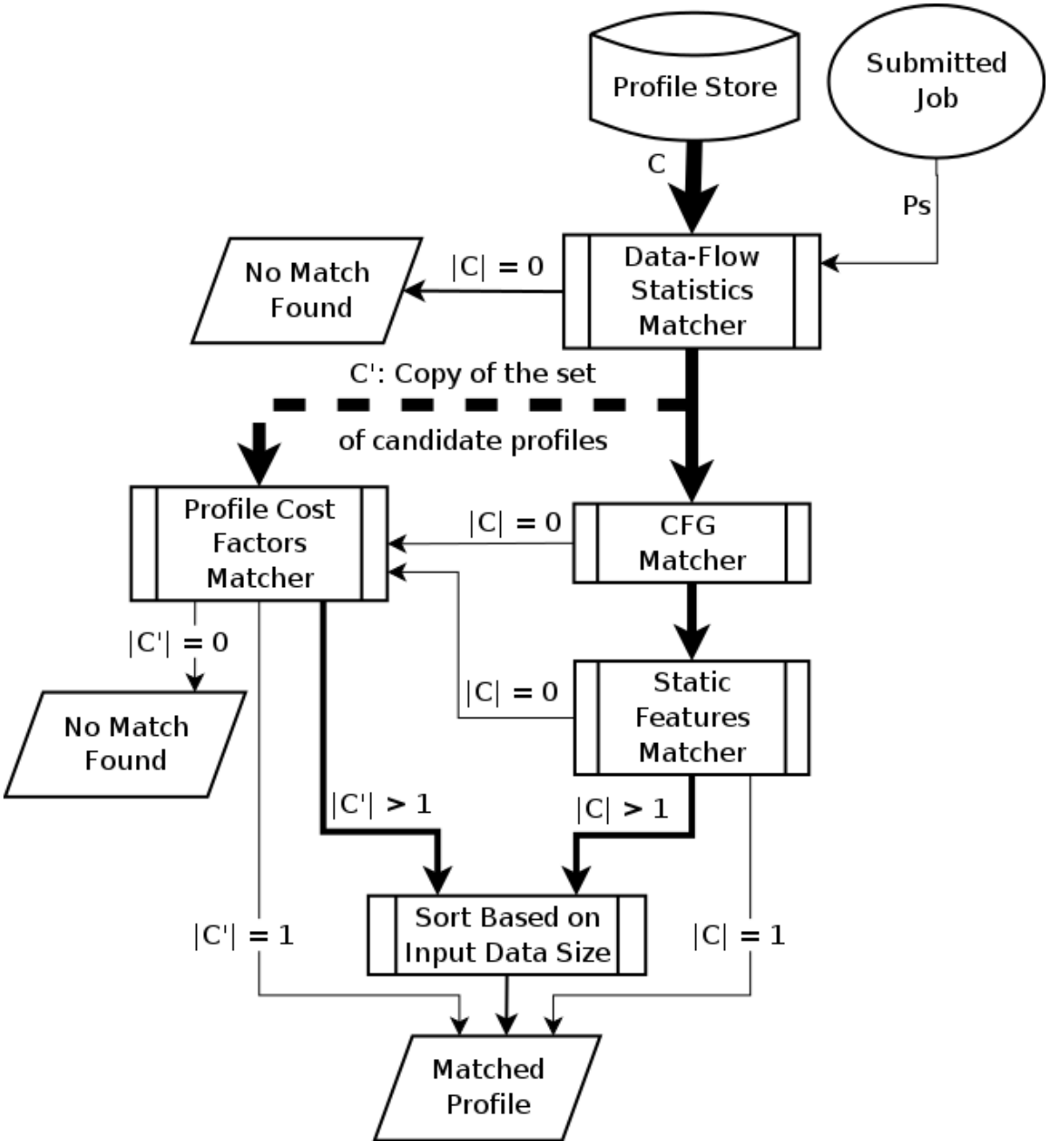


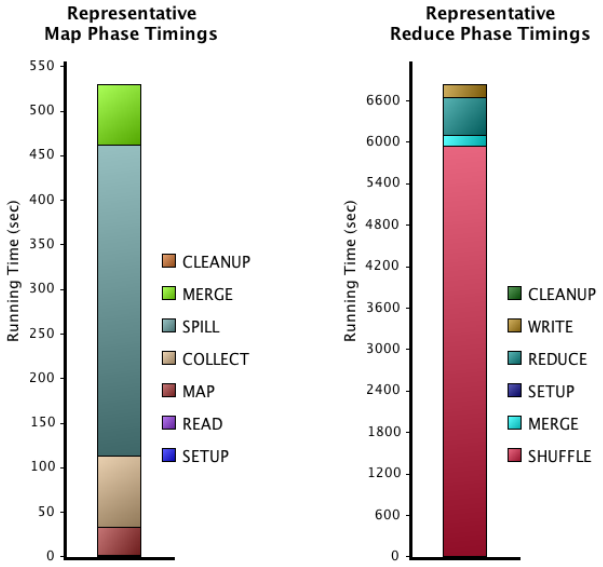
Figure 4.4: The Map/Reduce Profile Matching Workflow

this alternative filter becomes empty. For the case when the matcher returns *No Match Found*, the submitted MR job is executed using its submitted configuration parameters while profiling is turned on. The collected profile is stored in PStorM to be used for future matching.

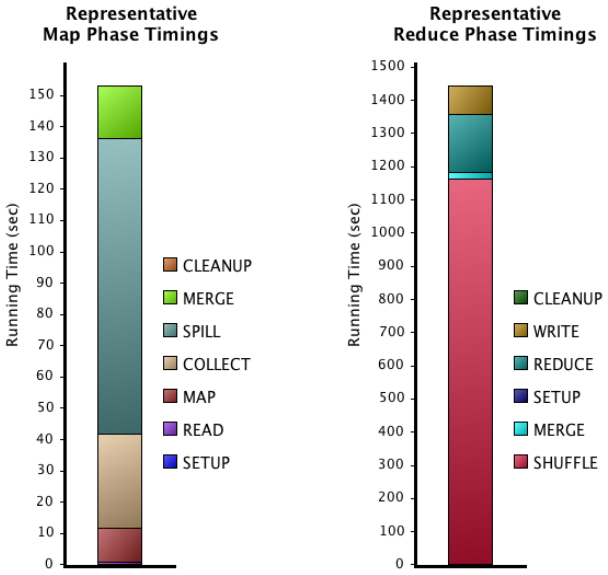
If matching does succeed, the result of map-profile matching is the map profile of job J_1, and the result of reduce-profile matching is the reduce profile of job J_2. The returned job profile is the composition of these two profiles. This profile composition step is useful particularly when the submitted job has never been executed before on the cluster. This step is based on the fact that every MR job is composed of two independent sets of map and reduce tasks. Hence, the collected job profile also contains two independent sub-profiles for the map tasks and the reduce tasks. Therefore, the map profile and the reduce profile of J_1 and J_2 can be composed into a complete profile for the submitted MR job. Our experiments (Section 6.2) support the design decision to return a composite profile for previously unseen jobs. We are able to provide an accurate profile to the CBO even for such jobs.

The following example supports the decision to return a composite job profile as a match for a new job instead of declaring a matching failure. The complete profile of the bigram relative frequency job [26] can be used by the CBO to find the optimal configuration settings for the word co-occurrence job [26]. Recall from Chapter 1 that the bigram relative frequency job counts the frequency of a pair of words in a corpus of documents relative to the frequency of the first word in that pair. The word co-occurrence job counts the frequency of the co-occurrence of words in a window of size n for a corpus of documents. We observe that the complete profile of the bigram relative frequency job has similar data flow statistics and profile cost factors as the complete profile of the word co-occurrence job when the window size equals 2. Figure 4.5 shows a breakdown of the execution times of the map and reduce phases of both jobs, demonstrating how similar the phase times are between the execution of the two jobs on the same data set.

In the matching workflow, filtering based on dynamic features precedes the two filters based on static features. The reason is that job profiles can differ between different executions of the same job with different parameters. For example, the job profiles collected during the execution of the word co-occurrence MR job with different window sizes have different data flow statistics and different profile cost factors. Hence, the job profile collected at the execution with one window size cannot be used by the CBO to recommend configuration parameters for the execution with the other window size. If the static features are used for matching before the dynamic features, profiles of other jobs that have similar data flow statistics would be excluded incorrectly. Hence, we would lose the opportunity to compose a profile using these excluded profiles.



(a) Phase Times of the Word Co-occurrence Job



(b) Phase Times of the Bigram Relative Frequency Job

Figure 4.5: Relatively Similar Phase Times for the Word Co-occurrence and Bigram Relative Frequency Jobs Executed on 35GB of Wikipedia Documents (Captured from the Starfish Visualization System [15])

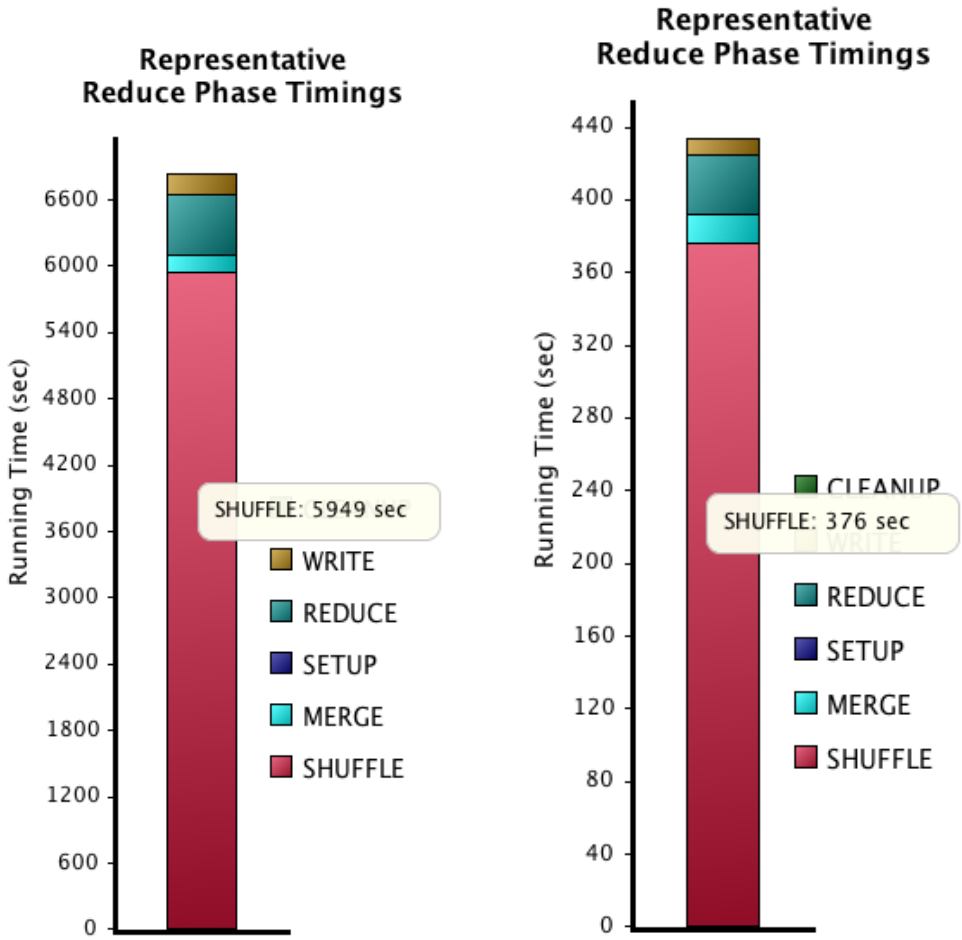
Next, we turn our attention to the rule used to break ties if more than one job profile remains in the candidate set at the end of profile matching. This rule uses the input data size to break ties and select one job profile to return. The reason for this rule is that the execution of the same job on different data sizes results in different intermediate data sizes and hence different shuffle times in the reduce tasks, and consequently different reduce profiles. This is illustrated in Figure 4.6, which shows the shuffle times of the word co-occurrence job on different data set sizes.

4.4 An Alternative Matching Technique Based on Machine Learning

The PStorM matching workflow is based on our domain knowledge of MapReduce. An alternative technique is to use a matching technique based on machine learning. In this section we present such a technique.

After exploration of the pattern recognition literature [3, 10, 18, 24], we found that matching two vectors of numerical and categorical features can be achieved by finding a generalized distance metric that calculates the distance between the two feature vectors directly. Such a metric combines the weighted distances calculated by matching each set of features with the same data type separately. Therefore, a regression analysis technique is required to calculate these weights. Consequently, a training data set is required to learn such a regression model.

In this thesis, we construct the training data set from a set of job profiles as follows. Each sample in the training data set should represent the distances between sets of features of the same data type extracted from a pair of job profiles. To construct our training data set, we use as the first job profile in that pair the complete profile of one job J. In order to enable the learned model to provide matching composite profiles, such profiles should be included in our training data set. Therefore, the second job profile is composed of the map profile of job J_1 and the reduce profile of job J_2, where J_1 and J_2 may or may not correspond to the same job (i.e., complete profile or composite profile). Using this method, the training sample contains nine values of distance/similarity measures: the first four values are the distances between the map profiles of J and J_1, and the second four values are the distances between the reduce profiles of J and J_2. The four distance/similarity measures for the map (or reduce) profiles consist of the Jaccard index score of the static features, the Euclidean distance between the two vectors of data statistics features, the Euclidean distance between the two vectors of the profile cost factors, and the result of



(a) Word Co-occurrence Job Executed on 35GB of Wikipedia Documents

(b) Word Co-occurrence Job Executed on 1GB of Randomly Generated Text

Figure 4.6: Different Shuffle Times During the Execution of the Word Co-occurrence Job on Different Data Sets

matching the CFGs of the two jobs. The ninth value in the training sample is the difference between the runtime predicted by the WIF engine for the job J given the first profile in the pair, and the runtime predicted by the WIF engine for the same job but given the second (composite) profile. This value represents how well the two profiles in the training sample match each other. That is, how close they are to each other. The distance metric used for matching is the weighted sum of the individual distances/similarities between different feature types, as shown in the following equation:

$$\begin{aligned} D \;=\; & w_1\, Jacc_{map} \;+\; w_2\, Eucl\_DS_{map} + \; w_3\, Eucl\_CS_{map} \;+\; w_4\, CFG\_Match_{map} \\ & +\; w_5\, Jacc_{red} \;+\; w_6\, Eucl\_DS_{red} + \; w_7\, Eucl\_CS_{red} \;+\; w_8\, CFG\_Match_{red} \end{aligned} \quad (1)$$

The goal of the learning algorithm is to learn the weights to use based on the training samples. The distance D used during training is the difference between the runtimes predicted by the WIF engine, described above.

In order to improve the quality of the machine learning model, we ensure that the training data set contains a sample that represents the distance between the profile of each job J and itself. The distance D for such a sample would be zero. Thus, such a sample provides the machine learning algorithm with an example of a perfect match.

A state-of-the-art technique that is used in this kind of learning problem and provides good results [7, 42] is *Gradient Boosted Regression Trees (GBRT)* [34]. GBRT produces the learned model in the form of an ensemble of decision trees. We used an implementation of this technique in the R [38] statistical software package to calculate the weights that combine the partial distances into a generalized distance metric (Equation 1). This learned distance metric is used to provide the matching map and reduce profiles for a submitted MR job. In particular, the matching profile returned for the submitted job is the profile that has the smallest distance to this job according to the learned distance metric (i.e., the nearest neighbour according to this metric). The R source code we implemented to learn this distance metric is provided in Appendix A.

Section 6.1 presents a comparison between our proposed domain-specific multi-stage matching technique and GBRT in terms of profile matching accuracy. We will see that our simple domain-specific matcher works as well as the more complex GBRT matcher, which requires an expensive training step.

Chapter 5

PStorM Profile Store

The other component of PStorM is its profile store, which provides a repository of the profiles collected on the cluster. The profile store should meet certain system requirements. First, it needs to provide *scalable* storage for the large number of profiles collected. This is of particular importance if Hadoop is running on a public cluster (i.e., Hadoop is provided as a service in the cloud). Each job profile is on the order of a few hundred bytes in size, but scalability is still required to support a large number of profiles and to ensure that access to the profile store is fast. Second, the kind of workloads that have to be supported by the profile store are expected to be *analytical*. The profile matcher reads job profiles frequently for the purpose of choosing the matching profile for a submitted MR job. Updates to the profile also follow an analytics-style pattern. Updates consist of adding new profiles as jobs get executed, and possibly deleting old profiles to free up space. There are no in-place modifications. Furthermore, we envision that PStorM can be used by other Hadoop job analysis and optimization systems, e.g., PerfXplain [22] and Manimal [20], which also require analytics-style access. Third, we require the data collected for each MR job to be organized in an *extensible* data model, so as to incorporate new types of data that are necessary for other job optimizers and analyzers that will make use of the massive information stored in PStorM.

We have found that HBase [13] meets the above requirements, and we adopt it as the storage system for the PStorM data store. HBase is a distributed column-family oriented data store. It is designed to scale efficiently in both dimensions: in rows by horizontal partitioning and replication mechanisms, and in columns by physically storing columns of each column family in different files. HBase is also considered a good candidate for the PStorM data store because HBase stores data in the Hadoop Distributed File System

(HDFS), which is part of any cluster on which PStorM runs. Hence, no new infrastructure components and few new daemons need to be added to the cluster.

5.1 PStorM Data Model in HBase

Data is stored in HBase in the form of key-value pairs. More specifically, the data consists of a set of *rows*, where each row is identified by a *row-key* and has one or more *column families*. A column family has one or more *columns* identified by a *column name*. The set of columns under the same column family can be different between rows. The data items in HBase are key-value pairs where the key is composed of the row-key, column-family identifier, column name, and a timestamp. Thus, to use HBase for the PStorM data store, we need to design an HBase data model for profiles, which requires us to specify the row-key, the column families, and the columns within these families.

A simple data model for the profile information collected about an MR job is to make the row-key be the job ID, the column family be the feature type (e.g., static or dynamic), and the column names be the feature names. Thus, the key used by HBase would be (job ID, feature type, feature name). The value indexed by this composite key is the feature value. This data model does not meet the extensibility requirement, because HBase does not allow adding more column families once a table is created, and a new column family is required for a new feature type.

Therefore, we organized the job information into another data model, illustrated in Table 5.1. The row-key is composed of the feature type as a prefix and the job ID. Only one column-family is used. Each column name is a feature name whose type is indicated by the prefix of the current row. For example, Table 5.1 shows two static features and two dynamic features for two MR jobs. This model supports extensibility in the two dimensions presented earlier. Adding a new feature type requires adding a new prefix to the row key. Adding a new feature to an existing feature type requires adding a new column to the column-family in the rows whose prefix represents that feature type. In addition, this model boosts the performance of the profile matcher. As explained in Section 4.3, the profile matcher calculates the similarity/distance scores between feature vectors of the same type at each stage of the matching algorithm. Therefore, storing the dynamic features and the static features in separate partitions enhances data locality from the viewpoint of the matcher. This is achieved automatically by HBase, because rows are partitioned horizontally into regions and the feature type is stored as the prefix of the row key.

Row-Key	Column Family			
	Column Name	**Column Name**	**Column Name**	**Column Name**
	CF			
	IN_ FORMATTER	OUT_ FORMATTER	MAP_SIZE_ SEL	RED_SIZE_ SEL
Static/Job1	Text	Text	-	-
Static/Job2	Sequence	Sequence	-	-
Dynamic/Job1	-	-	1.0	1.0
Dynamic/Job2	-	-	11.5	0.26

Table 5.1: PStorM Data Model in HBase

5.2 Alternative Data Models

We explored other data models before settling on the data model presented in the previous section. In this section, we present these models and the reasons they were excluded.

5.2.1 OpenTSDB Data Model

OpenTSDB is a distributed and scalable monitoring system [31]. It runs multiple and distributed Time Series Daemons (TSD). These TSDs collect logs and different performance metrics from nodes in a cluster and store this collected data as time series data points in HBase. In the data model used by OpenTSDB, the row-key has the following format:

$$< metric\_ID >,\ < base\_timestamp >,\ [< tag\_value\_pairs >\ ...]$$

where base_timestamp is the original timestamp of the data point rounded to m minutes. Only one column family is used, and the column-name is prefixed with the delta in seconds from the base_timestamp.

In order to store the PStorM profile collected for an MR job in OpenTSDB, the row-key should be in the following format:

$$< feature\_name >,\ < timestamp >,\ JobID =< job\_id >$$

Since rows in HBase are partitioned horizontally according to the row-key, data points of the same feature will be collocated in the same HBase region (i.e., partition). However,

data points of the same feature type (dynamic or static) will be spread among different HBase regions. This causes poor data locality from the viewpoint of the profile matcher. Profile matching time will be increased because data points would need to be collected from different regions in order to build the feature vector of each MR job.

5.2.2 One Table Per Feature Type

Another alternative data model is to store values for each feature type in a separate HBase table. Thus, there would be two tables, one for the static and one for the dynamic features. The physical design of HBase causes this data model to suffer from poor performance.

Rows for each table in HBase are horizontally partitioned into regions. Each region is assigned to one region server that manages its read/write requests. HBase maintains two system tables called *-ROOT-* and *.META.* tables. Entries in the -ROOT- table contain the IDs of the region servers that manage the regions of the .META. table. Entries in the .META. table are ordinary key-value pairs, such that the key is in the following format [9]:

$$< table\_name >,\ < starting\_key >,\ < region\_ID >$$

The value in this key-value pair is the ID of the region server to which the region represented by the key is assigned.

In this data model, entries in the .META. table will be like the following:

$$Jobs\_Static,\ < starting\_job\_id >,\ region\_1$$

$$Jobs\_Static,\ < starting\_job\_id >,\ region\_2$$

$$Jobs\_Dynamic,\ < starting\_job\_id >,\ region\_1$$

$$Jobs\_Dynamic,\ < starting\_job\_id >,\ region\_2$$

In contrast, entries in the .META. table following our data model presented in Table 5.1 will be like the following:

$$Jobs,\ Static\_ < starting\_job\_id >,\ region\_1$$

$$Jobs,\ Static\_ < starting\_job\_id >,\ region\_2$$

$$Jobs,\ Dynamic\_ < starting\_job\_id >,\ region\_1$$

$$Jobs,\ Dynamic\_ < starting\_job\_id >,\ region\_2$$

In principle, these two data models should result in the same performance and achieve the same design goals. However, practically, this alternative data model will result in higher load on the region servers because the region server will maintain an in-memory *Store* object for each column-family of each table in the served region [9]. Therefore, this data model was deemed unsuitable.

5.3 Profile Store Performance Optimization

The profile matcher workflow presented in Section 4.3 is composed of three consecutive filters. This workflow can be executed on the PStorM client side. That would require large data transfers from the region servers to the client in order to perform profile matching at the client, which would represent a bottleneck to system scalability as the number of profiles stored in PStorM grows.

HBase supports a filter-pushing mechanism in which filters (i.e., predicates) are pushed down from the client to the region servers. These filters are then applied at the region servers and only the data that passes the filter is sent to the client. PStorM uses this filter-pushing mechanism. Each filter in the workflow is serialized and sent to the region server. The region server performs a scan over regions assigned to it, and applies the filter directly. Then, the region server returns the job-IDs of the job profiles that successfully passed that filter. Hence, the filters are applied in a distributed fashion, and the amount of data transferred between the region servers and the client is reduced.

Chapter 6

Experimental Evaluation

All our evaluation experiments were conducted on Amazon EC2. We conducted the experiments on a Hadoop cluster composed of 16 Amazon EC2 nodes of the c1.medium type. The cluster is configured as one master node running the JobTracker and the NameNode daemons, and 15 workers running the TaskTracker and the DataNode daemons. Each worker node has 2 virtual cores (5 EC2 compute units), 1.7 GB of memory, 350 GB of instance storage, and is configured to have at most 2 available map slots and 2 reduce slots. Child processes of the TaskTracker are configured to have a maximum heap size of 300 MB. The code signature of the submitted jobs and their collected Starfish profiles are stored in HBase. HBase daemons, one HMaster and one HRegionServer, are run on the master node.

We developed a custom benchmark to evaluate PStorM. The benchmark consists of different Hadoop MapReduce jobs that have practical usage in various research and industrial domains. Most of the jobs were executed on two different data sets while profiling was enabled. The collected profiles were stored in PStroM. The MR jobs and the data these job were run on are given in Table 6.1.

As explained in Section 4.3, PStorM uses the multi-stage profile matching approach, which consists of three filters with two thresholds. For these experiments, the Jaccard threshold, θ_{Jacc}, is set to *0.5* and the Euclidean distance threshold, θ_{Eucl}, is set to $\frac{1}{2}\sqrt{number\_of\_dynamic\_features}$. Since numerical data in the feature vectors is normalized to the range [0,1], the maximum Euclidean distance between any two feature vectors is $\sqrt{number\_of\_features}$. The threshold θ_{Eucl} is adjusted to the half of this maximum possible Euclidean distance.

MapReduce Job	Application Domain	Data set
CloudBurst [35]	Bioinformatics	Sample genome and Lake Washington Genome [21]
Frequent Itemset Mining [29]	Data Mining	Webdocs data set of size 1.5GB [28]
Itembased Collaborative Filtering [29]	Recommendation Systems	Movies rating data sets with 1M and 10M ratings [33]
Join	Business Intelligence	1GB and 35GB of data generated by TPC-H benchmark
Word Count	Text Mining	1GB of random text and 35GB of Wikipedia docs
Inverted Index [26]	Text Mining	1GB of random text and 35GB of Wikipedia docs
Sort	Many Domains	1GB and 35GB of data generated by Hadoop's TeraGen
PigMix-17 Queries	Pig [30] Benchmark	1GB and 35GB of data generated by PigMix
Bigram Relative Frequency [26]	Natural Language Processing	1GB of random text and 35GB of Wikipedia docs
Word Co-occurrence Pairs [26]	Natural Language Processing	1GB of random text and 35GB of Wikipedia docs
Word Co-occurrence Stripes [26]	Natural Language Processing	1GB of random text

Table 6.1: Benchmark of Hadoop MapReduce Jobs

6.1 Profile Matching Accuracy in PStorM

In this section, the accuracy of the profile matcher of PStorM is evaluated. We conducted these experiments with the contents of the profile store in one of two states. In the first content state, when a submitted MR job is executed on a specific data set, PStorM has the complete profile collected during execution of the same job on the same data set. This state will be referred to as *SD* (for *Same Data*). This state acts as a sanity check for the profile matcher in PStorM, since any reasonable matching algorithm should retrieve the job profile that was collected during a previous execution of the same submitted job. In the second content state, when a submitted MR job is executed on a specific data set,

PStorM has the complete profile collected during the execution of the same job, but on a different data set. For example, when submitting the word co-occurrence job on 35GB of Wikipedia documents, the profile store has the profile for this job but on a 1GB data set. This content state will be referred to as DD (for *Different Data*). We will refer to these two complete profiles of the same job but collected during execution on different data sets as *job-profile twins*.

We used the number of correct matches as a fraction of the total number of job submissions as the accuracy metric for evaluating the profile matching algorithms. When the profile store is in the first content state (SD), a correct match is the complete profile of the same job executed on the same data set, while in the second content state (DD), a correct match is the twin of that complete profile.

Profile matching can be viewed as a domain-specific pattern recognition problem. Given a 1-task sample profile of the submitted job, plus the static features of this job, find the most similar profile stored in the profile store. PStorM uses a domain-specific matching algorithm. In particular, the features used by PStorM and the distance metrics are both domain specific. In the next two sections, we compare the features and distance metrics used by PStorM to more generic alternatives.

6.1.1 Feature Selection

One of PStorM's contributions is the proposed set of static features, which provide good indications of the profile cost factors and are more suitable than the cost factors since the cost factors exhibit high variance among sample task profiles of the same MR job. Another contribution is the domain-specific feature selection approach based on our Hadoop expertise, which leads to feature vectors with a mix of numerical and categorical features.

An alternative to using the PStorM features is to select a set of candidate features from the dynamic features that exist in the collected Starfish profile. These features are ranked according to their *information gain* scores, a commonly used approach in applied machine learning techniques [22]. The highest ranked F features are selected to build the feature vector, such that F equals the total number of static and dynamic features used by PStorM. Since all features in a Starfish profile are numerical, the highest ranked features will be numerical. Therefore, we can simply use the Euclidean distance with this feature selection method to evaluate the distance between job profiles. When matching a submitted MR job to the profiles stored in PStorM, the profile in the profile store whose distance from the 1-task sample profile of the submitted job is the lowest (i.e., the nearest neighbour), is selected as the matching candidate for the submitted job.

A second alternative to PStorM's feature selection is to use the static features proposed by PStorM, in addition to the dynamic features in the Starfish profile, but select from this augmented set of features using a generic machine learning feature selection approach. That is, take the idea of static features from PStorM, but not the specific set of features that PStorM chooses in a domain-specific way. As in the first alternative, feature selection in this case is also based on ranking the features according to their information gain scores. Since this augmented set of features includes static and dynamic features, the highest ranked F features might contain a mix of static and dynamic features. However, when we applied this approach to the profiles stored in the profile store, we found that the highest ranked F features are all numerical. Hence, the same matching algorithm is used as in the first alternative feature selection approach. The first alternative feature selection approach will be referred to as *P-features* (for *profile features*), and the second approach as *SP-features* (for *static and profile features*).

Figure 6.1 shows the matching accuracy scores achieved by P-features and SP-features as compared to PStorM. Since PStorM executes the matching algorithm on the map profiles separate from the reduce profiles, the matching scores are presented separately for the map and reduce sides. It can be seen from the figure that PStorM outperforms the two alternative feature selection approaches in both content states of the profile store. In the SD state, despite the fact that the complete profile of the submitted job on the same data set exists in the profile store, both P-features and SP-features failed to return the correct profile for more than 35% of the submitted jobs.

In the second content state, DD, PStorM did not achieve a 100% accuracy score. PStorM resulted in five and seven false-positive results at the map and reduce sides, respectively. Some of these mismatches are because there are four profiles whose twins are not stored in PStorM (i.e., the MR job is run on only one data set). The word co-occurrence stripes job did not complete its execution on the Wikipedia data set because it resulted in exceptions related to limited memory space. Also, the frequent itemset mining job is executed on only one data set as presented in Table 6.1. This job is composed of a chain of three MR jobs, and hence the profiles of these three jobs do not have profile twins in the profile store.

6.1.2 Multi-Stage Profile Matching

As shown in the previous section, the set of features used by PStorM results in the best matching accuracy at both of the content states of the profile store. This set of features contains numerical and categorical values. PStorM does not match features of both data types at once. Instead, it uses the multi-stage matching approach presented in Section 4.3.

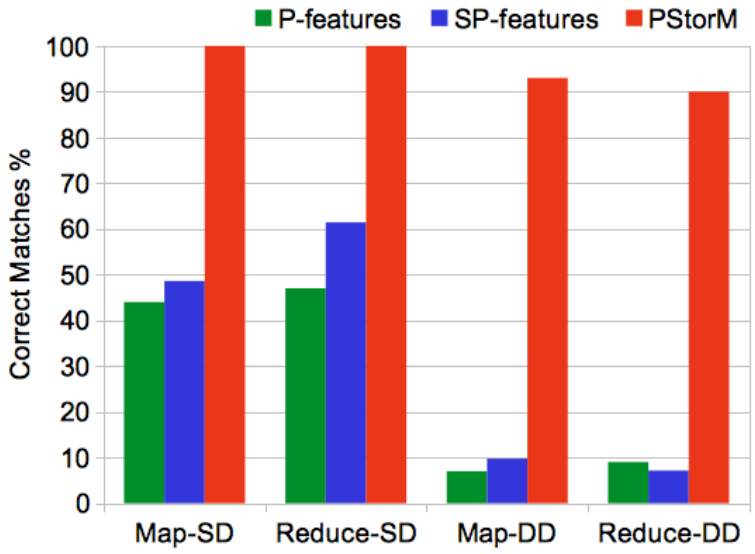


Figure 6.1: Matching Accuracy of PStorM Compared to Different Alternatives for Feature Selection

An alternative to the PStorM matcher is the GBRT matcher presented in Section 4.4. In this section, we compare the PStorM profile matcher with GBRT. Figure 6.2 shows the matching accuracy of PStorM and four different parameter settings for GBRT. We tried different parameter settings for GBRT to find the setting which resulted in the highest matching accuracy. We describe these parameter settings next.

The first GBRT parameter setting (GBRT 1 in Figure 6.2) corresponds to the default parameter settings of GBRT in the R statistical package, which are as follows:

- Fraction of training data used for learning = 50%
- Number of cross validation folds = 10
- Distribution = Gaussian
- Number of iterations = 2000
- Learning rate or shrinkage = 0.005

In the second parameter setting (GBRT 2), the Laplace distribution was used instead of Gaussian. In the third parameter setting (GBRT 3), the number of iterations was increased

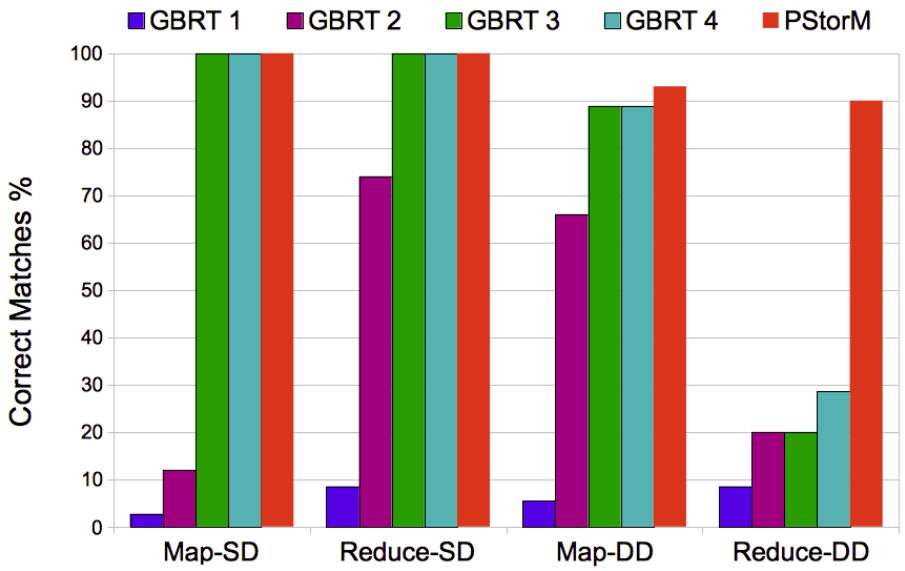


Figure 6.2: Matching Accuracy of PStorM Compared to GBRT

to 10,000, the learning rate was set to 0.001 [34], and the fraction of training data was increased to 80%. In the fourth parameter setting (GBRT 4), the fraction of training data was increased to 100%. This makes GBRT overfit the data, but it results in the highest matching accuracy, as seen in Figure 6.2.

Comparing PStorM and GBRT, we can see that PStorM is as accurate as GBRT or better in all cases, even when GBRT overfits the training data. GBRT is a powerful and mature machine learning algorithm, so we expect it to perform well in terms of the matching accuracy in most cases. However, the accuracy of GBRT comes at a cost since it is a complex algorithm that requires collecting training data and training a model for every new cluster. On the other hand, PStorM results in high matching accuracy using a simple algorithm that does not need training.

6.2 Tuning Effectiveness with PStorM

From the user's perspective, runtime speedup is the main goal of the entire parameter tuning exercise. When using PStorM, a user should see an improvement in runtime. That is, the total runtime of a submitted MR job with PStorM should be lower than the runtime using the default Hadoop configuration or the RBO recommendations.

Job Name	Runtime (min)
Word Count	12
Word Co-occurrence Pairs	824
Inverted Index	100
Bigram Relative Frequency	302

Table 6.2: Runtimes with the Default Hadoop Configuration

We would like to see such runtime improvement even for previously unseen MR jobs. Therefore, we introduce third content state of the profile store for this experiment, which we refer to as NJ (for *New Job*). In this content state, the submitted MR job is a new job that has never been executed before on any data set on the cluster, and hence it has no job profile stored in PStorM. In this state, the profile matcher in PStorM can either build a composite job profile or declare that no matching profile is found.

To evaluate whether PStorM leads to better tuning, we conducted an experiment with four different MR jobs, all of which are executed on the 35GB Wikipedia data set. The runtimes of these jobs with the default Hadoop configuration is shown in Table 6.2, and the speedup of different tuning options compared to this default is shown in Figure 6.3. The figure shows the speedups achieved by the RBO, and by the Starfish CBO using profiles returned by PStorM in the three content states of the profile store: SD, DD, and NJ.

The first observation we make about Figure 6.3 is that the RBO does not always improve performance over the default Hadoop configuration. In one case, the RBO actually results in a performance degradation (the inverted index job). The rules in the RBO make certain assumptions and only cover certain cases, so it is quite possible for the RBO to miss optimization opportunities. A user can never be assured that the RBO recommendations are better than the default Hadoop parameter settings. A better tuning alternative is a cost-based optimizer such as the one provided by Starfish.

Figure 6.3 shows that PStorM achieves speedups over the default configuration for all content states, even NJ, in which the submitted job has never been seen before. In the NJ content state, PStorM builds a composite job profile that guides the CBO to provide configuration parameters that are as good as (or close to) the SD state. That is, the profile provided by PStorM in the NJ content state is identical to the one that Starfish would collect in a complete profiling run of the submitted MR job. The speedups of PStorM are always higher than the RBO. The magnitude of the speedup varies from job to job, depending on how good the default Hadoop configuration parameters are for the job. For example, the speedup is only slightly higher than 1 for the inverted index job, which

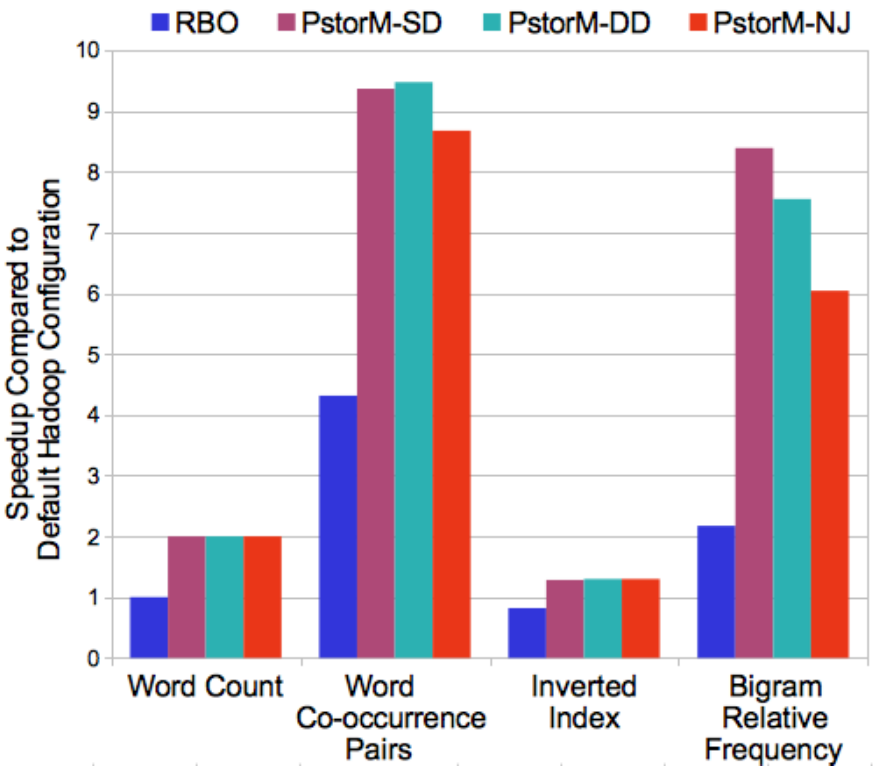


Figure 6.3: Speedups of Different MR Jobs With Different Configuration Parameter Settings

indicates that the default parameters are quite suitable for this job. On the other hand, the speedup for the word co-occurrence pairs job is around 9, and is double the speedup achieved by the RBO.

To summarize our experiments, we have shown that the PStorM profile matcher is highly accurate. Alternative feature selection algorithms cannot achieve the same level of accuracy as PStorM, and the similarity measure used by PStorM is as good as (or better) than a measure based on the GBRT machine learning approach. GBRT is a complex, powerful, and expensive approach, and PStorM achieves the same or better accuracy using a simpler and cheaper approach based on domain knowledge. We have also shown that the RBO is not a reliable tuning approach, and that PStorM with the Starfish CBO results in significant speedups even for previously unseen MR jobs.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Due to the wide adoption of the Hadoop MapReduce framework, tuning Hadoop configuration parameters has become increasingly important, especially since job performance is significantly affected by the configuration parameter settings. Feedback-based tuning approaches are effective in tuning the configuration parameters because they rely on execution profiles that capture the complexities of execution of MR jobs. Execution of MR jobs is complex and difficult to model because of the black-box nature of the map and reduce functions, and because of the heterogeneity of the clusters on which the jobs execute. A significant problem with feedback-based tuning approaches is that they require an initial complete execution of the job with the default configuration to collect an execution profile, and the collected execution profiles are not used for other jobs, even ones that are similar.

PStorM addresses this problem and provides a system for reusing profiles for multiple jobs. PStorM is composed of a scalable and extensible profile store for all the execution profiles captured on a cluster, and a profile matcher that can automatically provide a matching execution profile for a submitted MR job, even for jobs that have never been executed before on the cluster. PStorM employs a new set of features that are statically extracted from the byte code of the jobs, plus a set of dynamic features collected by executing a sample consisting of one map task and the reducers to process the output of this map task. PStorM uses a simple multi-stage domain-specific profile matching algorithm that outperforms a sophisticated and time consuming machine learning algorithm. We have shown experimentally that the combination of PStorM and Starfish achieves up to 9x speedup in runtimes compared to a default configuration.

7.2 Future Work

7.2.1 Sensitivity to User-Provided Parameters

Every MR job is developed by following a certain interface, and is executed using a well-defined framework. Therefore, the only differences between MR jobs are the customizable parts of this framework where the developer injects new logic. We explored in Section 4.1.2 some of these customizable parts as listed in Table 4.3.

We found that some jobs exhibit different execution profiles for different values of user-provided parameters, even when neither the job itself nor the input data set has changed. That is the reason the PStorM matching algorithm places the filter based on data flow statistics before the two filters based on static features, as shown in Figure 4.4. For example, the word co-occurrence job has different execution profiles at different window sizes. Also, different search keywords result in different execution profiles for the grep MR job.

An interesting direction for future work is incorporating the job parameters into the static feature vector, and investigating whether this can lead to more accurate matching results. With the addition of job parameters, the static feature vector can suffice to distinguish between different MR jobs stored in the profile store. Consequently, the dynamic feature vector would not be needed by the matcher, and hence even the small overhead of the 1-task sample profile collection can be eliminated.

7.2.2 Call Flow Graph Analysis

Another direction for future work is incorporating the *call flow graph* of the map and reduce functions into the feature vector used for profile matching. The call flow graph represents which functions in a program call which other functions, while the control flow graph represents the control flow within one function. In Section 4.1.3, we found that different control flow graphs of the map and reduce functions result in different CPU costs for the map and reduce tasks. Some map/reduce functions contain calls to different helper functions. Despite having similar control flow graphs, two functions can have very different execution profiles if they call different helper functions. Adding the call flow graphs of the map/reduce functions to the set of static features, and comparing the control flow graphs of the corresponding functions called, can provide better and more robust matching.

However, call flow graph extraction is not always possible by static code analysis alone, because of Java's object-oriented features such as polymorphism and inheritance. The

actual functions to be called are determined at runtime. Hence, more sophisticated dynamic code analysis is required, which may incur a high overhead. Investigating this overhead and reducing it can be part of this direction for future work.

7.2.3 Using PStorM on Different Clusters

In this thesis, we assumed that PStorM would be run on one cluster. In environments such as Amazon EC2, users can create different clusters on demand. It would be interesting to explore in future work how PStorM can accommodate profiles collected from different cluster sizes in one profile store. For example, studying whether the profiles collected on one cluster can be used for bootstrapping PStorM on another cluster.

7.2.4 Integration with PerfXplain

PerfXplain [22] is an automatic performance explanation system. PerfXplain analyzes the execution profiles collected for different MR jobs executed on the cluster in order to provide explanations for the difference between the observed performance of one job and its expected performance.

PerfXplain has two inputs: the log of past MR job executions and a query that asks for the explanation of the performance difference between a certain pair of jobs. The log contains the performance characteristics of the executed jobs measured at different phases of the map/reduce tasks. Features extracted from this log are the same as the dynamic features extracted from the Starfish execution profiles. One could envision that PerfXplain can get the dynamic features that it needs from PStorM, and also that PerfXplain can enrich its explanations with the addition of static features that are part of a PStorM profile.

The addition of the static features can lead to more accurate and precise performance explanations. For example, using different input formatters is one possible explanation for the difference in the IO cost of reading input data between two jobs of interest. Also, different control flow graphs of the map functions is an explanation for the difference in map-side CPU cost between the two jobs. These new explanations cannot be provided by PerfXplain, because it relies only on dynamic features extracted from a log of execution profiles.

In addition, the ability of PerfXplain to correlate different performance observations in order to provide explanations can be exploited to correlate different dynamic feature values

and static feature values. Therefore, PerfXplain can provide predictions for dynamic feature values by analyzing the submitted job statically without running any sample tasks of this job. These predictions can be useful for debugging purposes or for runtime prediction.

7.2.5 Feedback-Based Tuning of Dataflow Programs

This thesis focused on feedback-based tuning at the level of individual MR jobs. However, the analysis of big data often involves running not individual MR jobs but rather workflows of MR jobs that are generated by query languages such as HiveQL [39] and PigLatin [30].

For future work, PStorM can be extended to store the execution plans of PigLatin scripts or HiveQL queries, along with the execution profiles collected during the execution of the MR jobs that make up these plans. This new information can be useful for optimizing these execution plans [25] or for partial output materialization of sub-plans to be reused by other similar plans [8].

7.2.6 PStorM as a Service in the Cloud

PStorM makes use of the profiles collected during the execution of jobs on a certain cluster in order to provide matching profiles for new jobs submitted to the same cluster. Another direction for future work is to provide PStorM as a service in the cloud, such that PStorM can leverage the profiles collected on various clusters to provide matching profiles for new jobs submitted for execution on other different clusters. The challenge is that the profiles collected on different clusters have different profile cost factors, which play a crucial role in optimizing the configuration parameters. An initial step in this direction is presented in [14], where a new system is proposed to answer queries related to performance prediction on different cluster sizes. For example, given the performance of a job on a certain cluster, what is the predicted performance of the same job on another cluster? PStorM can make use of the system proposed in [14] to develop techniques to adjust the PStorM profile composed from profiles collected on clusters that are different from the current cluster.

References

[1] Ashraf Aboulnaga, Peter Haas, Mokhtar Kandil, Sam Lightstone, Guy Lohman, Volker Markl, Ivan Popivanov, and Vijayshankar Raman. Automated statistics collection in DB2 UDB. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

[2] Parag Agrawal, Daniel Kifer, and Christopher Olston. Scheduling shared scans of large data files. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2008.

[3] Amir Ahmad and Lipika Dey. A k-mean clustering algorithm for mixed numeric and categorical data. *Data and Knowledge Engineering*, 63(2), 2007.

[4] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 1999.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of the Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Jeffrey Dean and Sanjay Ghemwat. MapReduce: Simplified data processing on large clusters. In *Proc. of the Symp. on Operating Systems Design and Implementation (OSDI)*, 2004.

[7] Fernando Diaz, Donald Metzler, and Sihem Amer-Yahia. Relevance and ranking in online dating systems. In *Proc. of the Int. Conf. on Information Retrieval (SIGIR)*, 2010.

[8] Iman Elghandour and Ashraf Aboulnaga. ReStore: Reusing results of MapReduce jobs. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2012.

[9] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[10] David W. Goodall. A new similarity index based on probability. *Biometrics*, 22(4), 1966.

[11] Apache Hadoop. `http://hadoop.apache.org/`, 2012.

[12] Oktie Hassanzadeh and Mariano Consens. Linked movie data base. In *Proc. of the Workshop on Linked Data on the Web (LDOW)*, 2009.

[13] Apache HBase. `http://hbase.apache.org/`, 2012.

[14] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of the ACM Symp. on Cloud Computing (SOCC)*, 2011.

[15] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[16] Herodotous Herodotou. Hadoop performance models. Technical Report CS-2011-05, Duke University, 2011.

[17] Herodotous Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2011.

[18] Zhexue Huang. Clustering large data sets with mixed numeric and categorical values. In *Proc. of the Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD)*, 1997.

[19] Hadoop configuration guidelines. `http://www-01.ibm.com/support/docview.wss?uid=swg21573025`, 2011.

[20] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for MapReduce programs. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2011.

[21] Marina G. Kalyuzhnaya, David A. C. Beck, and Ludmila Chistoserdova. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 495, 2011.

[22] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. PerfXplain: Debugging MapReduce job performance. *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2012.

[23] YongChul Kwon, Herodotos Herodotou, Magdalena Balazinska, Rozemary Scarlat, Bill Howe, and Shivnath Babu. Hadepot: Repository of MapReduce applications. `http://nuage.cs.washington.edu/repository.php`, 2012.

[24] Cen Li and Gautam Biswas. Unsupervised learning with mixed numeric and nominal data. *IEEE Trans. on Knowledge and Data Engineering*, 14(4), 2002.

[25] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for MapReduce workflows. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2012.

[26] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool, 2010.

[27] Todd Lipcon. 7 tips for improving MapReduce performance. `http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/`, 2009.

[28] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. WebDocs: a real-life huge transactional dataset. `http://fimi.ua.ac.be/data/webdocs.pdf`, 2012.

[29] Apache Mahout: Scalable machine learning and data mining. `http://mahout.apache.org/`, 2012.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, 2008.

[31] OpenTSDB - A Distributed, Scalable Monitoring System. `http://opentsdb.net/schema.html`, 2011.

[32] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. Same queries, different data: Can we predict query performance? In *Proc. of the Int. Workshop on Self Managing Database Systems (SMDB)*, 2012.

[33] GroupLens Research. Movielens data sets. `http://www.grouplens.org/node/73`, 2011.

[34] Greg Ridgeway. *Generalized Boosted Models: A guide to the GBM package*, 2007.

[35] Michael C. Schatz. CloudBurst: Highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11), 2009.

[36] Sanjay Sharma. Advanced Hadoop tuning and optimizations. `http://www.slideshare.net/ImpetusInfo/ppt-on-advanced-hadoop-tuning-n-optimisation`, 2009.

[37] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[38] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2009.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, 2010.

[40] Apache Hadoop Vaidya guide. `http://hadoop.apache.org/docs/mapreduce/current/vaidya.html`, 2011.

[41] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative research (CASCON)*, 1999.

[42] Zhaohui Zheng, Hongyuan Zha, Tong Zhang, Olivier Chapelle, Keke Chen, and Gordon Sun. A general boosting method and its application to learning ranking functions for web search. In *Proc. of the Neural Information Processing Systems Conf. (NIPS)*, 2007.

APPENDICES

Appendix A

R Code Used to Build the GBRT Model

```
library("gbm", lib.loc="{R_DIR}/lib")

working_directory = "{WORKING_DIR}"
learning_dataset_filename = paste(working_directory,
                                  "profile_pairs_distances_after_WIF.txt",
                                  sep="")

distances_Lsd_filename = paste(working_directory,
                               "distances_Lsd.txt", sep="")
distances_Ldd_filename = paste(working_directory,
                               "distances_Ldd.txt", sep="")
distances_Lnj_filename = paste(working_directory,
                               "distances_Lnj.txt", sep="")

predictions_Lsd = paste(working_directory,
                        "distances_Lsd_predicted.txt", sep="")
predictions_Ldd = paste(working_directory,
                        "distances_Ldd_predicted.txt", sep="")
predictions_Lnj = paste(working_directory,
                        "distances_Lnj_predicted.txt", sep="")

# Load the dataset file
```

```
dataset = read.table(learning_dataset_filename, header=FALSE, sep="\t")

dataFrame <- data.frame(Y=dataset$V10,X1=dataset$V2,X2=dataset$V3,
                        X3=dataset$V4,X4=dataset$V5,X5=dataset$V6,
                        X6=dataset$V7,X7=dataset$V8,X8=dataset$V9)

# For binary outcome, use bernolli or adaboost
# For continuous outcomes, use gaussian to minimize the squared error, or
# laplace to minimize the absolute error.
# Outcome here is the difference in the predicted runtimes using the WIF.

gbm1 <- gbm(Y~X1+X2+X3+X4+X5+X6+X7+X8,        # formula
            data=dataFrame,                   # dataset
            var.monotone=c(0,0,0,0,0,0,0,0),
            distribution="laplace",
            n.trees=10000,                    # number of trees
            shrinkage=0.001,                  # shrinkage or learning rate,
                                              # 0.001 to 0.1 usually work
            interaction.depth=3,              # 1: additive model,
                                              # 2: two-way interactions, etc
            bag.fraction = 0.5,               # subsampling fraction
            train.fraction = 1,               # fraction of data
                                              # for training,
                                              # first train.fraction*N
                                              # used for training
            n.minobsinnode = 10,              # minimum number of obs
                                              # needed in each node
            keep.data=TRUE,
            cv.folds=10)

best.iter <- gbm.perf(gbm1,method="cv")

# Make predictions for Lsd:
distances_Lsd = read.table(distances_Lsd_filename, header=FALSE, sep="\t")
distances_Lsd$V10 <- rep(0, nrow(distances_Lsd))
distances_Lsd_frame <-
    data.frame(Y=distances_Lsd$V10,X1=distances_Lsd$V2,X2=distances_Lsd$V3,
               X3=distances_Lsd$V4,X4=distances_Lsd$V5,X5=distances_Lsd$V6,
```

```
            X6=distances_Lsd$V7,X7=distances_Lsd$V8,X8=distances_Lsd$V9)

distances_Lsd_predict <- predict(gbm1,distances_Lsd_frame,best.iter)
distances_Lsd$V10 <- distances_Lsd_predict

write.table(distances_Lsd, file=predictions_Lsd, quote = FALSE, sep = "\t",
           row.names=FALSE, col.names=FALSE)

# Make predictions for Ldd:
distances_Ldd = read.table(distances_Ldd_filename, header=FALSE, sep="\t")
distances_Ldd$V10 <- rep(0, nrow(distances_Ldd))
distances_Ldd_frame <-
    data.frame(Y=distances_Ldd$V10,X1=distances_Ldd$V2,X2=distances_Ldd$V3,
            X3=distances_Ldd$V4,X4=distances_Ldd$V5,X5=distances_Ldd$V6,
            X6=distances_Ldd$V7,X7=distances_Ldd$V8,X8=distances_Ldd$V9)

distances_Ldd_predict <- predict(gbm1,distances_Ldd_frame,best.iter)
distances_Ldd$V10 <- distances_Ldd_predict

write.table(distances_Ldd, file=predictions_Ldd, quote = FALSE, sep = "\t",
           row.names=FALSE, col.names=FALSE)

# Make predictions for Lnj:
distances_Lnj = read.table(distances_Lnj_filename, header=FALSE, sep="\t")
distances_Lnj$V10 <- rep(0, nrow(distances_Lnj))
distances_Lnj_frame <-
    data.frame(Y=distances_Lnj$V10,X1=distances_Lnj$V2,X2=distances_Lnj$V3,
            X3=distances_Lnj$V4,X4=distances_Lnj$V5,X5=distances_Lnj$V6,
            X6=distances_Lnj$V7,X7=distances_Lnj$V8,X8=distances_Lnj$V9)

distances_Lnj_predict <- predict(gbm1,distances_Lnj_frame,best.iter)
distances_Lnj$V10 <- distances_Lnj_predict

write.table(distances_Lnj, file=predictions_Lnj, quote = FALSE, sep = "\t",
           row.names=FALSE, col.names=FALSE)
```

Appendix B

Details of the Rule-Based Optimizer

We built our own Rule-Based Optimizer (RBO) by extracting different rules recommended by different Hadoop performance tuning experts. We used this RBO as an alternative against which to evaluate the Starfish CBO. In this appendix, the rules we used to build our RBO are listed along with a recipe of when and how to apply each rule.

mapred.compress.map.output

- Definition: This parameter is used to enable compression of the intermediate records generated by the mappers.
- Default Value: False
- Pros: Setting this parameter to true decreases the intermediate data size that is shuffled over the network to the reducers.
- Cons: Setting this parameter to true increases the CPU time on the map side to compress the intermediate records and on the reduce side to decompress these records.
- Suggestion [27, 36]: Set this parameter to true when the map function is expected to generate more intermediate records than the input records, or when the intermediate records are expected to be large, e.g., in the join job that uses the CompositeInput-Format class as the input formatter.

io.sort.mb

- Definition: The buffer size (in MB) where the output of the map function is serialized.

- Default Value: 100 MB
- Pros: Larger values increase the number of intermediate records serialized into this buffer, and hence reduce the number of spills to disk.
- Cons: Larger values increase the CPU time used to sort the intermediate records in this buffer before each spill to disk.
- Suggestion [36]: Increase the value of this parameter for jobs with larger size/number of intermediate records compared to the size/number of input records.

io.sort.record.percent

- Definition: The percentage of the map-side buffer (io.sort.mb) that is used to store meta-data about the serialized intermediate records in that buffer. A value of 0.05 means that 5% of the map-side buffer is used to store the meta-data of the intermediate records, and 95% is used for the intermediate records themselves. A map-side buffer spill occurs when the buffer space used to store either the intermediate records or their meta-data reaches a certain threshold (expressed as a percentage of the buffer size).
- Default Value: 0.05
- Pros: Higher values can sometimes decrease the number of buffer spills to disk.
- Cons: There are also cases where higher values can increase the number of buffer spills.
- Suggestion [36]: When the size of each intermediate record is small, set this parameter to larger values in order to accommodate more meta-date of more intermediate records before a buffer spill occurs.

Combiner usage

- Definition: Run the combiner on the map side, where the combine function is the same as the reduce function.
- Default Value: Disabled
- Pros: When enabled, the size of intermediate records shuffled from each mapper to the reducers over the network is reduced.

- Cons: Slightly more CPU cycles on the map-side, but that is negligible compared to the network cost.
- Suggestion [27]: Always enable the combiner whenever the reduce function is associative and commutative, e.g., sum, min., and max.

mapred.reduce.tasks

- Definition: Number of reduce tasks launched for any MR job.
- Default Value: 1
- Pros: Higher values of this parameter increase reduce-side parallelism.
- Cons: Very high values can lead to fewer intermediate records shuffled to each reducer, and hence the time of each reduce task will be dominated by its setup and cleanup times.
- Suggestion [19]: Set this value to 90% of the number of available reduce slots on the cluster (*mapred.tasktracker.reduce.tasks.maximum* * number of TaskTrackers). The effect of this is to run all the reducers in one wave even when some reducers suffer from delays or failures.