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

Mostafa Ead*
Amazon

Herodotos Herodotou†
Microsoft Research

Ashraf Aboulnaga*
Qatar Computing Research Institute

Shivnath Babu
Duke University

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 make them black-box in nature and therefore renders it difficult for users and administrators to make good parameter tuning decisions for a submitted MapReduce job. An approach that is gaining popularity is to provide automatic tuning decisions for submitted MapReduce jobs based on feedback from previously executed jobs. This approach is adopted, for example, by the Starfish system. Starfish and similar systems base their tuning decisions on an *execution profile* of the MapReduce job being tuned. This execution profile contains summary information about the runtime behavior of the job being tuned, and it is assumed to come from a previous execution of the same job. Managing these execution profiles has not been previously studied. This paper presents PStorM, a profile store and matcher that accurately chooses the relevant profiling information for tuning a submitted MapReduce job from the previously collected profiling information. PStorM can identify accurate tuning profiles even for previously unseen MapReduce jobs. PStorM is currently integrated with the Starfish system, although it can be extended to work with any MapReduce tuning system. Experiments on a large number of MapReduce jobs demonstrate the accuracy and efficiency of profile matching. The results of these experiments show that the profiles returned by PStorM result in tuning decisions that are as good as decisions based on exact profiles collected during pervious executions of the tuned jobs. This holds even for previously unseen jobs, which significantly reduces the overhead of feedback-driven profile-based MapReduce tuning.

1. INTRODUCTION

The MapReduce (*MR*) programming model [4] has become widely adopted for large scale data analytics in many organizations. MR systems, the most popular being Hadoop [10], have many tuning parameters, and these parameters have a significant impact on performance. Some example tuning parameters are: the amount of memory used for sorting intermediate results, the number of reduce tasks, and the number of open file handles. Setting MR 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 varying 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. Recent advances in Hadoop, particularly YARN [35] provide richer mechanisms for assigning resources to MR jobs, but do not answer the question of *how* to decide the amount of resources to give to a job. Thus, YARN and similar technologies make it even more important to make accurate tuning decisions for MapReduce.

Tuning MR jobs is difficult due to the complexity and scale of the software and the underlying clusters on which these jobs run. The MR 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 MR tuning is to adopt a *feedback-based* approach to tuning. In this approach, the system collects information about the execution of MR 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 MR complexity on performance is captured in the profiles, which makes feedback-based tuning simpler and more robust than approaches that do not use feedback (such as rule-based tuning).

Feedback-based tuning using execution profiles is used in the Starfish system [14] and also in [27]. Execution profiles are also used in the PerfXplain system [20] 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 duration, and memory used. The profiles can also be based on MR-specific information collected from the execution of an instrumented MR job. As an example, Figure 1 shows some of the MR-specific information in a profile from the Starfish system (which we use in this paper). As the figure shows, an execution profile can contain information collected at runtime from an instru-

*Work done at the University of Waterloo.
†Work done at Duke University.

```
<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: An Execution Profile from the Starfish System

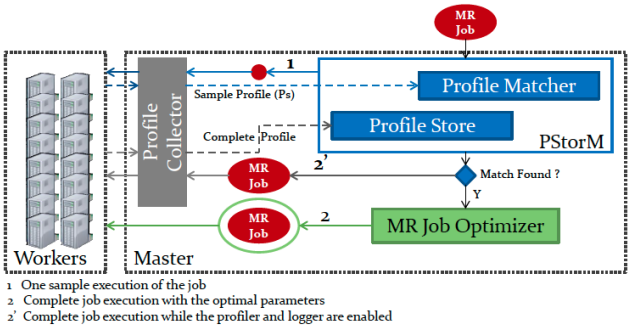


Figure 2: PStorM Architecture

mented MR 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 MR 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 MR [14, 20] 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 MR 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. Identifying that a job is the same as a previously submitted one is typically based on the program name or on a hash of the program byte code. Furthermore, the execution profile of one MR job is not used for tuning another job, even if the two jobs are similar.

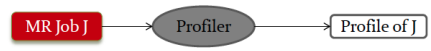
MR jobs submitted to a cluster can be expected to have some similarity since they process the same data and often reuse the same map or reduce functions. These map and reduce functions are often part of a library shared by all users. Even if the functions are not part of a library, users commonly create new MR jobs by modifying the code of existing jobs. In addition, users often submit refinements of the same program to the cluster during a session. Thus, a new MR job submitted to a cluster will frequently be similar to some MR job previously executed on the same cluster. The similarity between MR jobs is likely to be higher if the jobs are generated from high-level query languages such as Pig Latin [26] or Hive [32]. However, all this similarity is ignored by current systems that use profile-driven tuning; profiles are collected independently and a profile of one job is not used for tuning other jobs.

Even identifying that a job is identical to a previously executed job can be problematic for current systems. Relying on program or function names can be inaccurate since users commonly modify programs and substantially change their behavior while keeping the program and function names unchanged. Relying on a hash value of the program byte code can be overly conservative, since recompiling a program typically results in changed byte code.

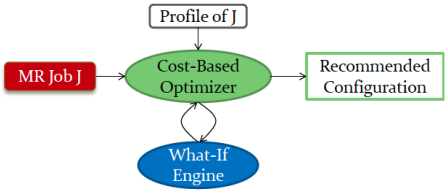
This paper addresses these problems and presents *PStorM*, a Profile Store and Matcher for feedback-based tuning of MR jobs. Figure 2 presents the PStorM architecture. PStorM stores all the execution profiles collected from different runs of MR jobs on the cluster, and uses these stored profiles to provide an accurate profile for a newly submitted MR job. PStorM consists of two components: a 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 MR jobs*. The profile matcher can also provide a *composite* profile for an MR 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, thereby reducing the overhead of feedback-based tuning and increasing its applicability.

The quality of 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 paper, 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 the next section). 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 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. 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 necessarily mean that the jobs are functionally equivalent. It simply means that the runtime behav-



(a) First Job Submission



(b) Subsequent Job Submissions

Figure 3: Starfish Tuning Workflow

ior 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 MR uses a very stylized form of programs and a very stylized form of job execution, and since jobs executed on a cluster often have a high degree of similarity.

The contributions of this paper are summarized as follows:

- A set of novel features of MR jobs that effectively distinguishes among job profiles (Section 3).
- A set of similarity measures for use with different types of features (Section 4).
- A domain specific multi-stage algorithm for matching profiles. The matching algorithm can create an output profile from different stored profiles, which is useful for previously unseen jobs (Section 5).
- A profile store that organizes the profile information collected from MR jobs (Section 6).

2. SYSTEM OVERVIEW

PStorM is composed of two main components: the profile store and the profile matcher. The profile store is an instance of an HBase database [12] 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 paper, PStorM is used to store execution profiles collected by Starfish [14], and the profiles returned by PStorM are used for Starfish optimization (Figure 2). We therefore present an overview of Starfish and its optimization workflow. Starfish is a system for feedback-based tuning of Hadoop MapReduce jobs. It is composed of three main components: *profiler*, *what-if engine (WIF engine)*, and *cost-based optimizer (CBO)*. Starfish uses the tuning workflow shown in Figure 3. The first time an MR job is submitted, the profiler collects an execution profile for this job. This execution profile contains fine-grained data flow and cost statistics about the execution of every phase in the map and reduce tasks of the job. Starfish stores the collected execution profiles in a file-based hierarchy with very simple storage and retrieval based on file names. In subsequent submissions of an MR job, the WIF engine uses cost models for different aspects of job execution to predict the runtime of the job given its profile. The CBO searches the space of possible configuration parameters and recommends the optimal configuration parameters for the new submission of the job. PStorM is meant to replace the profiler in Figure 3. Therefore, the role of PStorM, and the profile matcher in particular, is to find the job profile that helps the CBO find the optimal configuration for a submitted job.

If a complete profile of a job being optimized by Starfish is not available, Starfish can collect a profile for use by the CBO by running a sample of the tasks that make up this job [14]. Sampling exposes a tradeoff between the cost of profiling and the accuracy of the collected profile. Sampling more tasks incurs runtime overhead and consumes some of the task execution slots available on the cluster. 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 tasks of a job. In our work we have verified that, in most cases, tuning decisions based on a 10% sample do indeed provide speedups comparable to tuning decisions based on a full profile. The goal of PStorM is to achieve lower overhead than even the 10% sample.

For each MR job submitted to the cluster, PStorM uses the Starfish sampler to run a sample consisting of exactly one map task and the reducers required to process the output of this map task (Figure 2). A sample profile is collected from this run and 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 is provided to the Starfish CBO which recommends suitable parameter values for the submitted job. If no matching profile is found, the job is executed with profiling turned on. The collected profile from this execution is stored in the profile store and used for tuning future submissions of this job or other similar jobs.

The reason that PStorM requires less sampling than Starfish is that the purpose of sampling in the two systems is fundamentally different. 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.

As an alternative to the CBO, we also implemented a *Rule Based Optimizer (RBO)* for MapReduce jobs. The rule based optimizer is based on aggregating rules of thumb from different expert sources that specialize in Hadoop tuning [17, 23, 31, 33]. Our RBO consists of the union of sets of rules found at the various sources. These rules apply to distinct cases and do not conflict, and the resulting RBO does typically result in better runtimes than the default parameter settings. However, there are cases in which the RBO is actually *worse* than the default parameter settings. An RBO is an attempt to capture tuning expertise, but it is not as good as a cost-based profile-driven tuning.

Next, we discuss the various components of the 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 3), data preprocessing and normalization, finding the appropriate similarity measures (Section 4), the pattern matching workflow (Section 5), and finally thresholds adjustment.

3. 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 in order to identify the features that play an important role in its runtime predictions. As outlined in [13], 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.

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 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 2: Profile Cost Factors

- Data flow Statistics: A set of profile attributes that specify input/output data properties of the map, combine, and reduce tasks (examples in Table 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 2).

These data flow statistics and cost factors are extracted from the profile that is provided to the CBO (Figure 3).

Suppose that a job, J, was executed on the cluster, and the profile collected for it 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 contains similar data flow statistics and cost factors to P_J. The configuration parameters are supplied by CBO. Hence, the space of features that represent a profile is narrowed down to data flow statistics and cost factors of the profiled job.

3.1 Dynamic Features

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 and the reducers required to process the output of this map task. The number of reducers is specified by the current Hadoop configuration parameters, and the scheduling of these reducers is handled by the normal Hadoop task scheduler. During the execution of this sample, the profiler collects the sample profile P_s. The overhead for collecting the profile P_s is low, and 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 profile based on a sample of 10% of the map tasks, plus the reducers. Figure 4 shows the overhead of *10% profiling* and 1-task sampling for different MR jobs (details of these jobs are given in Table 5, and the jobs are executed on the 35GB Wikipedia data set). The overhead for each job is presented as a fraction of the runtime of the job when using the configuration recommended by the RBO while the profiler is turned off. In addition to having lower overhead, 1-task sampling consumes only one map slot while 10% profiling consumes 57 map slots (the data set was stored

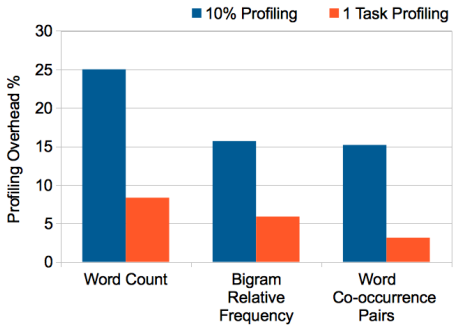


Figure 4: Profiling Overhead for 10% Profiling and 1 Task Profiling as a Fraction of the Job Runtime using the RBO Recommendations Without Profiling

in 571 HDFS splits). Thus, we see that 1-task sampling has minimal effect on the the response time of individual MR jobs and 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 this requirement. These data flow statistics, which we use for matching, are the ones shown in Table 1.

As an example to illustrate why these features are effective, consider the map size selectivity feature (MAP_SIZE_SEL). The map size selectivity of a sorting MR job is 1 for all sample profiles of that job (i.e., for all map tasks). 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. 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 cannot be used for matching because the values of these features can exhibit high variance among sample profiles of the same job. For example, the IO cost to read from HDFS can differ be-

Feature	Description
IN_FORMATTER	Input formatter class name
MAPPER	Mapper class name
MAP_IN_KEY	Input key data type
MAP_IN_VAL	Input value data type
MAP_CFG	Control flow graph of the map function
MAP_OUT_KEY	Intermediate key data type
MAP_OUT_VAL	Intermediate value data type
COMBINER	Combiner class name
REDUCER	Reducer class name
RED_OUT_KEY	Output key data type
RED_OUT_VAL	Output value data type
RED_CFG	Control flow graph of the reduce function
OUT_FORMATTER	Output formatter class name

Table 3: Static MR Job Features

tween two samples of the same job just because they read input splits whose size is different. 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 common case in any Hadoop cluster, and is one of the reasons MR has a straggler handling mechanism [4]. Thus, we cannot use the profile cost factors collected from the 1-task sample profile. Instead, we rely on static analysis to extract features of the MR job that provide indications about the cost factors, as we discuss next.

3.2 Static Features

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 equal to the number of reducers. Each reducer starts by shuffling its designated partition from all mappers to its local machine. 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 key partitioner, intermediate key 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 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 an MR job that joins two inputs is *CompositeInputFormat* and the input formatter of a word count MR job is *TextInputFormat*. Different input formatters lead to different READ_HDFS_IO_COST values in the map-side profiles. Similarly, 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.

3.3 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 and in the byte code generated by the Java compiler. 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 graph representing 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 [34] to extract these CFGs, and we add them to our set of static features.

Algorithm 1 Map Function of the Word Count Job

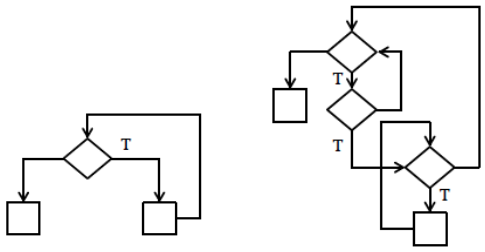
```
function MAP(Object key, Text line, Context context)
    iterator ← line.tokenize()
    while iterator.hasMoreTokens() do
        word ← iterator.currentToken()
        context.write(word, 1)
    end while
end function
```

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
```

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 Section 7. The map function of the word count job contains one loop, which is represented as a cycle in the CFG, shown in Figure 5(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 5(b). The CFGs are quite different and can help distinguish between the jobs. Moreover, different CFGs entail different values of the MAP_CPU_COST and REDUCE_CPU_COST features, which are part of the profile cost factors. Thus, we can use CFGs as proxies for comparing profile cost factors since these factors cannot be extracted directly from the 1-task samples, as discussed earlier.

Comparing the CFGs of the map and reduce functions of a job is more robust than comparing hash values of the source code or byte code 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 behavior, and will be matched if comparing



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

Figure 5: CFGs of the Map Functions of the Word Count and Word Co-occurrence MR Jobs

the CFGs. However, comparing source or byte code will result in a mismatch between these two versions of the word count job.

Matching programs based on their CFGs can be extremely complex, and is undecidable in the general case. 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 rely on the CFG but rely on the other features instead. Also, since the jobs being matched are MR jobs which follow a restricted programming model, the likelihood of finding a match based on the CFG is increased.

4. SIMILARITY MEASURES

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 (categorical and numerical).

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 paper we use the Jaccard index to match the static features, since it is a simple similarity measure that incurs low computation cost, and has been shown to outperform the other similarity measures [11]. 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 paper.

$$\begin{aligned}
CFG &\models Statement \\
Statement &\models normal_stmt \mid BranchStatement \\
BranchStatement &\models branch_cond\ IsLoop\ Successors \\
IsLoop &\models true \mid false \\
Successors &\models Statement\ Statement\ ExpCatchStmt \\
ExpCatchStmt &\models caught_exp \mid \varepsilon
\end{aligned}$$

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.

5. STEP-WISE 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 MR cluster, and ending with a matching profile retrieved from the profile store (if a match is found) and used by the CBO.

When a job is submitted to the cluster, its byte code is analyzed to extract the static features. In addition, 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 sampling 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 by analyzing 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 algorithm based on our domain knowledge, such that the distance between features of different types (numerical or categorical) is calculated in different stages of matching.

The profile matching workflow is shown in Figure 6, and it is applied twice, once for map-profile matching and once 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 only one candidate is left. That candidate is the matched profile returned by PStorM. 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 control

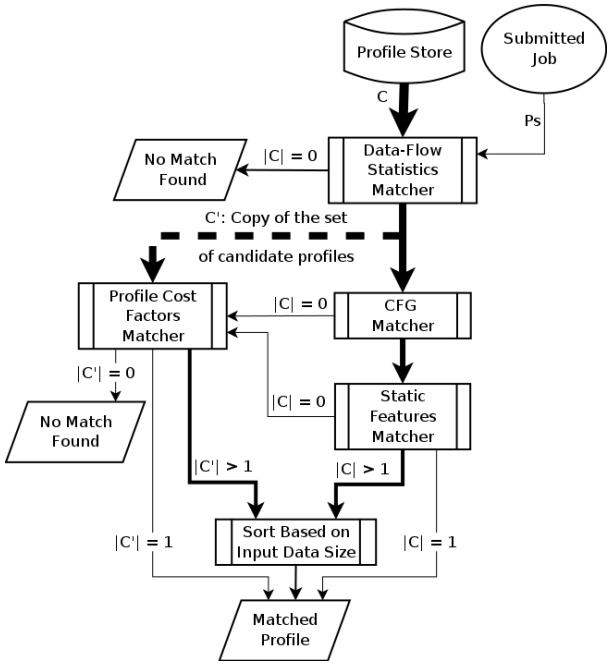


Figure 6: The Map/Reduce Profile Matching Workflow

flow graph 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 third filters. An empty set after these filters is interpreted to mean that the submitted job was never executed before on the cluster. In this case, an alternative filter is applied. Recall that the set C' contains profiles whose dynamic features (Table 1) have Euclidean distance less than θ_{Eucl} to P_s. The Euclidean distance between the profile cost factors (Table 2) of each job profile in C' and P_s is calculated, and jobs in C' with distances larger than the defined θ_{Eucl} are excluded. Then, the profile of the job whose input data size is closest to the submitted job is returned by the matcher.

The profile matcher returns *No Match Found* when the set of candidate job profiles after 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 with profiling turned on. The collected profile is stored in PStorM to be used for future matching.

If matching succeeds, 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 subprofiles 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 7.2) 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.

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.

If more than one job profile remains in the candidate set at the end of profile matching, we use the input data size to break ties and select one job profile to return. The reason for this tie breaking 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.

5.1 Alternative Matching Technique

Profile matching can be viewed as a generic nearest neighbor problem for which we need to define a suitable distance metric that can be used to compare feature vectors. The feature vectors in PStorM include numerical and categorical features, so we need a generalized distance metric that can handle both types of features simultaneously. The approach adopted in the pattern recognition literature [2, 7, 16, 21] is to calculate a distance value for each type of feature, and then combine these distance values using a weighted sum. The weights used in this weighted sum are learned using regression analysis on a training data set. Each point in the training data set consists of two feature vectors, the distances between the features of different types in these vectors, and an overall distance representing how close the two vectors are to each other. The regression analysis applied on this training data aims to find weights that make the weighted sum of the distances between different types of features as close as possible to the overall distance.

In this paper, we construct the training data set from a set of job profiles as follows. For each point (or sample) in the training data set, we need a pair of job profiles (or feature vectors). We use as the first profile in that pair the complete profile of a job J. The second profile in the pair is a profile made up 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 be the same job (i.e., we can have a complete profile or a composite profile). This approach to generating job profiles for the training data ensures that this data contains composite profiles, so that the learned model can be used for composite profiles.

In addition to the two job profiles, each training sample has distance/similarity metrics that measure the distance between different types of features in the two profiles. We record four metrics for the distance between the map profiles, and the same four for the distance between the reduce profiles. These four distance metrics are: (1) the Jaccard distance between the static features of the two profiles, (2) the Euclidean distance between the dynamic features of the two profiles, (3) the Euclidean distance between the profile cost factors of the two profiles, and (4) the result of matching the CFGs of the two jobs. The final (ninth) value in the training sample is the difference between the runtime predicted by the Starfish 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

profile. This value represents an overall measure of how well the two profiles match 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 is zero. Thus, such a sample provides the machine learning algorithm with an example of a perfect match.

A state-of-the-art learning algorithm that is used in this kind of learning problem and provides good results [5, 36] is *Gradient Boosted Regression Trees (GBRT)* [29]. GBRT produces the learned model in the form of an ensemble of decision trees. We used an implementation of this technique in the R [28] statistical software package to calculate the weights that combine the partial distances into a generalized distance metric (Equation 1). When finding a matching profile for a submitted MR job, we return the job in the profile store that has the smallest distance to the submitted job according to the learned distance metric (i.e., the nearest neighbour according to this metric).

Section 7.1 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 that may need to be repeated periodically as the profile store grows.

6. PROFILE STORE

The other component of PStorM is its profile store, which provides a repository of the profiles collected on the cluster. We emphasize that the profile matcher does not dictate any specific structure on the profile store. PStorM can return a profile for a submitted MR job whether the collected profiles are stored in flat files, a relational database, a NoSQL store, or any other type of data store. However, judicious design choices for the profile store can significantly improve performance and facilitate the implementation of matching and other functionality.

In PStorM, we adopt HBase [12] as the storage system for the profile store. A discussion of other alternatives that we considered can be found in [6]. HBase is a distributed column-family oriented data store that scales in the number of rows and the number of columns. HBase scales in rows by horizontal partitioning and replication mechanisms, and it scales in columns by physically storing columns of each column family in different files.

We chose HBase for several reasons. First, HBase is scalable in the number of rows so it can handle the large number of job profiles that would be collected on a cluster. Each job profile is on the order of only a few hundred bytes in size, but scalability is required since the number of profiles grows as the cluster is used. Second, the indexing provided by HBase ensures fast access during profile matching. Third, the type of updates needed for the profile store is efficiently supported by HBase. Updates to the profile store consist of adding new profiles as jobs get executed, and possibly deleting old profiles to free up space. There are no in-place modifications. This is precisely the type of updates that HBase is designed for. Fourth, HBase is integrated with Hadoop and enables the profile store to support *analytical* workloads. In this paper we focus on exact matching to retrieve profiles for a submitted MR job. However, we envision that the PStorM profile store can be used by other Hadoop job analysis and optimization systems, e.g., PerfXplain [20] and Manimal [18]. These systems may also require exact match retrieval, but using HBase enables them to run analytics-style scans of the profile store using Hadoop. Thus, the profile store becomes a basis for complex analysis and tuning of job performance on the Hadoop cluster. Fifth, HBase uses an *extensible* data model, so it is possible to incorporate new types of data that are necessary for other job optimization and analysis systems. Finally, HBase is a good candidate for the PStorM profile store since HBase is part of the Hadoop ecosystem and stores data in HDFS. Hence, no new infrastructure components and few new daemons need to be added to the cluster.

Row-Key	**Column Family**			
	Col. Name	**Col. Name**	**Col. Name**	**Col. Name**
	CF			
	MAP_ IN_ KEY	RED_ OUT_ KEY	MAP_ SIZE_ SEL	RED_ SIZE_ SEL
Static/Job1	Integer	Text	-	-
Static/Job2	Long	Integer	-	-
Dynamic/Job1	-	-	1.0	1.0
Dynamic/Job2	-	-	11.5	0.26

Table 4: PStorM Schema in HBase

6.1 HBase Schema

Using HBase requires us to define a schema for the profile store. In HBase, like other NoSQL systems, the logical and physical designs are intertwined, so defining the schema has a significant impact on performance.

The data model in HBase is that data is stored 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. HBase physically stores data items as key-value pairs where the physical key is a composite key made up of the row-key, column family identifier, column name, and a timestamp. The value corresponding to this physical key is the column value corresponding to the row-key. Thus, to use HBase for the PStorM profile store, we need to design an HBase schema for profiles, which requires us to specify the row-key, column families, and columns within these families.

A simple schema for profiles collected for 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 physical key used by HBase would be (job ID, feature type, feature name). The value indexed by this key is the feature value. With this schema, the profile store is not extensible, since HBase does not allow adding column families once a table is created, and a new column family is required for a new feature type.

Instead, we organize the job information into another schema, illustrated in Table 4. The row-key is made up of the feature type as a prefix and the job ID. Only one column-family is used. Each col-

umn name is a feature name whose type is indicated by the prefix of the current row. For example, Table 4 shows two static features and two dynamic features for two MR jobs. This schema 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 in the rows whose prefix represents that feature type.

In addition, this schema boosts the performance of the profile matcher. As explained in Section 5, 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 according to the row key, and the feature type is a prefix of the row key.

7. 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 2 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 5.

As explained in Section 5, PStorM uses a 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 half of this maximum possible Euclidean distance.

7.1 PStorM Accuracy

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 the previous execution of the 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. In the second content state (DD), a correct match is the twin of that complete profile.

In the next two sections, we evaluate the accuracy of the domain-specific profile matcher used by PStorM, and we compare it to more generic alternatives.

7.1.1 Feature Selection

One of the contributions of PStorM is the set of static features proposed, which provide a good proxy for the profile cost factors and are more suitable than the cost factors because the cost factors exhibit high variance among sample task profiles of the same MR job. Another contribution is the domain-specific feature selection algorithm based on our Hadoop expertise, which handles 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 can be found in the collected Starfish job profile. A common machine learning approach is to rank these features according to their *information gain* scores [20]. The highest ranked F features are selected to be part of 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 stored profile whose distance from the 1-task sample profile of the submitted job is the lowest (i.e., the nearest neighbor) is selected as the matching profile 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 7 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

MapReduce Job	Application Domain	Data set
CloudBurst [30]	Bioinformatics	Sample genome and Lake Washington Genome [19]
Frequent Itemset Mining [25]	Data Mining	Webdocs data set of size 1.5GB [24]
Collaborative Filtering	Recommendation Systems	Movie rating data sets with 1M and 10M ratings [8]
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 [22]	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 Benchmark	1GB and 35GB of data generated by PigMix
Bigram Relative Frequency [22]	Natural Language Processing	1GB of random text and 35GB of Wikipedia docs
Word Co-occurrence Pairs [22]	Natural Language Processing	1GB of random text and 35GB of Wikipedia docs
Word Co-occurrence Stripes [22]	Natural Language Processing	1GB of random text

Table 5: Benchmark of Hadoop MapReduce Jobs

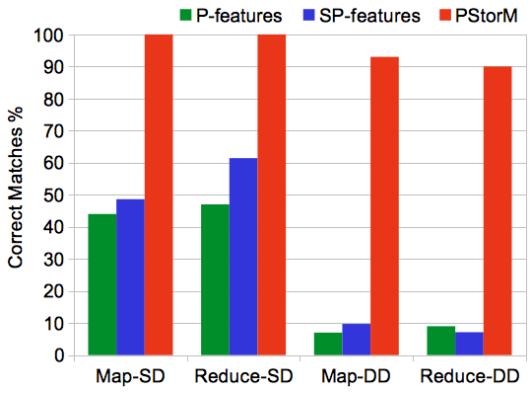


Figure 7: Correct Match Percentages for the Two Alternative Feature Selection Solutions (P-features and SP-features) vs. PStorM in the Two Content States of the Profile Store (SD and DD)

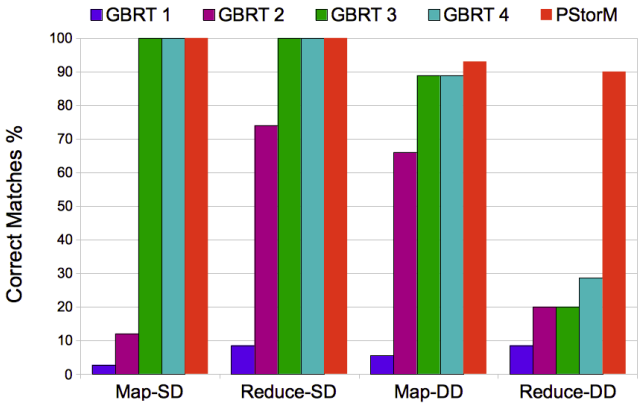


Figure 8: Correct Match Percentages for the Alternative Matching Solution (GBRT) with Different Parameter Settings vs. PStorM in the Two Content States of the Profile Store (SD and DD)

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).

7.1.2 Multi-Stage Profile Matching

As shown in the previous section, the set of features used by PStorM results in the best matching accuracy in both 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 algorithm presented in Section 5.

An alternative to the PStorM matcher is the GBRT matcher presented in Section 5.1. In this section, we compare the PStorM profile matcher with GBRT. Figure 8 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.

The first GBRT parameter setting (GBRT 1 in Figure 8) 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 to 10,000, the learning rate was set to 0.001 [29], 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 8.

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

Table 6: Job Runtimes with Default Hadoop Configuration

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 and as the profile store grows. On the other hand, PStorM results in high matching accuracy using a simple algorithm that does not need training.

7.2 PStorM Efficiency

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.

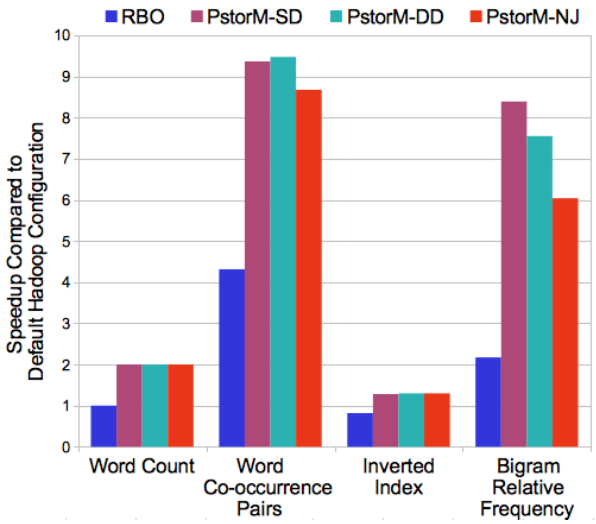


Figure 9: Speedups of Different MR Jobs Executed with the RBO Recommendations and the CBO Recommendations Based on a Profile Returned by PStorM at the Three Content States of the Profile Store (SD, DD, and NJ)

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 are shown in Table 6, and the speedups of different tuning options compared to this default are shown in Figure 9. The figure shows 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 9 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 9 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 consisting of the map profile of one job plus the reduce profile of another job. This composite profile guides the CBO to choose 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 results in tuning that is as good as using a complete, accurate profile 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 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 illustrate the types of tuning actions taken by the Starfish CBO, we look more closely at this job. The CBO (using the profile from PStorM) reduces the amount of memory used for sorting, increases the number of reduce tasks, and enables compression of mapper outputs, leading to the substantial speedups that we observe.

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.

8. RELATED WORK

Prior works have used profile-driven tuning of MapReduce jobs. Starfish is closely related to PStorM and the use of Starfish profiles for tuning was discussed earlier in the paper. Starfish profiles have also been used to determine cluster sizes for MR jobs [15]. Profiles returned by PStorM can be used for this application just as they are used for parameter tuning.

Another work that uses execution profiles for tuning is PerfXplain [20], which uses profiles for debugging MapReduce job performance and providing appropriate explanations for unexpected performance. PerfXplain allows the user to pose a performance question, and generates an explanation of why the user observed a different value of a certain performance measure than what was expected. PerfXplain classifies every pair of jobs based on their execution profiles as either matching the observed performance or matching the expected 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 job pairs into the aforementioned two classes. The profile store component of PStorM contains a wealth of information about MR jobs executed previously on the cluster, which can be used as a source of input for a tool like PerfXplain, leading to more precise and detailed explanations to the user.

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 were much simpler than what is required in PStorM.

In PStorM, we use static program analysis to extract CFGs that are used as features of MR jobs. The use of static program analysis to tune data flow programs has been explored in recent work. SatusQuo [3] uses program analysis to automatically convert imperative Java code in applications to SQL queries that execute in a database system. It identifies code fragments that manipulate lists of persistent data and have no side-effects. Like PStorM, it relies on the fact that code in applications is highly stylized so patterns are likely to be detected. PeriSCOPE [9] uses program analysis to reduce data movement in a pipeline of parallel dataflow jobs (executed in the SCOPE system). The type of program analysis and its objective are very different from PStorM.

9. CONCLUSION

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 executing MR jobs. A significant problem with feedback-based tuning approaches is providing the execution profile required for tuning a job. If the job has been executed before on the cluster, the challenge is identifying the correct profile to use from among all stored profiles. If the job is a previously unseed job, the challenge is composing a suitable profile for this job from the stored profiles without executing the job.

PStorM addresses these challenges through the use of a multi-stage domain-specific profile matching algorithm 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 also includes a scalable and extensible profile store based on HBase. This profile store supports the scalable and efficient retrieval of profiles required by the profile matcher. The profile store can also be extended to support other applications, including applications that perform complex analytics on the stored job profiles. The PStorM matching algorithm outperforms a sophisticated and time consuming matching algorithm based on machine learning, and in our experiments PStorM enables up to 9x speedup in runtimes compared to the Hadoop default configuration.

Acknowledgments: This work was partly funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Business Intelligence Network strategic networks grant, and by the US National Science Foundation (NSF) through grants 0917062 and 0964560.

10. REFERENCES

[1] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *VLDB*, 2004.

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

[3] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[4] J. Dean and S. Ghemwat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[5] F. Diaz, D. Metzler, and S. Amer-Yahia. Relevance and ranking in online dating systems. In *SIGIR*, 2010.

[6] M. Ead. PStorM: Profile storage and matching for feedback-based tuning of MapReduce jobs. Master's thesis, University of Waterloo, 2012.

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

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

[9] Z. Guo et al. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*, 2012.

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

[11] O. Hassanzadeh and M. Consens. Linked movie data base. In *Proc. of LDOW*, 2009.

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

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

[14] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB*, 2011.

[15] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.

[16] Z. Huang. Clustering large data sets with mixed numeric and categorical values. In *PAKDD*, 1997.

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

[18] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *PVLDB*, 2011.

[19] M. Kalyuzhnaya et al. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 2011.

[20] N. Khoussainova, M. Balazinska, and D. Suciu. PerfXplain: Debugging MapReduce job performance. *PVLDB*, 2012.

[21] C. Li and G. Biswas. Unsupervised learning with mixed numeric and nominal data. *TKDE*, 14(4), 2002.

[22] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool, 2010.

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

[24] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. WebDocs: a real-life huge transactional dataset. `http://fimi.ua.ac.be/data/webdocs.pdf`, 2012.

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

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[27] A. D. Popescu, V. Ercegovac, A. Balmin, M. Branco, and A. Ailamaki. Same queries, different data: Can we predict query performance? In *SMDB*, 2012.

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

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

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

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

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

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

[34] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, 1999.

[35] Apache Hadoop NextGen MapReduce (YARN). `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`, 2013.

[36] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. In *NIPS*, 2007.