ApproxHadoop: Bringing Approximations to MapReduce Frameworks

Íñigo Goiri[†*] Ricardo Bianchini[†‡] Santosh Nagarakatte[‡] Thu D. Nguyen[‡]

[‡]Rutgers University
{ricardob, santosh.nagarakatte, tdnguyen}@cs.rutgers.edu

[†]Microsoft Research
{inigog, ricardob}@microsoft.com

Technical Report DCS-TR-709, Department of Computer Science, Rutgers University
August 2014, Revised December 2014

Abstract

Research has shown that approximate computing is effective at reducing the resource requirements, computation time, and/or energy consumption of large-scale computing. In this paper, we We propose and evaluate a framework for creating and running approximation-enabled MapReduce programs. Specifically, we propose approximation mechanisms that fit naturally into the MapReduce paradigm, including input data sampling, task dropping, and accepting and running a precise and a user-defined approximate version of the MapReduce code. We then show how to leverage statistical theories to compute error bounds for popular classes of MapReduce programs when approximating with input data sampling and/or task dropping. We implement the proposed mechanisms and error bound estimations in a prototype system called ApproxHadoop. Our evaluation uses MapReduce applications from different domains, including data analytics, scientific computing, video encoding, and machine learning. Our results show that ApproxHadoop can significantly reduce application execution time and/or energy consumption when the user is willing to tolerate small errors. For example, ApproxHadoop can reduce runtimes by up to 32× when the user can tolerate an error of 1% with 95% confidence. We conclude that our framework and system can make approximation easily accessible to many application domains using the MapReduce model.

1. Introduction

Motivation. Despite the enormous computing capacity that has become available, large-scale applications such as data analytics and scientific computing continue to exceed available resources. Furthermore, they consume significant amounts of time and energy. Thus, approximate computing has and continues to garner significant attention for reducing the resource requirements, computation time, and/or energy consumption of large-scale computing (*e.g.*, [5, 6, 10, 18, 40]). Many classes of applications are amenable to approximation, including data analytics, machine learning, Monte Carlo computations, and image/audio/video processing [4, 15, 27, 32, 43]. As a concrete example, Web site operators often want to know the popularity of individual Web pages, which can be computed from the access logs of their Web servers. However, relative popularity is often more important than the exact access counts. Thus, estimated access counts are sufficient if the approximation can significantly reduce processing time.

In this paper, we propose and evaluate a framework for creating and running approximation-enabled MapReduce programs. Since its introduction [14], MapReduce has become a popular paradigm for many large-scale applications, including data analytics (*e.g.*, [2, 3]) and compute-intensive applications (*e.g.*, [16, 17]), on server clusters. Thus, embedding a general approximation approach in MapReduce frameworks can make approximation easily accessible to many applications.

MapReduce and approximation mechanisms. A MapReduce job consists of user-provided code executed as a set of map tasks, which run in parallel to process input data and produce intermediate results, and a set of reduce tasks,

[*] This work was done while Íñigo Goiri was at Rutgers University.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694351

which process the intermediate results to produce the final results. We propose three mechanisms to introduce a general approximation approach to MapReduce: (1) *input data sampling*: only a subset of the input data is processed, (2) *task dropping*: only a subset of the tasks are executed, and (3) *user-defined approximation*: the user provides a precise and an approximate version of a task's code. These mechanisms can be easily applied to a wide range of MapReduce applications (Section 5).

Computing error bounds for approximations. Critically, in Section 3, we show how multi-stage sampling theory [29] and extreme value theory [12] can be used to compute error bounds (*i.e.*, confidence intervals) for approximate MapReduce computations. Specifically, we apply the former theory to develop a unified approach for computing error bounds when using input data sampling and/or task dropping in applications that use a set of aggregation (*e.g.*, sum) reduce operations. We apply the latter theory to compute error bounds when using task dropping in applications that use extreme value (*e.g.*, min/max) reduce operations. Such error bounds allow users to intelligently trade accuracy to improve other metrics (*e.g.*, performance and/or energy consumption). This coupling of mechanisms and statistical theories for computing error bounds in the MapReduce framework is one of our main contributions.

ApproxHadoop. We have implemented our proposed mechanisms and error bound estimations in a prototype system called ApproxHadoop, which we describe in Section 4. For MapReduce programs that use ApproxHadoop's error estimation, the user can specify the desired error bounds at a particular confidence level when submitting a job. As the job is executed, ApproxHadoop gathers statistics and determines a mix of task dropping and/or input data sampling to achieve the desired error bounds. Alternatively, the user can explicitly specify the fraction of tasks that can be dropped (*e.g.*, 25% of map tasks) and/or the data sampling ratio (*e.g.*, 10% of data items). In this case, ApproxHadoop computes the error bounds for the specified levels of approximation. This second approach can also be used for running arbitrary ApproxHadoop programs that can tolerate approximations, but for which ApproxHadoop's error-bounding techniques do not apply; of course, ApproxHadoop cannot compute error bounds for such jobs. For user-defined approximations, the user can specify the fraction of tasks that should run the corresponding approximate version of the code.

Evaluation. We use ApproxHadoop to implement and study approximation-enabled applications in several domains, including data analytics, scientific computing, video encoding, and machine learning. In Section 5, we present experimental results for these applications. These results show that ApproxHadoop allows users to trade small amounts of accuracy for significant reductions in execution time and energy consumption. For example, the execution time of an application that counts the popularity of projects (subsets of articles) in Wikipedia from Web server logs for one week (217GB) decreases by 60% when the user can tolerate a maximum error of 1% with 95% confidence. This runtime decrease can be even greater for larger input data (and the same maximum error and confidence); *e.g.*, 32× faster when processing a year of log entries (12.5TB).

Our results also show that, for a wide range of error targets, ApproxHadoop successfully chooses combinations of data sampling and task dropping ratios that achieve close to the best possible savings while *always achieving the target error bounds*. For the same project popularity application, ApproxHadoop reduces the execution time by 79% when given a target maximum error of 5% with 95% confidence.

Contributions. In summary, we make the following contributions: (1) we propose a general set of mechanisms for approximation in MapReduce, (2) we show how statistical theories can be used to compute error bounds in MapReduce for rigorous tradeoffs between approximation accuracy and other metrics, (3) we implement our approach in the ApproxHadoop prototype, and (4) via extensive experimentation with real systems, we show that ApproxHadoop is widely applicable to many MapReduce applications, and that it can significantly reduce execution time and/or energy consumption when users can tolerate small amounts of inaccuracy.

2. Background

MapReduce. MapReduce is a computing model designed for processing large data sets on server clusters [14]. Each MapReduce computation processes a set of input key/value pairs and produces a set of output key/value pairs. Each MapReduce program defines two functions: `map()` and `reduce()`. The `map()` function takes one input key/value pair and processes it to produce a set of intermediate key/value pairs. The `reduce()` function performs a reduction computation on all the values associated with a given intermediate key to produce a set of final key/value pairs.

A MapReduce computation is executed in two phases by a MapReduce framework, a Map phase and a Reduce phase. To execute the Map phase, the framework divides the input data into a set of blocks, and runs a *map task* for each block that invokes `map()` for each key/value pair in the block. To execute the Reduce phase, the framework first collects all the values produced for each intermediate key. Then, it partitions the intermediate keys and their associated values among a set of *reduce tasks*. Each reduce task invokes `reduce()` for each of its assigned intermediate key and associated values, and writes the output into an output file.

In practice, it has been observed that MapReduce executions often take the form of *waves* of map and reduce tasks, where most map/reduce tasks require similar amounts of processing time as each other [51]. We leverage this observation later in our implementation of ApproxHadoop.

Hadoop. Hadoop is the best-known, publicly available implementation of MapReduce [1]. Hadoop comprises two

main parts: the Hadoop Distributed File System (HDFS) and the Hadoop MapReduce framework. Input and output data to/from MapReduce programs are stored in HDFS. HDFS splits files across the local disks of the servers in the cluster. A cluster-wide *NameNode* process maintains information about where to find each data block. A *DataNode* process at each server services accesses to data blocks.

The framework is responsible for executing MapReduce jobs. Users submit jobs to the framework using a client interface; the user provides configuration parameters via this interface to guide the splitting of the input data and set the number of map and reduce tasks. Jobs must identify all input data at submission time. The interface submits each job to the *JobTracker*, a cluster-wide process that manages job execution. Each server runs a configurable number of map and reduce tasks concurrently in compute *slots*. The *JobTracker* communicates with the *NameNode* to determine the location of each job's data. It then selects servers to execute the jobs, preferably ones that store the needed data locally if they have slots available. Each server runs a *TaskTracker* process, which initiates and monitors the tasks assigned to it. The *JobTracker* starts a duplicate of any straggler task, *i.e.* a task that is taking substantially longer than its sibling tasks.

3. Approximation with error bounds

We propose three mechanisms for approximation in MapReduce: (1) input data sampling, (2) task dropping, and (3) user-defined approximation [20]. In this section, we use theories from statistics to rigorously estimate error bounds when approximating using input data sampling and/or task dropping.

3.1 Aggregation

Multi-stage sampling theory. We leverage multi-stage sampling [29] to compute error bounds for approximate MapReduce applications that compute aggregations (*e.g.*, counting accesses to Web pages from a log file). The set of supported aggregation functions includes *sum*, *count*, *average*, and *ratio*. For simplicity, we next discuss two-stage sampling. Depending on the computation, it may be necessary to use additional sampling stages as discussed at the end of the section.

Two-stage sampling works as follows. Suppose we have a population of T units, and the population is partitioned into N clusters. Each cluster i contains M_i units so that $T = \sum_{i=1}^{N} M_i$. Suppose further that each unit j in cluster i has an associated value v_{ij}, and we want to compute the sum of these values across the population, *i.e.* $\sum_{i=1}^{N}\sum_{j=1}^{M_i} v_{ij}$. (We describe the approximation of sum in the remainder of the subsection; approximations for the other operations are similar [29].)

To compute an approximate sum, we can create a sample by randomly choosing n clusters, and then randomly choosing m_i units from each chosen cluster i. Two-stage sampling then allows us to estimate the sum from this sample as:

$$\hat{\tau} = \frac{N}{n}\sum_{i=1}^{n}(\frac{M_i}{m_i}\sum_{j=1}^{m_i} v_{ij}) \pm \epsilon \quad (1)$$

where the error bound ϵ is defined as:

$$\epsilon = t_{n-1,1-\alpha/2}\sqrt{\widehat{Var}(\hat{\tau})} \quad (2)$$

$$\widehat{Var}(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n}\sum_{i=1}^{n} M_i(M_i - m_i)\frac{s_i^2}{m_i} \quad (3)$$

where (s_u^2) is the inter-cluster variance (computed using the sum and average of the values associated with units from each cluster in the sample), (s_i^2) is the intra-cluster variance for cluster i, and $t_{n-1,1-\alpha/2}$ is the value of the Student t-distribution with $n-1$ degrees of freedom at the desired confidence $1-\alpha$. Thus, to compute the error bound with 95% confidence, we use the value $t_{n-1,0.975}$ [29].

Applying two-stage sampling to MapReduce. To apply two-stage sampling to MapReduce, we associate population, clusters, and the value associated with each unit in the population to the respective components in MapReduce. As an example, consider a program that counts the occurrence of a word W in a set of Web pages, where the Map phase counts the occurrence of W in each page, and the Reduce phase sums the counts. Suppose that the program is then run on an input data set with T pages (input data items), and the framework partitions the input into N data blocks. In this case, the population corresponds to the T pages, with each page being a unit. Each data block i is a cluster, where M_i is the number of units (pages) in the block. The Map phase would produce $<W, v_{ij}>$ for each unit (page) j in cluster (data block) i, where v_{ij} is the value associated with that unit.

With the above associations, an approximate computation can be performed by executing only a subset of randomly chosen map tasks, using task dropping to avoid the execution of the remaining map tasks. Further, each map task only processes a subset of randomly chosen pages (input data items) from its data block using input data sampling. Together, these actions are equivalent to choosing a sample using two-stage sampling. Thus, in the Reduce phase, Equations 1 and 2 can be used to compute the approximate sum and error bounds.

For jobs that produce multiple intermediate keys, we view the summation of the values associated with each key as a distinct computation. As an example, consider a program that counts the occurrence of every word that appears in a set of Web pages. The Map phase now produces a set of counts for each word appearing in the input pages. Assuming there are z words, Equations 1 and 2 are used in the Reduce phase to produce z approximate sums, each with its own error bound.

Figure 1 illustrates an approximate computation that produces several final keys and their associated sums. In this computation, the sample comprises the 2^{nd} and 4^{th} input data items from block 1, all data items from block 2, the 1^{st}, 3^{rd}, and 5^{th} data items from block 4, and the 2^{nd} data item from block 6. The Map phase produces three intermediate keys k_1, k_2, k_3. Thus, Equations 1 and 2 are used three times to compute $\hat{\tau}_1 \pm \epsilon_1$, $\hat{\tau}_2 \pm \epsilon_2$, and $\hat{\tau}_3 \pm \epsilon_3$.

In some computations, the Map phase may *not* produce a value for every intermediate key from each processed input data item (*e.g.*, in the example in Figure 1, a value was produced for k_1 from the 2nd data item in block 1 but not for the 4th data item.). This means that some units (input data items) do not have associated values for that intermediate key. When this happens, there are two cases: (1) the missing values are 0, or (2) there are no defined values (*i.e.*, part of the Map computation is to filter out these input data items). Our approximation approach works correctly for the first case but not the second. Going back to the second word counting example above, if a page p does not contain a word w (but other pages do), then the Map phase will not produce a count for w from data item p. However, we can correctly view the Map phase as also (implicitly) producing $<w, 0>$ for p. Thus, our approximation with error bounds works for this application.

We cannot handle the second case because the number of data items in a block is no longer the correct unit count for some intermediate keys; the data items that do not have associated values for an intermediate key have to be dropped from the population for the key. It is then impossible to compute accurate unit counts for the block for all intermediate keys without processing all data items in the block. Thus, our use of two-stage sampling depends on the assumption that a value of 0 can be correctly associated with an input data item, if the Map phase did not produce a value for the item for some intermediate key. This is the only assumption that we make in our application of multi-stage sampling.

Finally, we can either adjust the task dropping and input data sampling ratios to achieve a target error bound (*e.g.*, a maximum error of $\pm 1\%$ across all output keys), or compute the error bound for specific dropping/sampling ratios.

Three-stage sampling. In some MapReduce computations, it may be desirable to associate the population units with the intermediate <key, value> pairs produced by the Map phase for each intermediate key, rather than the input data items. For example, we might want to compute the average number of occurrences of a word W in a paragraph, and each input data item is a Web page. In this case, the Map phase would produce $<W, count>$ for each paragraph, so that the average should be computed over the number of pairs produced rather than the number of input pages. We use three-stage sampling to handle such computations. The programmer must understand her application and explicitly add the third sampling level.

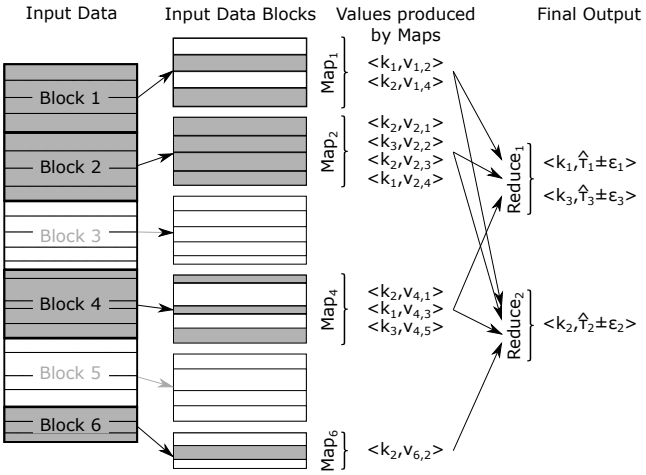


Figure 1. Example usage of two-stage sampling in a MapReduce job. Only 4 map tasks (1, 2, 4, and 6) are executed, processing 10 input data items. $v_{x,y}$ is the value produced for an intermediate key by Map_x from data item y in block x.

Limitation: Missed intermediate keys. With our approach to sampling, it is possible to completely miss the generation of an intermediate key. In the second word counting example above, suppose that a word w only appears in one input Web page. If the sampling skips this page, the computation will not output an estimated count for w. If the set of all words are known *a priori*, we can correctly estimate the count for all words not in the output as 0 plus a bound, with a certain level of confidence. Otherwise, it is impossible to know whether words with non-zero counts were lost in the approximate computation. Thus, our online sampling approach is not appropriate if it is important to discover *all* intermediate keys, including the rarely occurring ones. Nevertheless, we could estimate the overall number of keys (with a certain confidence interval) by extrapolating from a sample, as described in [22]. Further, creating a stratified sample via pre-processing of the input data can help address this limitation.

3.2 Extreme values

Extreme value theory. We leverage extreme value theory [12, 28] to estimate results and compute error bounds for approximate MapReduce applications that compute extreme values (*e.g.*, optimizing for minimum cost). The supported operations include *minimum* and *maximum*.

In extreme value theory, the Fisher-Tippett-Gnedenko theorem states that the cumulative distribution function (CDF) of the minimum/maximum of n independent, identically distributed (*IID*) random variables will converge to the Generalized Extreme Value (GEV) distribution as $n \to \infty$, if it converges. The GEV distribution is parameterized by three parameters μ, σ, and ξ, which define location, scale,

and shape, respectively. This theorem can be used to estimate the min/max in a practical setting, where a finite set of observations is available. Specifically, given a sample of n values, it is possible to estimate a fitting GEV distribution using the Block Minima/Maxima and Maximum Likelihood Estimation (MLE) methods [12, 28].

The fitting process for estimating a minimum when given a sample of n values $v_1, v_2, ..., v_n$ is as follows. (Estimating a maximum is similar.) First, divide the sample into m equal size blocks, and compute the minimum value $v_{i,min}$ for each block i; this Block Minima method transforms the original sample to a sample of minima. Next, compute a fitting GEV distribution G for $v_{1,min}, v_{2,min}, ..., v_{m,min}$ using MLE. This fitting will give values for μ, σ, and ξ for G, as well as the confidence intervals around them (*e.g.*, 95% confidence intervals). These confidence intervals are used to compute G_l and G_h, the fitted GEV distributions that bound the errors around G.

G is then used to estimate the minimum by computing the value min where $G(min) = p$ for some low percentile p (*e.g.*, 1st percentile). The confidence interval around min is defined by $[min_l, min_h]$, where $G_l(min_l) = p$ and $G_h(min_h) = p$.

Applying extreme value theory to MapReduce. We apply the above to approximate MapReduce min/max computations as follows. First, assume that the Map phase produces values (for a specific intermediate key) corresponding to a sample of random variables. (This is our only assumption about the Map computation.) To approximate, we can then simply drop some of the map tasks, leading to a smaller sample being produced. In the Reduce phase, we would use the above GEV fitting approach to estimate the min/max and the confidence interval. Of course, smaller samples lead to larger confidence intervals (error bounds) for the GEV fitting. Thus, similar to multi-stage sampling, the amount of approximation (*i.e.*, the percentage of dropped map tasks) must be adjusted properly to achieve the desired error bounds.

In some computations, each map task may compute multiple values and output only the min/max of these values. Thus, the values already comprise a sample of min/max, allowing them to be used directly without applying the Block Minima/Maxima method. In turn, this increases the sample size for GEV fitting and so may allow many more map tasks to be dropped when targeting a specific error bound.

Figure 2 shows an example approximate computation that uses GEV. Each map task computes a part of an optimization procedure and outputs the minimum value that it finds. (There is only one intermediate key.) The reduce uses the GEV fitting and estimation approach to determine that the desired error bound has been achieved after map tasks 1 and 2 complete. Thus, map task 3 is discarded.

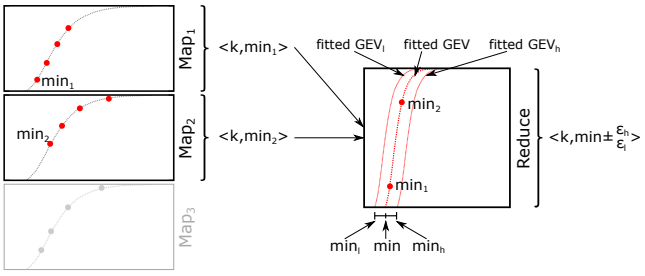


Figure 2. Example usage of GEV theory in a MapReduce job. Each map task computes the values shown (red points) and reports the minimum. Only maps 1 and 2 are executed.

4. ApproxHadoop

We have implemented the three approximation mechanisms [20], along with the error estimation techniques described in Section 3, in a prototype system called ApproxHadoop that extends the Hadoop MapReduce framework. This section describes ApproxHadoop's interfaces for application development and job submission, and the changes required to implement the approximation mechanisms and error estimation. As previously mentioned, we limit our discussion to input data sampling and task dropping.

4.1 Developing ApproxHadoop programs

Predefined classes. adoop offers a set of pre-defined reduce functions (via Java classes) that can be used by programmers (*e.g.*, a reduce function that sums the values for a given intermediate key). ApproxHadoop similarly offers a set of pre-defined approximation-aware map templates and reduce functions—*e.g.*, classes `MultiStageSamplingMapper` and `Multi-StageSamplingReducer` for aggregation and `ApproxMinReducer` for extreme values; see Appendix A for the full list. These classes implement the desired approximation (by leveraging the ApproxHadoop mechanisms), collect the needed information for error estimation (*e.g.*, the total number of data items in a sampled input data block), perform the reduce operation, estimate the final values and the corresponding confidence intervals, and output the approximated results. To perform approximations with dropping/sampling, the user inherits ApproxHadoop's pre-defined classes instead of the regular Hadoop classes. The rest of the program remains the same as in regular Hadoop, including the code for `map()` and `reduce()`.

Using ApproxHadoop, the word count example from [14] would be adapted as in Figure 3. Only lines #2, #8, and #17 are different than the corresponding Hadoop code.

User-defined approximation. The user can provide a precise version and an approximate version of `map()` and/or `reduce()`. For applications that take accuracy information as an input parameter (*e.g.*, the stopping condition for an iterative optimization procedure), the approximate version may simply call the precise code with a more relaxed value for

```
1  class ApproxWordCount:
2    class Mapper extends
         MultiStageSamplingMapper:
3      //key: document name
4      //val: document contents
5      void map(String key, String val):
6        for each word w in val:
7          context.write(w, 1);
8    class Reducer extends
         MultiStageSamplingReducer:
9      //key: a word
10     //vals: a list of counts
11     void reduce(String key, Iterator vals):
12       int result=0;
13       for each v in vals:
14         result+=v;
15       context.write(key, result);
16   void main():
17     setInputFormat(ApproxTextInputFormat);
18     run();
```

Figure 3. An example approximate word count MapReduce program using ApproxHadoop.

this parameter. The user can also provide code for computing error bounds to be run every time a map task completes. For example, in a video application, the user's code may estimate the reduction in quality associated with an approximated set of frames.

4.2 Job submission

Input data sampling and task dropping. hen the user wants to run an approximate program that uses input data sampling and/or map dropping, she can direct the approximation by specifying either:

1. The percentage of map tasks that can be dropped (*dropping ratio*) and/or the percentage of input data that needs to be sampled (*input data sampling ratio*); or,
2. The desired *target error bound* (either as a percentage or an absolute value) at a confidence level.

Users who understand their applications well (*e.g.*, from repeated execution) can use the first approach. In this case, ApproxHadoop will randomly drop the specified percentage of map tasks and/or sample each data block with the specified sampling ratio. For supported reduce operations, ApproxHadoop will also compute and output error bounds.

In the second case, for the supported reduce operations, ApproxHadoop will determine the appropriate sampling and dropping ratios. For other computations, the user would need to provide code for estimating errors. When the Map phase produces multiple intermediate keys, ApproxHadoop assumes that the specified error bound is the maximum error desired for any intermediate key.

User-defined approximation. When running a program with user-defined approximation, the user can specify the percentages of maps and reduces that should be run using the approximate versions. Users can also specify whether ApproxHadoop should start approximate map/reduce tasks to duplicate straggler precise map/reduce tasks. We refer to this as "speculative approximation". **Eventual precision.** Users can also direct ApproxHadoop to save the intermediate results from some of the map tasks, and reuse these results later in a more precise re-execution of the job. This capability allows users to get fast approximate results, which is useful when answers are needed quickly or when computing resources are expensive. Users can then improve accuracy by re-executing jobs more precisely later, without incurring the expense of re-executing map tasks that were already completed with sufficient precision.

4.3 The ApproxHadoop framework

Input data sampling. We implement input data sampling in new classes for input parsing. These classes parse an input data block and return a random sample of the input data items according to a given sampling ratio. For example, we implement `ApproxTextInputFormat`, which is similar to Hadoop's `TextInputFormat`. Like `TextInputFormat`, `ApproxTextInputFormat` parses text files, producing an input data item per line in the file. Instead of returning all lines in the file, however, `ApproxTextInputFormat` returns a sample that is a random subset of appropriate size.

Task dropping. We modified the *JobTracker* to: (1) execute map tasks in a random order to properly observe the requirements of multi-stage sampling, and (2) be able to kill running maps and discard pending ones when they are to be dropped; dropped maps are marked with a new state so that job completion can be detected despite these maps not finishing. We also modified the Reducer classes to detect dropped map tasks and continue without waiting for their results.

User-defined approximation. We extend the default Hadoop Mapper and Reducer classes to accept multiple versions of `map()` and `reduce()`. We also modify the classes to pass parameters to different map/reduce tasks, allowing the classes to be executed with parameters specifying the levels of approximation that the user desires for them. **Error estimation.** As already mentioned, our pre-defined approximate Mapper and Reducer classes implement error estimation. Specifically, the Mapper collects the necessary information, such as the number of data items in the input data block, and forwards it to the Reducer. The Reducer computes the error bounds using the techniques described in Section 3. We have also modified the *JobTracker* to collect error estimates from all reduce tasks, so that it can track error bounds across the entire job. This is necessary for choosing appropriate ratios of map dropping and input data sampling.

Incremental reduce tasks. To estimate errors and guide the selection of sampling/dropping ratios at runtime, reduce tasks must be able to process the intermediate outputs before all the map tasks have finished. We accomplish this by adding a barrier-less extension to Hadoop [47] that allows reduce tasks to process data as it becomes available.

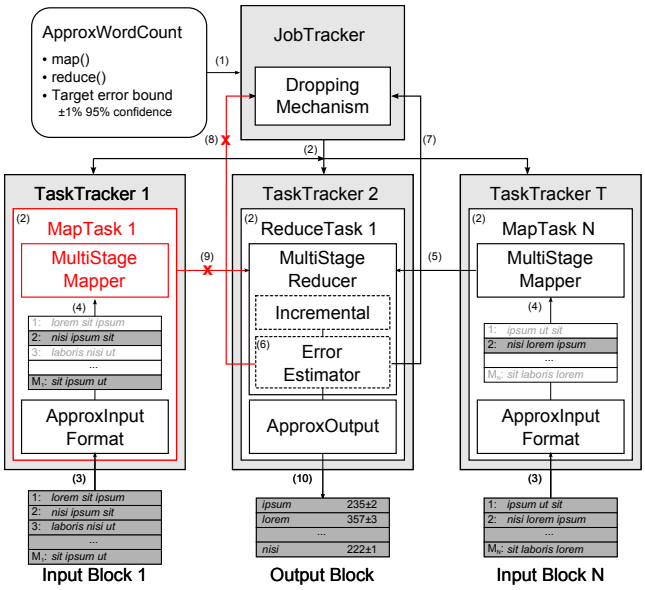


Figure 4. Example execution of the ApproxWordCount program with multi-stage sampling in ApproxHadoop.

Speculative approximation. When the user specifies speculative approximation for a user-defined approximate program, we modified the *JobTracker* to schedule approximate tasks duplicating straggler precise tasks. ApproxHadoop chooses the task that completes first. **Eventual precision.** We implement support for eventual precision by leveraging the Hadoop capabilities to keep the outputs of the map tasks in the *TaskTracker*. We store the old job information using the `JobHistory` infrastructure. We pass this history file to the new job. When we have to start a task which was previously executed, we just create an empty one pointing to the old task. Finally, we modify the reduce mechanism to make it able to fetch data from old tasks. Nectar [21] uses a similar approach where jobs can access the intermediate data from previous jobs.

Example execution. Figure 4 depicts a possible approximate execution of the ApproxWordCount program. The user first submits the job (arrow 1), specifying a target error bound and confidence level. The *JobTracker* then starts N map tasks (only two are shown) and 1 reduce task (2). Each map task executes by sampling its input data block (3 & 4). Map task N finishes first and reports its statistics to the reduce task (5). The reduce task estimates the error bounds (6), decides that the target has been achieved, and so asks the *JobTracker* to terminate the remaining map tasks (7). The *JobTracker* then kills map task 1, which may still be executing (8). The reduce task is informed that map task 1 will not complete (9), allowing it to complete its execution and output the results (10).

4.4 Estimating and bounding errors for aggregation

To compute the error bounds when multi-stage sampling is used, we need to know which cluster (*i.e.*, input data block) each key/value pair came from. Thus, each map task tags each key/value pair that it produces with its unique task ID. Each map task processing an input data block i also tracks the number of units (*i.e.*, data items) M_i in the block, as well as the number of sampled units m_i. This information is passed to all of the reduce tasks. We implement this functionality in our pre-defined Mapper and Reducer classes.

User-specified dropping/sampling ratios. The computation is straightforward when the user specifies the ratios. The framework randomly chooses the appropriate number of map tasks and executes them. Each task is directed to sample the input data block at the user-specified sampling ratio. The reduce tasks then compute the error bounds as discussed in Section 3.1.

User-specified target error bound. Selecting dropping/sampling ratios is more challenging when the user specifies a target error bound, because the ratios required to achieve the bound depend on the variance of the intermediate values. Also, different combinations of n (number of clusters) and m_i/M_i (sampling ratio within each cluster) will achieve similar error bounds.

Our approach for choosing n and m_i is as follows. Suppose that a subset n_1 of map tasks are executed first and the target error is X%. The framework collects information from these tasks to guide the sampling of the remaining map tasks. Specifically, we want to find the minimum amount of time required to complete the job while meeting the constraint:

$$t_{n-1,1-\alpha/2}\sqrt{\widehat{Var}(\hat{\tau})} \leq X\% \times \hat{\tau} \tag{4}$$

Thus, we set up an optimization problem and solve it. The optimization problem is to minimize the remaining execution time $RET = n_2 t_{map}(\bar{M}, \bar{m})$, where n_2 is the number of remaining map tasks, and $t_{map}(\bar{M}, \bar{m})$ is the time required to execute a map task with $\bar{M}$ input data items, processing only $\bar{m}$ items. Note that this assumes that the input data is equally divided among the map tasks, and that processing time for each data item does not vary greatly. We model the running time of a map task as:

$$t_{map}(M, m) = t_0 + Mt_r + mt_p \tag{5}$$

where t_0 is the base time to start a map and t_r and t_p are the average times required to read and process one data item, respectively. We do not model or optimize the running time of reduce tasks, because the Map phase is typically the most time-consuming, and optimizing for the map tasks usually decreases the reduces' runtime.

We also rewrite Equation 3 as:

$$\widehat{Var}(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n}CVar \tag{6}$$

where

$$CVar = n_2\bar{M}(\bar{M}-\bar{m})\frac{\bar{s}^2}{\bar{m}} + \sum_{i=1}^{n_1} M_i(M_i-m_i)\frac{s_i^2}{m_i} \quad (7)$$

Finally, we estimate t_o, t_r, t_p, $\bar{M}$, and $\bar{s}$ using statistics gathered from the execution of the n_1 completed map tasks. We then solve for n_2 and $\bar{m}$ using binary search, under the constraint given by Equation 4, $n = n_1 + n_2$, and $n_2 \leq N - n_1$. The time required to solve the problem is negligible (compared to the time to compute the approximate result and its error bounds), even for large numbers and sizes of clusters.

By default, for a job using multi-stage sampling with a user-specified target error bound, ApproxHadoop executes the first wave of map tasks without sampling. It then uses statistics from this first wave to solve the optimization problem and set the dropping/sampling ratios for the next wave. After the second wave, it selects ratios based on the previous two waves, and so on.

The above approach implies that a job with just one wave of map tasks cannot be approximated. However, the user can set parameters to direct ApproxHadoop to run a small number of maps at a specific sampling ratio in a first pilot wave. Statistics from this pilot wave are then used to select dropping/sampling ratios for the next (full) wave of maps. This reduces the amount of precise execution and allows ApproxHadoop to select dropping/sampling ratios even for jobs whose maps would normally complete in just one wave. Running such a pilot wave may lengthen job execution time (*e.g.*, by running two waves of maps instead of one), but can increase system throughput and decrease energy consumption because fewer maps are executed without sampling.

4.5 Estimating and bounding errors for extreme values

To compute the error bounds when GEV is used, the Reducer needs to know whether the values it receives for each key are already in Block Minima/Maxima format. If not, the Reducer transforms the data to this format.

User-specified dropping ratio. The framework only runs a randomly chosen subset of maps as specified by the user. The reduce then estimates the min/max and corresponding error bounds, as discussed in Section 3.2.

User-specified target error bound. The reduce estimates the min/max and the error bounds as each map completes. The overhead of this estimation is negligible. When the target error bound has been achieved, the reduce asks the *JobTracker* to kill and/or drop all remaining maps. The reduce will then observe that all maps have completed (or been dropped) and will complete the overall computation.

5. Evaluation

We evaluate ApproxHadoop using applications with varying characteristics (see Table 1) and using various approximations for each application. We first explore the approximation mechanisms and errors for some of the applications when users specify the dropping/sampling ratios. Next, we evaluate ApproxHadoop's ability to dynamically adjust its approximations to achieve user-specified target error bounds while minimizing execution time. Third, we study user-defined approximation. inally, we explore ApproxHadoop's sensitivity to the distribution of key values, job sizes (and the impact on energy consumption), and input data sizes.

			Approx.			
Application	**Input data**	**Size**	**S**	**D**	**U**	**Err.**
Page Length	Wikipedia dump	9.8GB	✓	✓		MS
Page Rank		(40GB)	✓	✓		MS
Request Rate	Wikipedia log	46GB (217GB)	✓	✓		MS
Project Popul			✓	✓		MS
Page Popul			✓	✓		MS
Request Rate	Webserver log	330MB (11GB)	✓	✓		MS
Page Popul			✓	✓		MS
Page Traffic			✓	✓		MS
Total Size			✓	✓		MS
Request Size			✓	✓		MS
Clients			✓	✓		MS
Client Browser			✓	✓		MS
Attack Freq			✓	✓		MS
DC Placement	US and Europe	480KB		✓	✓	GEV
Video Encoding	Movie	816MB			✓	U
K-Means	Apache mail list	7.3GB			✓	U

Table 1. List of evaluated applications. Approximations: sample input data (S), drop computation (D), and user-defined approximation (U). Error estimation: multi-stage sampling (MS), generalized extreme values (GEV), and user-defined (U).

5.1 Methodology

Hardware. We run our experiments on a cluster of 10 servers (larger experiments on a 60 server cluster). Each server is a 4-core (with 2 hardware threads each) Xeon machine with 8GB of memory and one 164GB 7200rpm SATA disk. The cluster is interconnected with 1Gb Ethernet. Each server consumes 60 Watts when idle and 150 Watts at peak. Reported energy consumption numbers are based on a power model we built from measuring one server. We use a separate client machine to submit jobs, measuring job execution times on this client.

Software. ApproxHadoop extends Hadoop 1.1.2. Each server hosts a *TaskTracker* and *DataNode*, while one server also hosts the *JobTracker* and *NameNode*. We configure each server to have 8 map slots and 1 reduce slot. Experiments with multiple reduce tasks always run with the number of reduce tasks equal to the cluster size.

Metrics and measurements. Evaluation metrics include job execution time, energy consumption, and accuracy. For accuracy, we report the actual errors between approximate and precise executions, as well as the estimated 95% confidence intervals for the approximate executions. When a job out-

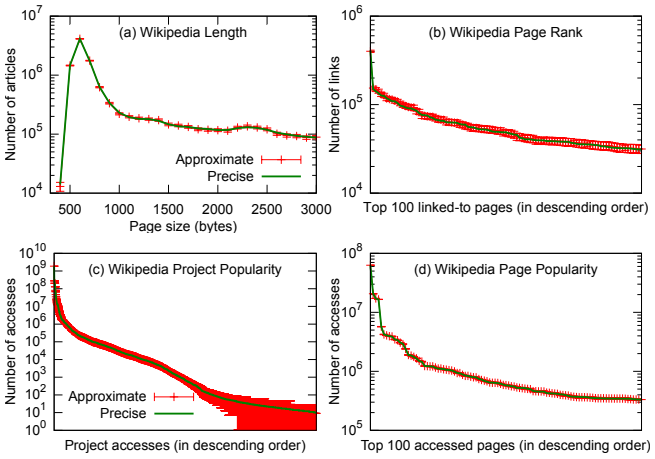


Figure 5. Results of data analysis for (a) WikiLength and (b) WikiPageRank; and log analysis for (c) Project Popularity and (d) Page Popularity. The error bars show the intervals of one approximated run at 1% input data sampling ratio.

puts multiple key-value pairs, we report the actual error and confidence interval for the key with the maximum predicted absolute error. We repeat each experiment 20 times and often report the average, minimum, and maximum for each metric.

5.2 Results for user-specified dropping/sampling ratios

Data Analysis. We study two publicly available large-scale data analysis applications, WikiLength and WikiPageRank, that analyze all English Wikipedia articles. These applications are representative of large-scale processing on collections of Web pages (*e.g.*, for Web search). WikiLength produces a histogram of lengths of the articles [48], with the Map phase producing a key-value pair $<s, 1>$ for each article whose size is in bin s and the Reduce phase summing the count for each key s. WikiPageRank counts the number of articles that point to each article [26], emulating one of the main processing components of PageRank [34]. The Map phase of this application produces a pair $<a, 1>$ for each link that it finds to article a and the Reduce phase sums the count for each key a.

We use the Mapper and Reducer classes that implement multi-stage sampling to create approximation-enabled versions of WikiLength and WikiPageRank. We then use the applications to analyze the May 2014 snapshot of Wikipedia [48, 49]. This snapshot contains more than 14 million articles and is compressed into 9.8GB using bzip2 to allow random access by the map tasks. (Uncompressed, the snapshot requires 40GB of storage.) The 9.8GB partitions into 161 blocks and thus our jobs have 161 maps, running in a little more than two waves in the Xeon cluster.

Figures 5(a) and 5(b) plot parts of the results for a precise execution and an approximate execution with an input data sampling ratio of 1% (each map task processes 1 out of every 100 input data items) for each application. The error bars show the 95% confidence intervals of the approximated values. Taking one size for WikiLength as an example, consider the values for 1000B articles; the precise value is 230,793, while the approximated value is 221,802±9,165. The actual error is 8,991 (*i.e.*, 3.89% = (230,793 - 221,802)/230,793).

Figure 6 shows the impact of approximations on the runtime (Y-axis on right) and estimation errors (Y-axis on left) for WikiLength and WikiPageRank. Each data point along a curve in the graphs plots the average value (over 20 executions) for the corresponding metric, and a range bar limited by the minimum and maximum values (over the 20 executions) of the metric. The gray horizontal band across each graph depicts the range of runtimes from multiple runs of the precise program.

Figure 6(a) shows the impact of different input data sampling ratios when no map tasks were dropped. Observe that runtime can be reduced by 21% (173.6 → 137.5 secs) by processing 1% of the articles, resulting in a 95% confidence interval of 0.81% and an actual error of 0.34%. Figures 6(b)-(c) show the impact of combining task dropping with input data sampling. Observe that dropping tasks reduces execution times more than input data sampling, but with wider confidence intervals. For example, by dropping 50% of the maps, we can reduce the runtime to below 105 secs. However, even without input data sampling (*i.e.*, 100% sampling ratio), the confidence interval at this dropping ratio is 7.38% while the actual error is 2.83%.

Task dropping has a stronger impact on execution times because it eliminates all processing (data block I/O accesses and computation on data items) for each dropped block, whereas input data sampling still requires all data items in a block to be read even though some of them will not be processed as part of the chosen sample. On the other hand, task dropping leads to wider confidence intervals for two reasons: (1) data within blocks usually has "locality" (*e.g.*, the data was produced close in time), and (2) sampling within blocks (input data sampling) adds more randomization than sampling blocks (task dropping), because in our setup, block sizes are substantially larger than the number of blocks ($M \gg N$).

Though not shown in the figures, the approximate version of WikiLength does miss sizes for which counts are small (rare occurrences). For example, in the case of 1% input data sampling, counts were reported for 1028 sizes compared to 5018 in the precise computation. For these missing sizes, the error bound was ±197, which is substantially smaller than the maximum error bound, ±33,480, for the sizes that we did find. This is consistent with our discussion of the limitations of multi-stage sampling in Section 3.1.

To quantify the *overhead* of our implementation, we compare the runtime of the precise version against the approximation with no sampling and no dropping. In this case, the

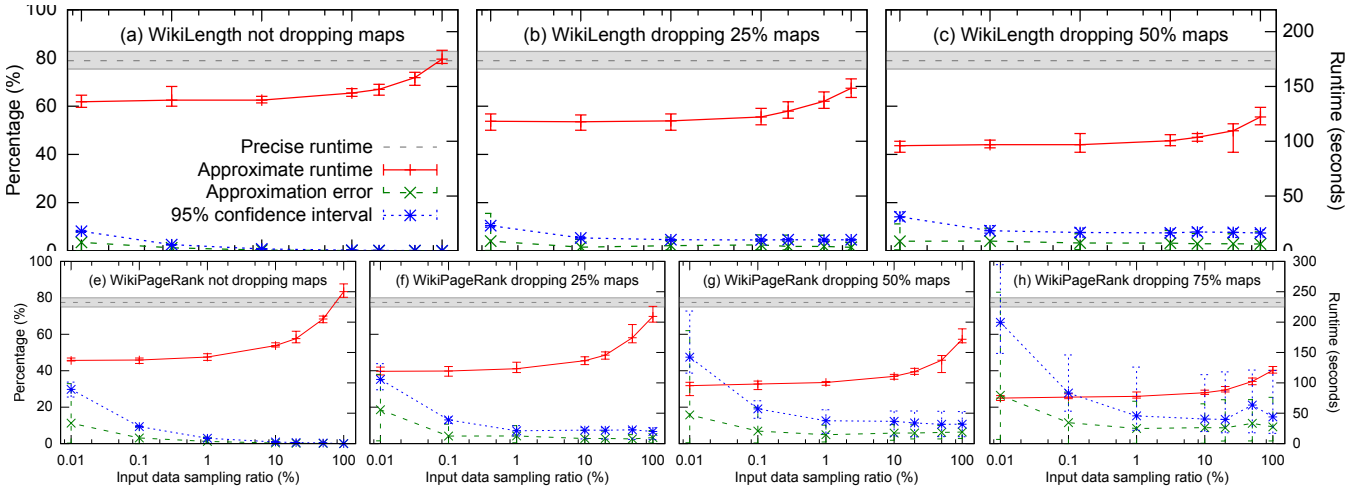


Figure 6. Performance and accuracy of WikiLength and WikiPageRank for different dropping/sampling ratios.

average runtime increases from 173.6 to 175.0 secs, which is less than 1%.

Figure 6(e)-(h) show the results for WikiPageRank. These results show the exact same trends as observed for WikiLength. Thus, in summary, we observe that input data sampling and map dropping can lead to significantly different impacts on job runtime and error bounds. Properly combining the two via multi-stage sampling can lead to the lowest runtime for specific error bound targets (and vice versa).

Log Processing. Log processing is a second important type of data analysis commonly done using MapReduce [8]. Thus, we use ApproxHadoop to process the access log for Wikipedia [49]. This log contains log entries for the first week of 2013 and is compressed into 46.0GB (216.9GB uncompressed).

In this case, each unit is an access log entry containing information like access date, access page, and request size. We compute the Project Popularity (the English project, rooted at `http://en.wikipedia.org`, is the most popular project across more than 2,064 projects with more than 1.9 billion accesses) and Page Popularity (`http://en.wikipedia.org/wiki/Main_Page` is the most accessed page).

Figures 5(c) and 5(d) plot the results for a precise execution and an approximate execution with an input data sampling ratio of 1% for the two programs. Though the confidence intervals for unpopular projects may seem large in Figure 5(c), this effect is caused by reporting small numbers in log scale. These intervals are actually narrower than those for more popular projects. Figure 7 plots the runtime and errors for these applications, as a function of the dropping/sampling ratios. The figure shows very similar trends to those for WikiLength and WikiPageRank. However, the error percentages are significantly lower than those for WikiLength and WikiPageRank. Another interesting observation is that Page Popularity shows significantly higher approximation overhead when not dropping tasks and not sampling the input data. The reason is that Page Popularity requires an inordinate amount of memory (it produces 246,389,084 keys for a total of >11GB) to properly calculate the approximation errors. This causes the ApproxHadoop execution to memory-swap more than the precise execution. The higher swapping also causes greater variability in the execution time. (Note that, when dropping maps, some of the actual errors are larger than the confidence intervals; only 95% of the estimations are expected to fall in the 95% confidence intervals).

Datacenter (DC) Placement. We now explore the impact of approximation on an optimization application. Specifically, the application uses simulated annealing to find the lowest costing placement of a set of datacenters in a geographic area (*e.g.*, the US), constrained by a maximum latency to clients (*e.g.*, 50ms) [19]. The area is divided into a two dimensional grid, with each cell in the grid as a potential location for a datacenter. In the MapReduce program, each map executes an independent search through the solution space, outputting the minimum cost placement that it finds. The single reduce task outputs the overall minimum cost placement.

Note that this application itself is already an approximation: the optimization is not guaranteed to produce an optimal result, but will produce better results as the search becomes more comprehensive and/or a finer grid is used. Thus, user-defined approximations are built in, and are activated by using runtime parameters. We enhance the capabilities for approximations in this application by allowing the dropping of map tasks and the execution of combinations of map tasks with different precision. In both cases, we use the Mapper and Reducer classes implementing extreme value theory to estimate an approximate minimum value and the confidence interval along with the actual minimum cost solution found. The overheads of this approximation are negligible.

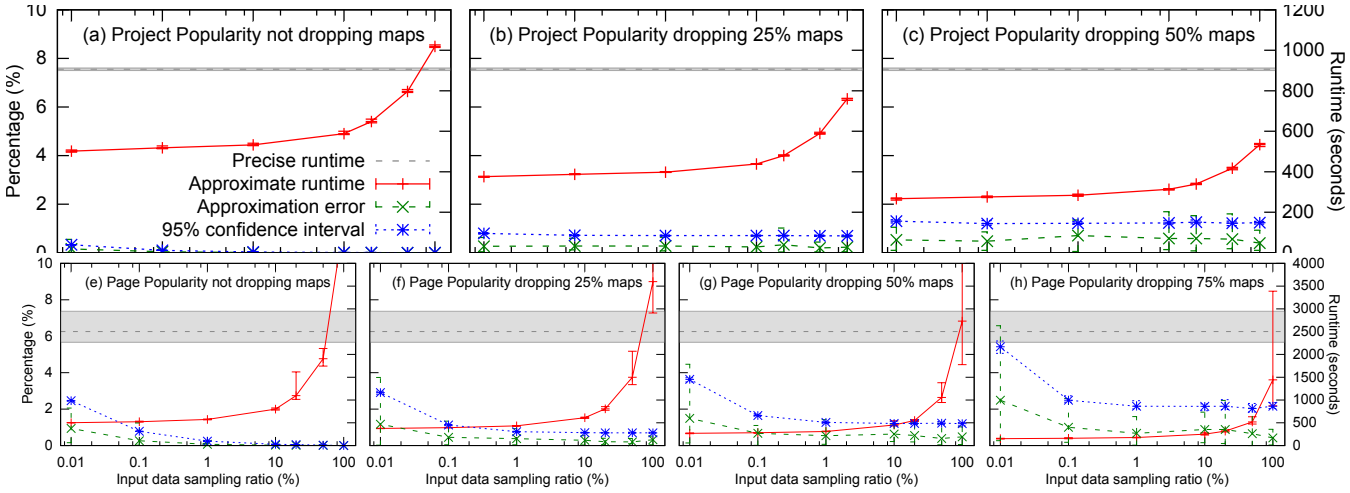


Figure 7. Performance and accuracy of Wikipedia log processing for Project Popularity and Page Popularity.

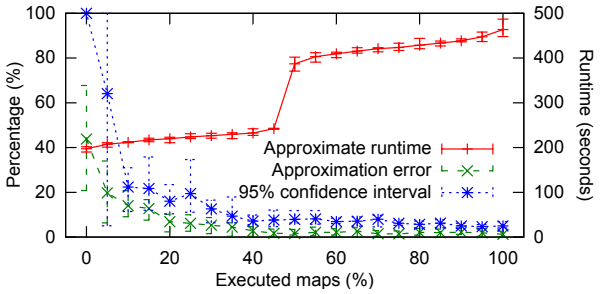


Figure 8. Performance and accuracy of DC Placement for different dropping ratios. The optimization targets a network of datacenters with a 50ms max latency.

Figure 8 shows the impact of map dropping on the optimization with a 50ms maximum latency constraint and the default grid size. We run the experiments with 80 map tasks, with each server configured with only 4 map slots, which is the most efficient setting for this CPU-bound application. Note that the runtime decreases slowly with map dropping (right-to-left) until about 50% of the maps are dropped. This sharp drop in runtime results from the dropping of an entire wave of maps. Error bounds also grow relatively slowly until we drop more than 50% of the maps. Targeting error bounds of at most 10%, we drop 70% of the maps to reduce the runtime by 51%.

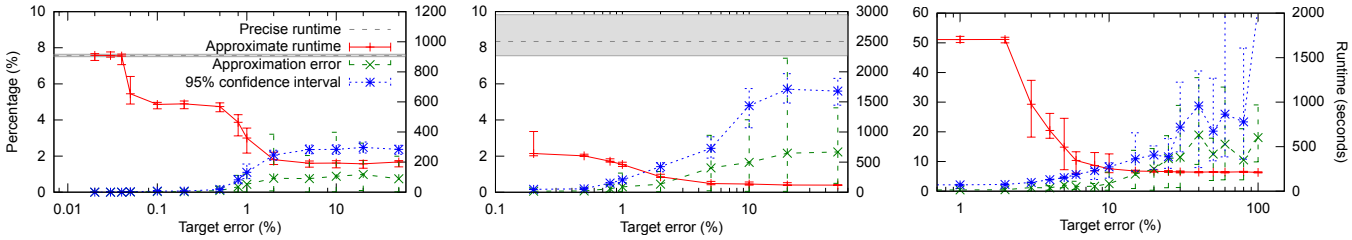
5.3 Results for user-specified target error bounds

We now demonstrate ApproxHadoop's ability to achieve a given target error bound by dynamically adjusting the dropping/sampling ratios. We consider two applications, log processing and optimization, which use multi-stage sampling and extreme value theory for approximation, respectively.

Log Processing. Figure 9(a) plots the runtime and errors as a function of the target error bound for Project Popularity. For small target errors (<0.05%), ApproxHadoop decides that no approximation is possible. Thus, in these cases, there are no errors and the runtimes are the same as for the precise version (the overheads are negligible). From 0.05% to 0.5%, ApproxHadoop is able to use different input data sampling ratios to reduce the runtimes by more than 37%. Above 0.5%, ApproxHadoop can start dropping maps to further reduce the runtimes. For example, for a 1% target error, ApproxHadoop reduces the runtime from 908 to 360 secs (60%). Finally, for target errors above 2%, the required error bound is achieved after the 1st wave of map tasks complete, allowing all remaining maps to be dropped. Thus, ApproxHadoop cannot reduce runtime further, giving a maximum runtime reduction of 79%. Importantly, ApproxHadoop achieved the target error bounds in all experiments, as shown by the 95% confidence interval curve.

Page Popularity presents a more interesting case. Recall from Figure 7 that ApproxHadoop cannot compute Page Popularity for the Wikipedia access log without memory-swapping in our cluster. In fact, our memory capacity is not even large enough to run the first wave of maps precisely without swapping. However, it is possible to compute Page Popularity efficiently for this log by directing ApproxHadoop to use a small pilot wave (see end of Section 4.4).

Figure 9(b) plots the behavior of Page Popularity, when ApproxHadoop relies on a pilot sample ran with a 1% input data sampling ratio. (The pilot takes an average of 93 ± 11 secs to complete for this application.) The results show that we cannot target errors lower than 0.2%. Note that the execution time is somewhat variable at the 0.2% target error, as the reduce tasks memory-swap in some experiments. For larger target bounds, ApproxHadoop can approximate enough that swapping does not occur. Overall, the use of the pilot sample



(a) Project Popularity (b) Page Popularity using a pilot sample (c) DC Placement with 50ms max latency

Figure 9. Performance and accuracy for Project Popularity when targeting different maximum errors.

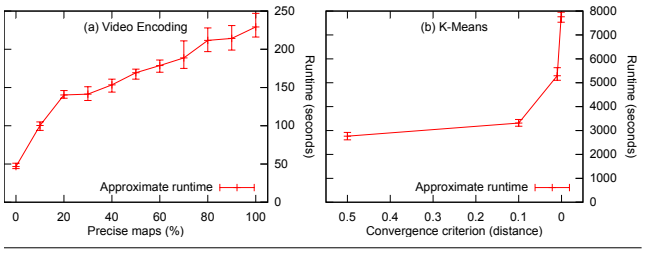


Figure 10. Performance of (a) Video Encoding and (b) K-Means using user-defined approximation.

improves performance significantly. For example, assuming a target bound of 1%, the pilot sample reduces the execution time by 78% compared to the precise execution, and 80% compared to ApproxHadoop without the pilot sample. (We include the time to perform the pilot in these calculations.) Finally, note again that all confidence intervals are smaller than the target error bounds.

DC Placement. Figure 9(c) plots the runtime and errors, as a function of the target error bounds, for an execution with 320 map tasks. Since this application uses only task dropping and the errors introduced by dropping maps are relatively large, ApproxHadoop cannot improve execution time until the target bound is larger than 2%. (Note that we are showing errors for these cases, even though ApproxHadoop is not performing any approximation, because the application itself is an approximation.) At 6%, ApproxHadoop reduces the runtime by more than 80% by dropping 285 maps. For targets higher than 6%, ApproxHadoop achieves the target error after processing the first wave of maps, and so it drops all remaining maps. Thus, ApproxHadoop can reduce runtime by up to 87%, while consistently achieving the target bound.

5.4 Results for user-defined approximations

Video Encoding. We now explore the impact of approximation on a video encoding application. For this application, approximation enables users to play a video as it is getting encoded, using eventual precision.

The original application splits the input video file into chunks (*e.g.*, 64MB), each of which is then encoded by a map task. A single reduce task collects the outputs from the maps, and concatenates them into the final output. The application uses the x264 codec to encode the original video file into different versions, as is done in video-sharing Web sites like YouTube. x264 can be parameterized to encode with different quality levels.

We modify the original application to specify two versions of map: one precise version (`map()`) that encodes its input at high definition (1080p), and a second approximate version (`mapApprox()`) that encodes it at lower quality (240p) but much faster. The user specifies what percentage of the video has to be encoded in high definition. The application encodes the first part (the right percentage) of the video in high definition, and the latter part in lower quality.

In our experiments, the application is used to encode the publicly available high definition version of the movie Elephants Dream [46]. This 816MB video file is divided into 67 blocks of 12.2MB. Figure 10(a) plots the encoding runtime, as a function of the amount of approximation. The figure shows that the performance scales almost linearly with the amount of approximation.

K-Means. Finally, we explore an application that is representative of using MapReduce in machine learning [11]. Specifically, we study K-Means clustering, which partitions a set of data items into k clusters based on the distance between them. We use the MapReduce implementation from Mahout [2]. This implementation reads the list of centroids and starts clustering the different values, according to the distance to the centroid. It then repeats this process iteratively with different centroids, until it meets a convergence criterion (minimum amount of cluster change before the application terminates) that is provided by the user.

Our experiments use the application to process the Apache mail list in 2011, which occupies a total of 7.3GB. The application is a workflow of MapReduce jobs that cluster the emails based on content similarity, rather than project. The workflow runs 1 phase to get the data (1 map), 3 phases to vectorize it (37 maps), 6 phases to merge these vectors (6 maps), N iterations to perform the clustering (6 maps), and 1 final phase to gather the data (6 maps).

Like DC Placement, this application already includes user-defined approximation that is directed by input param-

eters. Thus, in our experiments, we select different convergence factors and observe the impact on performance. Figure 10(b) shows the runtime of the workflow, as a function of the convergence parameter. By increasing the convergence criterion to 0.1, we are able to reduce the runtime from 7761 to 3312 secs (>57%).

5.5 Sensitivity analysis

Impact of the distribution of key values. An important factor when estimating errors is the distribution of the program outputs (key values). We now evaluate the impact of this distribution by considering Log Processing on a log with different characteristics, our department's Web server access log. The log spans 80 weeks from November 2012 to June 2014 and contains more than 40 million requests. The compressed size of the log is 330MB, while the original size is 11GB. The data is divided into 80 files (one per week), each of which fits in a single data block (*i.e.*, <64MB). Each unit in the files is an access (a line), which contains information like timestamp, accessed page, and page size.

We study applications that analyze the log for several typical statistics measures (Table 1): (1) *Request Rate:* computes the average number of requests per time unit (*e.g.*, each hour within a week); (2) *Page Popularity:* computes the number of requests for each page; (3) *Page Traffic:* computes the total amount of network traffic used to serve each page; (4) *Total Size:* computes all the traffic seen by the server; (5) *Request Size:* computes a histogram of the number of requests vs. reply (page) size; (6) *Clients:* computes the total number of requests originating from each client; (7) *Client Browser:* computes the most used browser by the clients; and (8) *Attack Frequencies:* computes the number of attacks (for a set of well-known attack patterns) on the Web server per client.

Figure 11(a) plots a precise and an approximate execution of Request Rate with an input data sampling ratio of 1%. The figure shows the pattern of request rates that one would expect for a Web site, as well as small errors and narrow confidence intervals. For the same execution, Figure 11(b) plots the request rates in descending order. This figure shows that request rates are fairly stable (they vary by roughly 33%), *i.e.* quite a different distribution than what we see in Figure 5. Despite this significantly different distribution, Figure 12(a) shows that the impact of varying the input data sampling ratio is very similar to Figures 6 and 7. (We only consider input data sampling in this experiment, because the job only executes one wave of maps, meaning that dropping maps does not lead to significant reductions in runtime.)

Page Popularity, Page Traffic, Total Size, Request Size, Clients, and Clients Browser all behave akin to Request Rate, with average (over 20 runs) execution time reductions of 25%, 33%, 25%, 32%, 25%, 27%, respectively, when targeting a maximum error of ±1% with 95% confidence. As these applications execute a single wave of maps, we used a pilot sample with 1% sampling ratio.

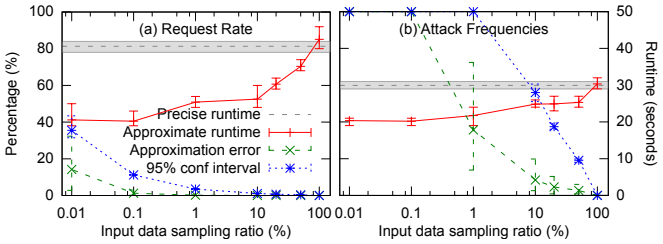


Figure 12. Performance and accuracy of Web server log processing: (a) Request Rate and (b) Attack Frequencies.

More interestingly, consider the precise and approximate results for Attack Frequencies in Figure 11(c), again with 1% input data sampling ratio. Since this application computes rare values, we see larger errors and wider confidence intervals. Figure 12(b) shows that ApproxHadoop can reduce execution time significantly at the cost of higher errors and wider intervals. This application emphasizes the fact that approximation is most effective for estimating values computed from a large number of input data items, rather than those from only a few.

Impact of job size on energy consumption. One of the goals of approximate computing is to conserve energy. By reducing the execution time, as in the experiments so far, and not increasing the power consumption, we save energy. The energy savings is roughly proportional to the reduction in execution time. However, in certain scenarios, ApproxHadoop can also save energy independently of reductions in execution time.

To see this, consider the experiments processing our department's Web server log. They show that input data sampling reduces the runtime (Figure 12), but dropping maps has no significant effect when jobs have a single wave of maps. (Recall that when the user specifies the dropping/sampling rates, they can be applied in the first wave of maps.) The same may occur for applications that have a few more waves as well. For applications where dropping maps does not reduce the runtime, we can transition the servers that have no maps to execute (corresponding to the maps that were dropped) to a low-power state (*i.e.*, ACPI S3) when they become idle. Figure 13 shows the energy consumptions resulting from both dropping maps and input data sampling for Request Rate and Attack Frequencies. As we would expect, decreasing the amount of input data ApproxHadoop samples reduces energy consumption, as this reduces execution time. However, dropping more maps also saves energy, even though this does not reduce execution time.

Impact of data size. Finally, we evaluate the impact of the dataset size on ApproxHadoop by running Log Processing on a larger Wikipedia access log. The log has one file per day of the year for a total of 365 files and 12.5TB, which become 2.3TB when compressed. We experiment with different subsets of the log, as we list in Table 2. Our Log Pro-

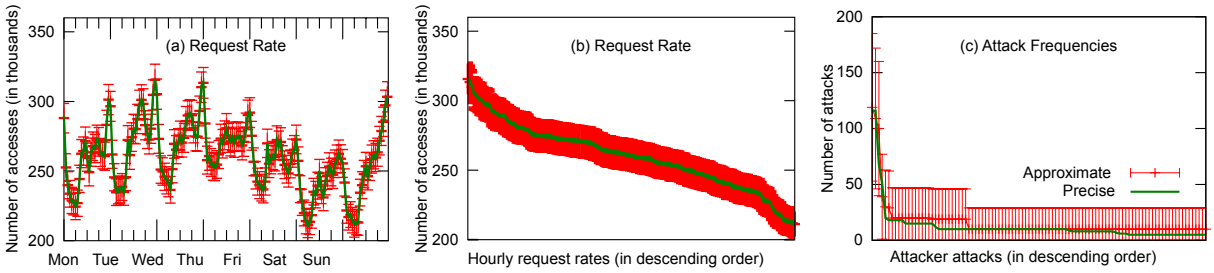


Figure 11. Results of processing our Web server log.

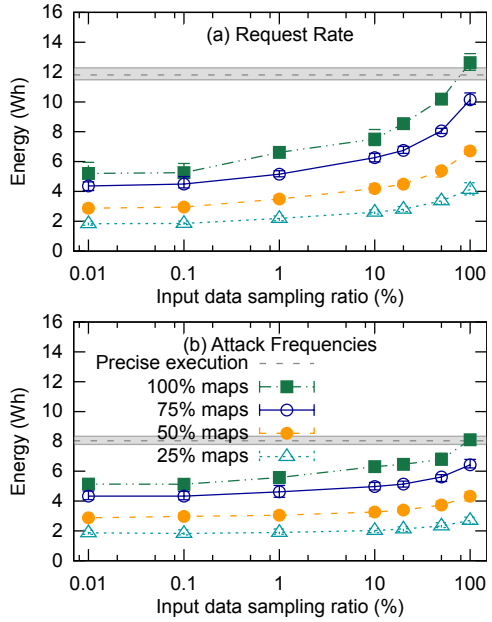


Figure 13. Energy in processing our Web server log using (a) Request Rate and (d) Attack Frequencies for multiple dropping/sampling ratios.

Period	Accesses	Compress	Uncompress	#Maps
1 day	499M	5.7 GB	27.0 GB	92
2 days	1.1G	12.4 GB	58.7 GB	201
5 days	2.8G	32.1 GB	151.7 GB	518
1 week	4.0G	46.0 GB	216.9 GB	740
10 days	5.9G	67.5 GB	317.9 GB	1086
15 days	9.0G	103.2 GB	484.9 GB	1661
1 month	19.4G	221.8 GB	1.0 TB	3567
3 months	55.8G	633.1 GB	2.9 TB	10172
6 months	109.2G	1.2 TB	5.7 TB	19947
1 year	234.2G	2.3 TB	12.5 TB	38246

Table 2. Sizes of the Wikipedia access log [49] for different periods starting on January 1st 2013.

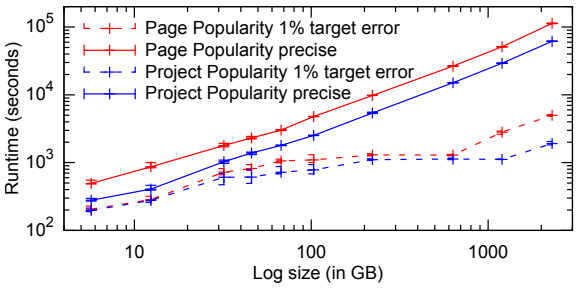


Figure 14. Performance of Page and Project Popularity for different log sizes. Both axes are in log scale.

cessing experiments on Wikipedia so far have only used the first week's data.

Because our Xeon cluster does not have enough disk space, for these experiments, we use another cluster with 60 nodes and a total disk capacity of 14TB. Each server in this cluster is a 2-core (with 2 hardware threads each) Atom machine with 4GB of memory, one 250GB 7200rpm SATA disk (200GB for data), interconnected with 1Gb Ethernet. We configure each Atom server with 4 map slots and 1 reduce slot.

Figure 14 compares the runtime of the precise and approximate (targeting a 1% maximum error with 95% confidence) executions of Project and Page Popularity. As one would expect for these applications, the figure shows that the runtime of the precise program scales linearly with the input size. More importantly, approximating the results shortens runtimes significantly, *especially for the larger input sizes.* For example, the approximate executions of Project and Page Popularity for the year are more than 32× and 20× faster, respectively, while always achieving error bounds lower than 1%.

6. Related work

Researchers have explored numerous approximation techniques at the language [6, 40] and hardware [41] levels, for query processing [4, 10, 44], and for distributed systems [13, 23, 35]. They have also explored techniques for probabilistic reasoning about approximate programs [7, 9, 25, 30, 36, 42, 50]. We discuss the most closely related works on approximation in this section.

Approximation mechanisms. Slauson and Wan [45] have studied map task dropping in Hadoop. Riondato *et al.* [38] have used input data sampling with error estimation in a spe-

cific MapReduce application. However, these works have not considered the two mechanisms together, nor did they consider a general framework for error estimation in the presence of both mechanisms. Rinard [37] has also studied task dropping, albeit in an entirely different approach that requires previous executions with known output and significant pre-computation for estimating errors.

Our proposed sampling/dropping mechanisms are also related to loop/code perforation techniques in SpeedPress for sequential code [24, 31–33, 43]. Green [6] similarly provides a framework to adapt the approximation level based on the defined quality of service for sequential code. Bornholt *et al.* [9] propose uncertain data types to abstract approximations at the language level (*e.g.*, C#) and estimate errors from approximate data sources (*e.g.*, GPS). Unlike these systems for sequential code, ApproxHadoop proposes mechanisms for distributed MapReduce programs.

Paraprox [39] creates approximate data-parallel kernels using function memoization and runtime tuning. The execution is checked a posteriori using an easily checkable quality metric. In contrast, ApproxHadoop provides statistical error bounds around the computed approximate results within a distributed MapReduce framework.

Approximations in query processing. In the domain of database query processing, Garofalakis *et al.* [18] use wavelet-coefficient synopses and Chaudhuri *et al.* [10] use stratified sampling to build samples to provide bounded errors. SciBorq [44] also builds samples based on past query results. BlinkDB [4] uses stratified sampling to produce data samples with the desired characteristics before the execution of the queries. In contrast, we focus on the MapReduce paradigm, and use multi-stage sampling (stratified sampling is a subset) to compute approximations online without pre-computation. Online sampling is a powerful tool for MapReduce environments, where large data sets (*e.g.*, logs) are often processed only a few times (or even just once).

MapReduce and Hadoop. MapReduce Online [13] modifies Hadoop to avoid the barrier between the Map and Reduce phases, and outputs intermediate (called "approximate") results during the execution of the reduce tasks. However, unlike ApproxHadoop, it does not attempt to compute error bounds. GRASS [5] uses speculation to reduce the impact of straggler tasks in jobs that need not complete all their tasks. In contrast, ApproxHadoop starts approximate duplicates for precise stragglers, while providing the ability to bound errors.

7. Conclusions

In this paper, we proposed a general set of mechanisms for approximation in MapReduce, and demonstrated how to use existing statistical theories to compute error bounds (95% confidence intervals) for popular classes of approximate MapReduce programs. We also implemented and evaluated ApproxHadoop, an extension of the Hadoop data-processing framework that supports our rigorous approximations. Using extensive experimentation with real systems and applications, we show that ApproxHadoop is widely applicable, and that it can significantly reduce execution time and/or energy consumption when users can tolerate small amounts of inaccuracy. Based on our experience and results, we conclude that our framework and system can make efficient and controlled approximation easily accessible to MapReduce programmers.

Acknowledgments

We thank A. Verma and his co-authors for the barrier-less extension to Hadoop [47]. We also thank David Carrera and the ASPLOS reviewers for comments that helped us improve this paper. This work was partially supported by NSF grant CSR-1117368 and the Rutgers Green Computing Initiative.

References

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

[2] Apache Mahout. `http://mahout.apache.org`.

[3] Apache Nutch. `http://nutch.apache.org`.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2013.

[5] G. Ananthanarayanan, M. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[6] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[7] S. Bhat, J. Borgström, A. D. Gordon, and C. Russo. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[9] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[10] S. Chaudhuri, G. Das, and V. Narasayya. Optimized Stratified Sampling for Approximate Query Processing. *ACM Transactions on Database Systems (TODS)*, 32(2), 2007.

[11] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning

on Multicore. *Advances in Neural Information Processing Systems*, 19, 2007.

[12] S. Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer, 2001.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[15] A. Doucet, S. Godsill, and C. Andrieu. On Sequential Monte Carlo Sampling Methods for Bayesian Filtering. *Statistics and Computing*, 10(3), 2000.

[16] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the IEEE International Conference on e-Science (e-Science)*, 2008.

[17] Z. Fadika, E. Dede, M. Govindaraju, and L. Ramakrishnan. Adapting MapReduce for HPC environments. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2011.

[18] M. N. Garofalakis and P. B. Gibbons. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[19] I. Goiri, K. Le, J. Guitart, J. Torres, and R. Bianchini. Intelligent Placement of Datacenters for Internet Services. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[20] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. Technical Report DCS-TR-709, Department of Computer Science, Rutgers University, 2014.

[21] P. K. Gunda, L. Ravindranath, C. A. ThekkathA, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[22] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1995.

[23] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.

[24] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[25] O. Kiselyov and C.-C. Shan. Embedded Probabilistic Programming. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages (DSL)*, 2009.

[26] J. Lin. Cloud9: A Hadoop Toolkit for Working with Big Data. `http://lintool.github.io/Cloud9`.

[27] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. *Proceedings of the IEEE*, 82(1), 1994.

[28] S. Liu and W. Q. Meeker. Statistical Methods for Estimating the Minimum Thickness Along a Pipeline. *Technometrics*, 2014.

[29] S. Lohr. *Sampling: Design and Analysis*. Cengage Learning, 2009.

[30] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6. Microsoft Research Cambridge, 2014. `http://research.microsoft.com/infernet`.

[31] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of Service Profiling. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[32] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically Accurate Program Transformations. In *Proceedings of the International Static Analysis Symposium (SAS)*, 2011.

[33] S. Misailovic, S. Sidiroglou, H. Hoffmann, M. Carbin, A. Agarwal, and M. Rinard. Code Perforation: Automatically and Dynamically Trading Accuracy for Performance and Power, 2014. `http://groups.csail.mit.edu/cag/codeperf/`.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[35] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11), 2011.

[36] A. Pfeffer. A General Importance Sampling Algorithm for Probabilistic Programs. Technical Report TR-12-07, Harvard University, 2007.

[37] M. Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the Annual International Conference on Supercomputing (ICS)*, 2006.

[38] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal. PARMA: A Parallel Randomized Algorithm for Approximate Association Rules Mining in MapReduce. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2012.

[39] M. Samadi, J. Lee, A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-Tuning Approximation for Graphics Engines. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[40] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[41] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate Storage in Solid-State Memories. In *Proceedings of*

the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[42] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and Verifying Probabilistic Assertions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[43] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[44] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2011.

[45] J. Slauson and Q. Wan. Approximate Hadoop, 2012. `http://www.joshslauson.com/pdf/cs736_project.pdf`.

[46] O. O. M. Team. Elephants Dream. `http://www.elephantsdream.org`.

[47] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell. Breaking the MapReduce Stage Barrier. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, 2010.

[48] Wikipedia. Wikipedia Database, 2014. `http://en.wikipedia.org/wiki/Wikipedia_database`.

[49] Wikipedia. Wikimedia Downloads, 2014. `http://dumps.wikimedia.org`.

[50] D. Wingate, A. Stuhlmueller, and N. D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

A. Appendix

Table 3 shows the full set of parameters, pre-defined classes, and methods that ApproxHadoop offers to the programmers.

Name	Type	Functionality	Description
`output.quality.confidence`	P	Error estimation	Confidence (*e.g.*, 95% confidence)
`output.quality.abserror`	P	Error estimation	Maximum absolute error (*e.g.*, ± 10)
`output.quality.relerror`	P	Error estimation	Maximum relative error (*e.g.*, $\pm 2\%$)
`getError()`	M	Error estimation	Function to estimate the error from an output
`getConfidence()`	M	Error estimation	Function to estimate the confidence from an output
`ApproxType`	C	Output	Type with error bound (*e.g.*, `ApproxInteger` as 9 ± 1)
`ApproxInputFormat`	C	Sample input data	Sampling format (*e.g.*, `ApproxTextInputFormat`)
`ApproxTextInputFormat`	C	Sample input data	Sampling lines from a text file
`ApproxXMLInputFormat`	C	Sample input data	Sampling elements from a XML file
`mapred.input.approx.skip`	P	Sample input data	How many values to skip/sample (*e.g.*, 10%)
`mapred.input.approx.max`	P	Sample input data	Number of values to process (*e.g.*, 10 lines)
`mapred.input.clustering`	P	Sample input data	Use clustered input (*e.g.*, true)
`mapred.reduce.approx.drop.maps`	P	Drop computation	Number of maps a reduce can skip (*e.g.*, 5 maps)
`mapred.map.approx.drop.percent`	P	Drop computation	Completed maps to start dropping (*e.g.*, 75%)
`mapred.map.approx.drop.extratime`	P	Drop computation	Time to wait to start dropping tasks (*e.g.*, 10 seconds)
`mapred.map.approx.drop.inipercent`	P	Drop computation	Maps to drop at submission (*e.g.*, 25%)
`mapred.map.approx`	P	User-defined	Percentage of approximate/precise maps (*e.g.*, 50%)
`mapApprox()`	M	User-defined	Approximate version of a `map()`
`reduceApprox()`	M	User-defined	Approximate version of a `reduce()`
`ApproxTotalReducer`	C	Pre-def classes	Reducer to approximate the summation of a value
`ApproxMinReducer`	C	Pre-def classes	Reducer to approximate the minimum of all values
`ApproxSumReducer`	C	Pre-def classes	Reducer to approximate the total
`MultiStageSamplingMapper`	C	Pre-def classes	Mapper for multi-stage sampling
`MultiStageSamplingReducer`	C	Pre-def classes	Reducer for multi-stage sampling
`mapred.map.tasks.spec.exec`	P	Concurrent approx	Set to speculate tasks
`mapred.map.tasks.spec.exec.approx`	P	Concurrent approx	Set to speculate tasks using approximation
`keep.task.files`	P	Eventual precision	Set to keep the output of tasks between runs
`mapred.job.previous`	P	Eventual precision	Location of job history (*e.g.*, HDFS path)

Table 3. Interface to ApproxHadoop. P = parameters, M = methods, C = classes.