11-5-2015

Stratified Online Sampling for Sound Approximation in MapReduce

Nitin .
Purdue University, nnitin@purdue.edu

Mithuna Thottethodi
School of Electrical and Computer Engineering, *Purdue University*, mithuna@purdue.edu

T.N. Vijaykumar
Purdue University, vijay@purdue.edu

Milind Kulkarni
Electrical and Computer Engineering, *Purdue University*, milind@purdue.edu

Stratified Online Sampling for Sound Approximation in MapReduce

Nitin , Mithuna Thottethodi, T. N. Vijaykumar, and Milind Kulkarni

School of Electrical and Computer Engineering, Purdue University
{nnitin,mithuna,vijay,milind}@ecn.purdue.edu

Abstract

In the era of big data, many applications perform approximate computations to achieve performance improvements by producing less-precise, yet reasonable, results. Hadoop MapReduce is a widely-used big data framework that processes large amounts of data. A recent work, ApproxHadoop, extends Hadoop with a runtime abstraction to produce approximate results with statistical error bounds. ApproxHadoop uses a carefully-designed multistage sampling strategy to guide its approximation without exceeding target error bounds. However, ApproxHadoop suffers from a major limitation: It performs global uniform sampling across the entire key space of input data (modulo the effects of multistage sampling). Such uniform sampling not only oversamples popular keys but also perniciously undersamples rare keys, potentially skipping computations for the latter entirely. We present MaRSOS (MapReduce with Stratified, Online Sampling), to provide approximation with bounded errors across all keys. MaRSOS makes two key contributions to achieve this target: (1) A novel telescoping-based online sampling strategy that performs per-key sampling without missing rare keys; (2) A feedback system that allows efficient collaboration among distributed map tasks to minimize oversampling. Across a range of MapReduce benchmarks, we demonstrate that MaRSOS can deliver performance improvements while (statistically) bounding per-key errors. MaRSOS is guaranteed to never miss rare keys and varies the sampling rate based on key popularity to achieve better per-key errors than a global sampling approach that samples at the same overall rate.

1. Introduction

Many applications enjoy an intriguing mismatch between the requirements of the problem being solved and the results that the program delivers. The problem *specification* may require relatively coarse results (e.g., what is the average temperature in each city in the US on a given day, to the nearest degree) while the program *implementation* may deliver unnecessarily precise results (e.g., averaging hundreds of temperature readings per city to give results to 3 decimal places). By producing overly accurate results, this mismatch between specification and implementation incurs unnecessary overheads in energy and time.

Sampling can address this mismatch: instead of computing precise answers using all of the data in an input data set, a program can instead perform computation over a subset of the data to produce a less-precise, but still reasonable, answer to the problem. So, for example, rather than using all the temperature readings for a given day, a sampling-based program will instead average a subset of the readings to yield a *sample mean* with reasonable accuracy.[1]

Problems where imprecise, or approximate, answers are acceptable are common in decision support settings, and as a result, there are a number of database systems that support *approximate queries* over data sets, allowing programmers to design queries that give results with specified error bars [1, 3, 5]. Interestingly, these kinds of problems are also common in the kinds of aggregation, selection and analysis tasks that are commonly implemented using MapReduce [6]. Exact answers are typically not required, or even expected, when performing analysis tasks such as "which of my customers spend the most money?" These types of problems are immensely amenable to sampling.

ApproxHadoop is an extension of Hadoop that provides a set of reduction operators amenable to sampling [7]. When a program is written using these special reduction operators, ApproxHadoop samples the input data using multistage sampling to compute an approximate version of the desired reduction. By carefully designing the sampling process, ApproxHadoop is able to give global error bounds on approximate MapReduce computations.

Unfortunately, ApproxHadoop suffers from one critical drawback. It samples *globally*—the entire input stream is sampled at the same rate (modulo the effects of multistage sampling). Although it may seem as though this is not necessarily a problem, global sampling of this form is not always a good match for MapReduce computations. While MapReduce problems look like single queries, in reality, they are computing values across an entire *key space*, and the goal of the program is to compute values for each key, rather than across the entire data set. As a result, uniform sampling will *oversample* popular keys, producing more precision than is required, and, more perniciously, will *undersample* rare keys, potentially skipping the computation for those keys entirely. In other words, while ApproxHadoop provides global bounds on error, these bounds encompass unnecessarily low error bounds for popular keys and unacceptably high error bounds for rare keys—and *only apply to the keys that have been observed*. In other words, the situation for keys that are missed entirely is even worse: *ApproxHadoop provides no error bounds—indeed, provides no results at all—for keys that are missed.*

For some applications, this drawback to uniform sampling may be irrelevant. For example, consider an input set that contains data detailing customer transactions, specifying customer ID and transaction value. To find the customers that have con-

[1] To be precise, the sample mean will have an associated 95% confidence interval, and this confidence interval can be made small by using a larger sample size.

ducted the most transactions, it is unnecessary to accurately capture customers that have only conducted one or two transactions. However, for other problems, it is not permissible to miss rare keys, or to have large errors for such keys. For example, to find high-value customers in the same data set—those that have spent the most in aggregate—completely missing rare keys can be disastrous. A customer with one or two expensive transactions can be high value, but *a uniform sampling system like ApproxHadoop is likely to completely miss the correct result*, as the key (customer ID) appears only rarely in the data set. Indeed, the ApproxHadoop authors acknowledge this issue, stating, "[O]ur online sampling approach is not appropriate if it is important to discover all intermediate keys, including the rarely occurring ones" [7]. Section 2.2 discusses more scenarios where it is important to capture rare keys.

For problems where uniform sampling is inappropriate, we need an approximation strategy that tailors its sampling to each key. We must ensure that we do not miss rare keys, while simultaneously aggressively sampling popular keys to minimize computation. This type of sampling, with per key sampling rates, is an instance of *stratified sampling* [10].

The notion of stratified sampling to control per-key error, as opposed to global error, is not a new one. In the realm of DSS (decision support systems), stratified sampling is the standard technique used to ensure that uncommon keys are sufficiently sampled [1, 2, 3, 5]. However, these prior systems typically perform sampling by constructing sampled data sets *offline*, and then running approximate queries over the reduced data. This offline step means that the entire tuple space must be read to construct the samples prior to performing the computation. In MapReduce problems, which are often concerned with large-scale analytics of a data set rather than repeated queries over the same data, the cost of constructing the samples offline cannot be amortized, so these approaches are not appropriate.

Online sampling is difficult, however. Because popular keys must be sampled at low rates while rare keys are sampled frequently, the correct sampling strategy is to ensure that we end up with roughly the same number of samples for each key [1]. The challenge in performing online stratified sampling, then, is that we must sample the tuple space of a MapReduce problem so that each key generates the same number of samples, uniformly chosen from the set of tuples with that key, *even though we do not know how many tuples have a given key*. Moreover, because MapReduce systems have multiple Mappers processing input simultaneously, we must perform this online stratified sampling in a distributed manner. This problem has not been addressed by existing DSS systems.

In this paper we present MaRSOS (MapReduce with Stratified, Online Sampling), a MapReduce system that tackles these challenges. Like ApproxHadoop, it presents a MapReduce abstraction that allows programmers to readily develop approximate MapReduce applications. Unlike ApproxHadoop, it does not use uniform sampling; MaRSOS provides a runtime system that performs *online, distributed, stratified sampling*, ensuring that we achieve a roughly similar number of samples for each key through a novel telescoping sampling algorithm and feedback system.

Contributions

The primary contribution of MaRSOS is that it can solve approximate MapReduce problems *without losing rare keys*, while providing a strong approximation guarantee: *each key* can have statistically-bounded error. To achieve this contribution, MaRSOS employs two key novelties:

- An online, distributed sampling strategy that performs *per-key* uniform sampling to generate a desired number of samples for each key. Thus, rare keys are automatically sampled frequently while popular keys are sampled infrequently.
- A feedback system that efficiently allows distributed Map tasks in a MapReduce system to collaborate to achieve their per-key uniform sampling without wasting effort in oversampling.

Note that neither of these mechanisms are present in DSS systems, which have not tackled the challenge of online, distributed stratified sampling.

Across a range of MapReduce problems, we demonstrate (i) our system is able to deliver performance improvements from sampling while (statistically) bounding per-key errors; (ii) our system never misses rare keys; and (iii) by varying the sampling rate based on key popularity, our system generates better per-key errors than a global sampling approach that samples at the same overall rate.

2. Background

2.1. Hadoop Architecture

The MapReduce programming model focuses on 'maps' that process the input data to generate key/value tuples and 'reduce' tasks that perform a reduction on collections of records that share the same key. In addition to the map and reduce computation, the underlying execution model consists of additional phases of execution such as (pre-map) input partitioning, (optional) combining with each map, (post-map) shuffle, (pre-reduce) sorting, and final (post-reduce) output.

Logically these execution phases may be thought of as stages with barriers between them, although the barriers can be relaxed in practice as we describe later. Initially, the dataset which resides on a distributed filesystem like HDFS [14], is partitioned in to chunks that independent map tasks may process in parallel. Map tasks then process their respective input data chunks and emit intermediate key/value tuples. To ensure that all tuples with the same key are consolidated at the same reduce task, a *shuffle* communication phase partitions the intermediate data based on keys and pulls the relevant intermediate data buckets to the corresponding reduce tasks. Further, to consolidate tuples with the same keys from various map tasks, each reduce task sorts the intermediate data. Finally, the reduce tasks performs the reduction operation on all the tuples with the same key and emits the final output file to the HDFS.

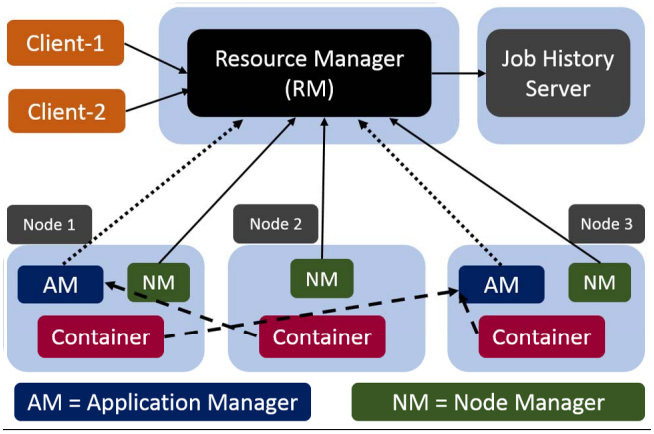

Figure 1: Yarn Resource Manager Schematic

In practice, there are several optimizations to the above execution model. For example, each map task may sorts its intermediate data locally to achieve 'combining' – a partial reduction of its intermediate data. Such combining also facilitates the sorting stage as the problem reduces to one of merging sorted lists. Similarly, shuffle is performed greedily without waiting for all map tasks to conclude to achieve better computation/communication overlap. Finally, even the reduce operation can be initiated in a barrierless fashion to compute partial reductions before all maps are complete.

Hadoop's architecture provides yarn, a cluster resource management layer, that hosts the MapReduce runtime environment. Yarn consists of a centralized *ResourceManager* that co-ordinates with per node *NodeManagers* to provide *Containers* (compute and memory resources) to its hosts. MapReduce runs as an yarn application inside an *ApplicationMaster* (AM) and obtains resources from the *ResourceManager*. Once resources are allocated, the MapReduce AM becomes the sole caretaker of its map and reduce tasks. Figure 1 shows the high level schematic of the yarn framework. While yarn manages efficient utilization of cluster resources, the MapReduce AM is responsible for tracking the progress of its tasks and relaunching them if they fail.

2.2. Uniform vs. Stratified Sampling

The key distinguishing feature of MaRSOS is that it performs stratified sampling to provide per-key statistical guarantees for approximate MapReduce programs. This section outlines the differences between uniform and stratified sampling, and discusses the circumstances in which the latter is required.

Uniform sampling (simple random sampling) in MapReduce is easily characterized. Consider the set of tuples, T, that is generated by applying the map function to the input data. Given a probability p, each tuple in T will appear in the sample, S, with equal probability, p. The reduction is then performed over S, instead of T. Note that ApproxHadoop actually performs *multi-stage* sampling, where T is first divided into clusters, and a subset of those clusters are sampled with equal probability, p [7], so that processing of entire clusters can be skipped (including reading the input itself). The following discussion on the issues with uniform sampling still applies, so we consider simple random sampling hereafter.

The primary drawback of uniform sampling is that every tuple in T has an equal probability of being selected. A tuple t in T is actually a key-value pair, $< k, v >$. If we consider the set of sampled tuples, S, each key $k_1, k_2, ...$ will appear in tuples in S in the same proportion as they do in T, modulo the variability introduced by sampling. Hence, tuples with popular keys will be prevalent in S, while tuples with rare keys will be uncommon, or, thanks to the variability of sampling, simply not present.

The proportionality of the key distribution in the sample set has an obvious drawback when the reduction being performed by the MapReduce task is *per key*, rather than across all the tuples: because the statistical error of most values of interest computed from samples is inversely proportional to the square root of the number of samples [10], the error in any *per key* value computed on the sample set will be low if the key is popular, but potentially quite high if the key is rare. Or, worse, the key might be missed entirely, providing no results at all, let alone reasonable error bounds, for that key.

As a result of the differing error rates for each key, setting the sampling probability p is challenging. If p is chosen to ensure that rare keys have acceptable errors, then p will have to be very high (indeed, if it is unacceptable to miss keys, p will have to be 1). Since the performance and energy wins of sampling are inversely proportional to p, protecting against this worst-case outcome foregoes any benefits of sampling. If, instead, p is chosen to target the error bounds of the popular keys, or an overall error bound across all the tuples, rare keys will necessarily have very high errors.

Stratified sampling addresses this problem by *partitioning* T into per-key sets [10]. Each key k_i has an associated subset of T, T_i, and a probability, p_i. The sampling process generates a separate sample set S_i for each subset of T, uniformly selecting tuples from T_i with probability p_i. Because each key is sampled at its own probability, stratified sampling can avoid oversampling popular keys or undersampling rare keys. So, for example, if a particular subset is small, its sampling probability can be set to 1, while letting larger subsets use lower sampling probabilities to preserve the energy and performance gains. The obvious challenge is how to implement stratified sampling efficiently. How can we set the sampling probability for a given subset, especially since we do not know *a priori* which subsets will be small and which will be large? How can we coordinate numerous, distributed map tasks to perform stratified sampling without introducing excessive communication? These are precisely the questions that the design of MaRSOS answers.

Use cases A natural question to ask is under what circumstances uniform sampling is appropriate, and under what circumstances stratified sampling should be used instead. First, we note that stratified sampling can be made equivalent to uniform sampling by ensuring that each partition of T is sam-

pled at the same rate. But there may still be advantages to uniform sampling. In particular, stratified sampling necessarily requires examining each tuple of T to determine its key. Hence, stratified sampling is not amenable to the multistage techniques of ApproxHadoop, which can drop entire map tasks, avoiding the overhead of reading in input data. Stratified sampling thus offers less potential performance and energy savings than uniform or multistage sampling. However, it is not meaningful to compare the two sampling schemes, as they are functionally different (the former gives per-key error bounds, while the latter does not). Our intention is not to argue that uniform sampling, a la ApproxHadoop, should never be done; rather we identify a set of criteria where MaRSOS is more appropriate, or even necessary.

The problem being solved must involve per-key data. If the problem concerns aggregates across all of T, independent of key, then there is no penalty for under- or over-sampling a particular key, so uniform sampling would be fine. If the problem, on the other hand, involves calculating different values for each key, then it is possible that stratified sampling is required.

The above criterion is not enough, however. *The problem's solution must be sensitive to adding or removing a small number of tuples, relative to the size of T.* If adding a small number of tuples, with keys chosen by an adversary, can significantly change the results, then uniform sampling *is not* appropriate. Note that this criterion encompasses all problems where it is important to capture every key—if an adversary inserts a *new* key, we must guarantee that it is seen. Similarly, if the problem concerns anomaly detection, or is specifically looking for rare keys, then uniform sampling can be easily thrown off. In contrast, if the problem requires finding *popular* keys, then adding a small number of tuples is unlikely to change the results, so uniform sampling may still be sufficient. Essentially, if the problem is sensitive to the behavior of rare keys, then stratified sampling is required.

Some use cases that meet the above two criteria include:

- Identifying high value customers from a database of transactions. As described in the introduction, because high value customers do not necessarily conduct a large number of transactions, we cannot undersample rare keys (customers).
- Identifying when words enter use. This problem can be solved by running word-count-like MapReduce problems on corpora from different years. As it is important to identify when a word arises, rare words cannot be missed.
- Identifying anomalous behaviors in log files. If entries in log files are keyed by the program that generates the message, we cannot afford to miss entries from programs that generate very few messages.

3. Distributed Online Stratified Sampling

The needs of stratified sampling demands that MaRSOS set its sampling probabilities on a per-key basis: popular keys should be sampled infrequently, while rare keys should be sampled frequently. While this seems simple enough in theory, the problem in practice is that it is impossible to know *a priori* which keys will be popular, and which will be rare. Hence, sampling rates cannot be set ahead of time, and must instead be determined online, as the input data is processed.

To understand the issue, consider the single-key case (generalization to multiple keys is straightforward). We can consider the problem of online sampling as one of processing a stream of data items of unknown length. When each item is inspected, it is preserved in the sample with some probability. Our challenge is to design a sampling scheme that meets two criteria:

1. Each item in the data stream has an equal chance of appearing in the sample.
2. If the data stream is short, we want to sample frequently, while if it's long, we want to sample rarely

The first criterion is important, as there is no guarantee that the items in the input stream are not structured in some way, such that oversampling one portion of the stream could bias the results. Consider the scenario of using sampling to compute the average value of the data items when the input stream is sorted (e.g., in a chained MapReduce problem, where one MapReduce task produces sorted data that is fed to a second); oversampling the beginning of the stream will bias the computed average.

The second criterion, that the sampling rate be inversely proportional to the stream length, is easier to reason about from a different angle. Fundamentally, what we require is that the *number of samples* be roughly constant. If we can pick n samples from the input stream, regardless of the stream's length, then the second criterion is immediately satisfied. We make two observations: (i) if the stream has fewer than n items, then we will not perform sampling at all, an acceptable outcome for such a small data set; (ii) in the multi-key case, this is tantamount to saying that we should sample the same number of items for each key, an outcome consistent with practices in DSS database systems [1, 3].

3.1. Reservoir Sampling

One naïve way to generate an n-item sample would be to merely choose the first n items from the stream. But that violates our first sampling criterion, as it does not account for any structure in the input stream. The process of choosing n items from a stream of data while ensuring that each item in the stream appears with equal probability is called *reservoir sampling* [16]. The basic algorithm for reservoir sampling is as follows. For each item i in a stream of N items labeled $0 \ldots N-1$: if $i < n$, place the item in the *reservoir* (sample set). Otherwise, with probability $n/(i+1)$, randomly eject one item from the reservoir, and place item i in the sample set.

Reservoir sampling seems to neatly solve our problem, providing a sampling scheme that meets both criteria. However, reservoir sampling relies on global knowledge of the sampling state; it uses a global enumeration of items to determine sampling probabilities. As a result, it is not at all obvious how to apply reservoir sampling in a distributed setting, where multiple Mappers are charged with sampling their subset of the

Algorithm 1 Telescoping sampling algorithm

```
p ← 1                                   ▷ Probability of taking a sample
R ← {}                                  ▷ Reservoir
n ← Target # samples

Add x to R with probability p
function TakeSample(x)
    if rand() ≤ p then
        R ← R ∪ {x}
    while |R| ≥ 2n do
        R ← R with half of its items removed
        p ← p/2
```

input stream.[2]

One might think that a scheme whereby each Mapper keeps its own local sample set that is periodically reconciled into a global sample set might work. However, note that if each Mapper has a different notion of how many items are in the stream in total, each sample set will have been sampled at a different rate. Reconciling these local sample sets is non-trivial. MaRSOS instead adopts a different, telescoping sampling strategy that uses fixed intervals of sampling probabilities to facilitate the reconciliation of local sample sets.

3.2. Telescoping Sample Sets

MaRSOS uses a telescoping sampling algorithm to construct its samples. This algorithm has two properties: (i) it only samples at fixed probabilities (in particular, 1, .5, .25, etc.), and (ii) it ultimately generates between n and $2n$ samples at a given probability. Note that the imprecision in point (ii) is due to the fixed sampling probabilities in point (i). The amount of oversampling this incurs is slight, and does not lead to much performance or energy overhead. Essentially, the sampling algorithm we devise settles upon the sampling rate p of the form $1/2^l$ that will generate between n and $2n$ samples. The key challenge is to find p in an online manner while sampling an incoming stream of items.

MaRSOS's sampling algorithm, in the non-distributed case, is presented in Algorithm 1 (we discuss how to extend it to a distributed setting next). We call it a "telescoping" algorithm because it successively drops the sampling rate by a factor of 2 while downsampling the existing sampled set to correspond to the new sampling rate.

One way to understand this algorithm is to view it as a variant of reservoir sampling where we delay the phase of reservoir sampling that "kicks out" existing samples from the reservoir. Instead, the reservoir continually fluctuates between n and $2n$ items. When the reservoir fills to $2n$ items at sampling rate p, we discard half the items in the reservoir (essentially, batch removing the items that would have been kicked out had we done so continuously). The reservoir now contains n items, sampled at rate $p/2$. We then sample incoming items at

[2]Note that although several DSS schemes use reservoir sampling to build their offline sampled data sets, they do not do so in a distributed manner [1, 5].

Algorithm 2 Reconciling multiple reservoirs into a single reservoir

```
Input:
    {R_1, ..., R_M}                 ▷ Reservoirs from Mappers 1...M
    {p_1, ..., p_M}                 ▷ Sampling probabilities
    n                               ▷ Target # of samples
Output:
    R_F                             ▷ Combined reservoir
    p_F                             ▷ Final sampling rate
procedure Reconcile
    R_F ← {}
    p_F ← min(p_1, ..., p_M)
    for i ∈ [1, M] do
        while p_i ≠ p_F do
            R_i ← R_i with half of its items removed
            p_i ← p_i/2
        R_F ← R_F ∪ R_i
    while |R_F| ≥ 2n do
        R_F ← R_F with half of its items removed
        p_F ← p_F/2
```

$p/2$, re-filling the reservoir to $2n$ items, and so on. Hence, the algorithm maintains the following invariant:

Claim 1. *If MaRSOS has seen N elements of a given key, with $N > n$, it will have generated between n and $2n$ samples of that key, uniformly sampled at a rate*

$$p = \frac{1}{2^{\lfloor lg(N/n) \rfloor}}$$

Crucially, unlike reservoir sampling, our telescoping sampling algorithm *does not* require knowing how many total items have been seen. Sampling merely requires knowing the current sampling rate, which is always an inverse power of two.

3.3. Distributed Telescoping Sampling

The algorithm presented above is not distributed: it assumes that there is a single actor performing the sampling. In reality, in a MapReduce program, there are multiple Mappers, each of which is independently sampling its portion of the input data set. Hence, we design a variant of this algorithm that allows *multiple* actors to be sampling, each constructing its own reservoir of samples, which are then reconciled into a single reservoir that still satisfies the invariant that the final reservoir have a single, uniform sampling rate.

The distributed algorithm turns out to be a simple extension of Algorithm 1. First, note that each of M Mappers could implement Algorithm 1 completely independently, assembling its reservoir of samples out of the *subset* of the input it sees. We are thus left with a *set* of reservoirs, $R_1, \ldots, R_M$, each with an associated sampling rate $p_1, \ldots, p_M$. These reservoirs can be combined into a single reservoir satisfying the necessary invariant according to Algorithm 2.

The algorithm proceeds as follows: it first downsamples all of the individual reservoirs to the sampling rate of the "largest" reservoir—the reservoir that was assembled from the largest

subset of the input, and hence has the lowest sampling rate. At this point, all of the reservoirs contain samples generated from their respective input subsets at the same sampling rate, and hence can be merged together. If the merged reservoir is larger than the target number of samples, it is downsampled until it contains between n and $2n$ samples, satisfying the invariant.

Note that although Algorithm 2 generates a new reservoir that satisfies the invariant, each Mapper, because it generates its local reservoir independently, may dramatically oversample its input space; a Mapper only drops its sampling rate if it has *locally* seen enough items to do so. Although eventually the final reservoir is downsampled to the appropriate rate, this individual oversampling means that Mappers perform unnecessary work that is irrelevant to the final result, leaving potential performance improvements on the table.

We make two observations: (i) the reconciliation algorithm tolerates individual Mappers sampling too frequently; and (ii) each Mapper's local sampling rate is never *lower* (i.e., sampling less frequently) than the final sampling rate. These observations lead to a more efficient algorithm.

1. During execution, local Mappers can periodically, and independently, reconcile their local reservoir into the global reservoir. This is sound because each reservoir has always uniformly sampled the input data used to generate the reservoir. Reconciling two or more reservoirs using Algorithm 2 yields a new reservoir that has uniformly sampled the input sets that generated its constituent reservoirs. Hence, reconciliation can happen continuously and asynchronously.
2. Moreover, local Mappers can periodically update their local sampling rate based on the global reservoir's current sampling rate *without having seen sufficient data locally*. This is sound because ultimately all the local reservoirs are sampled together. Hence, to satisfy the invariant, all that is necessary is that the initial sampling rate for the unified reservoir be high enough to generate at least n samples. Because the current global reservoir always has at least n samples (or has a sample rate of 1), its sampling rate is always high enough that after reconciliation there will be sufficient samples.

In other words, Mappers can lazily update the global reservoir *and* can lazily update their local sampling rate based on that global reservoir. The following section describes how this algorithm is implemented in MaRSOS.

3.4. Distributed Sampling Implementation

Recall from Section 2.1 that a MapReduce application consists of a *map* phase and a *reduce* phase. The latter is further divided into a *shuffle* phase, where data is fetched from individual Mappers, a *merge* phase, where data from all of the Mappers is merged, and finally the actual reduction phase. However, of the three phases, only the shuffle can occur before all of the Mappers have completed. In order to perform our distributed telescoping sampling algorithm, we need a global reservoir that reconciles the sampling rates observed by individual Mappers. Because each key has its own associated

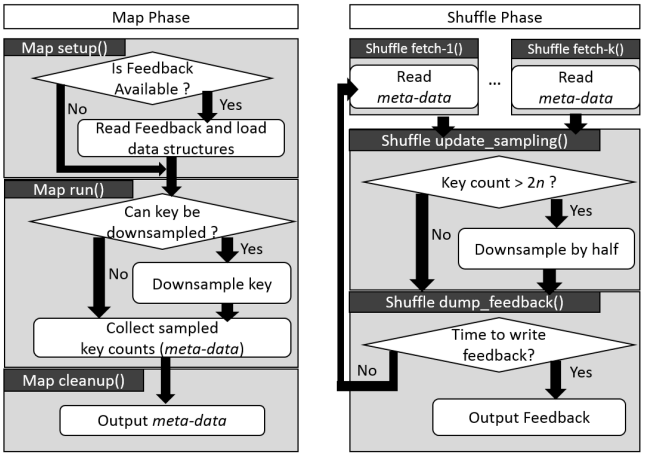

Figure 2: Top-level schematic of Shuffle Feedback

Shuffle phase, which fetches each completed Mapper's output data for that key and thus has a global view, we implement the telescoping sampling algorithm inside the Shuffle.

Figure 2 shows a top level schematic of the feedback mechanism's implementation. The left section of the figure shows the actions that take place at the *Map* end while the right section shows the ones at the *Shuffle* end.

The default action of an individual Mapper is to sample all of its keys. We modify the Mapper function to look for any feedback generated by any Shuffle during the Mapper's setup phase. The Mapper reads the feedback and loads a local hash-table with keys and their corresponding downsampling rates. Next, during the run phase, upon encountering a *key*, the Mapper queries the local hash-table to find the sampling probability p at which the key is to be sampled. We then toss a random biased-coin, with heads probability p, and sample the key if head occurs. If the key is not found in the hash-table, the Mapper sets p for that key to 1 (i.e., the sample is always taken). Additionally, the Mapper maintains a count of sampled keys along with their sampling rates and dumps this metadata during its *cleanup* phase.

After a Mapper finishes its task, the Shuffle periodically fetches each Mapper's output data (note that this is part of the baseline Hadoop architecture). We modify the Shuffle to piggyback on this fetch to gather the metadata as well. The Shuffle maintains a hash-table that maintains the current global sampling rates (p_F in Algorithm 2) for all of the keys the Shuffle is responsible for. The metadata from the Mappers is used to update this hash-table and reconcile the Mappers' data into the Shuffle's global reservoirs for its keys. The contents of this hash-table are then output (via HDFS) so that the Mappers can read this information as their feedback.

All of the above actions that are specific to MaRSOS, can be turned on or off through a simple configuration switch from MapReduce applications.

3.5. Choosing Target Number of Samples

The final implementation decision in MaRSOS is to choose n, the target number of samples for each key. This target should be chosen to balance reducing error (which requires a larger n) and increasing performance (which requires a smaller n). While we do not present a specific method for choosing n in this paper, we elucidate some of the considerations governing its selection. Section 5 presents a series of sensitivity studies that examine the effects of different choices of n.

Impact on performance Obviously, choosing a smaller n means that sampling will be more aggressive. This is true on two fronts: first, it means that more keys will be subject to sampling—we only sample keys for which we have seen at least $2n$ items (*cf.* Algorithm 1). Second, it means that the keys that are sampled will be sampled more aggressively, as we must see fewer items before dropping the sampling rate.

Unlike with uniform sampling, there is not a direct connection between sampling rate and performance, as the amount of sampling that is performed depends on the distribution of rare and popular keys, and is hence highly input dependent. However, in our experience, most problems' inputs are dominated by popular keys, and hence whether relatively unpopular keys are sampled or not does not affect performance much. While it might seem that keeping n low also means that popular keys will get sampled more aggressively, decreasing the sampling rate of popular keys by one step requires *halving* n. Conversely, increasing n will not affect popular keys' sampling rates until it is doubled. Hence, there is not much penalty to choosing a larger n.

Impact on error Error exerts a countervailing force to performance: a larger n results in lower error. However, there are diminishing returns to increasing n. For most sampling statistics of interest, standard error (or relative error) is inversely proportional to $\sqrt{n}$. Hence, when n is small, increasing n rapidly decreases error, but when n is large, it requires large increases in n to materially affect error.

We note that error in many cases is also related to the *variance* of the distribution being sampled—so, for example, the standard error of the mean is proportional to the sample standard deviation. This is clearly an input-dependent property, which makes it difficult to choose an n to target a specific error. ApproxHadoop uses a feedback mechanism to vary the sampling rate based on observed variance [7], and a similar technique could be applied in MaRSOS. However, we note that this mechanism relies on an i.i.d. assumption that may not hold for many inputs.

Overall concerns Putting these two considerations together, we see that there is not much performance impact in choosing a larger n (as popular keys will still be aggressively sampled), but also not much reason to choose an extremely large n (it will have little affect on error rates). In our experience, setting n to 1000 yields a good balance of error and performance across our benchmarks.

4. Experimental Methodology

We evaluate MaRSOS by extending Apache Hadoop 2.6.0. The MapReduce framework (MapReduce 2.0) runs as a YARN application. YARN is a cluster resource management layer build on top of HDFS. It consists of a *ResourceManager* that allocates resources to its applications.

We run our experiments on a 26 node cluster. Each cluster node runs Linux 2.6.32 on a 4-core AMD Opteron (tm) Processor 6320 with 8 GB of memory and 500 GB of SATA disk allocated for the MapReduce system. We configure one of the nodes to be the *NameNode* while another node is chosen to act as the *ResourceManager* and also the *JobHistory* server. The remainder (24) of the nodes act as datanodes and run an instance each of *NodeManager* and *DataNode*. The maximum memory allocated for a map task and a reduce task are set to 1 GB and 2.5 GB respectively.

Our benchmark suite consists of several benchmarks chosen from PUMA [4]. Two of the benchmarks, *husers* and *huseravg*, are straightforwardly extended from PUMA. The benchmarks along with their description are listed in Table 1. The data size used for each of the benchmarks is 100 GB. The number of reducers is equal to that of datanodes.

5. Results

We evaluate MaRSOS in three ways. Section 5.1 evaluates MaRSOS's performance: how much does dropping tuples help? Section 5.2 evaluates MaRSOS's accuracy: how much error does stratified sampling introduce? How does its accuracy compare to uniform sampling (a la ApproxHadoop)? Finally, Section 5.3 studies MaRSOS's sensitivity to different choices of n (the target number of samples per key).

5.1. Performance

Figure 3 shows the speedups of MaRSOS over baseline, unmodified Hadoop for each of our benchmarks. For all benchmarks, we set $n = 1000$. In addition, to understand the overheads introduced by the feedback implementation (Section 3.4), we compare to a variant of MaRSOS that tracks

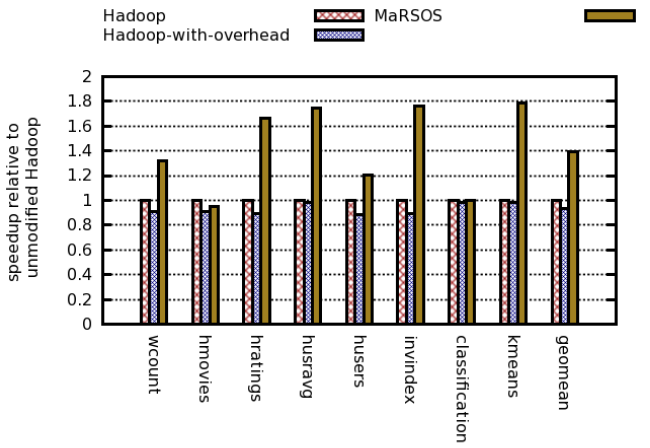

Figure 3: Speedup relative to Hadoop

Table 1: Benchmark Details

Name	Data Source	Description
wcount	Wikipedia	Counts words in documents. Mappers read input documents and emit ⟨word, 1⟩ tuples while reducers accumulate all tuples per word. With approximation, Mappers can immediately drop words after reading them.
hmovies	Netflix	Generates histogram for #movies per rating. Mappers read input movie data in the form of ⟨movie,reviews⟩, where each review is a ⟨userid,rating⟩ tuple. Mappers process input to calculate average rating per movie and emit ⟨rating, 1⟩ tuples. Reducers accumulate all tuples of a rating. With approximation, Mappers can only drop tuples after map processing is complete.
hratings	Netflix	Generates histogram for #reviews per rating. Mappers read input movie data, ⟨movie,reviews⟩, and emit ⟨rating, 1⟩ tuples. Reducers accumulate all tuples of a rating. With approximation, Mappers can drop tuples immediately after reading input data.
huseravg	Netflix	Generates a histogram for average rating per user. Mappers read input movie data, ⟨movie,reviews⟩, and emit ⟨userid, rating⟩ tuples. Reducers perform average rating per userid. With approximation, Mappers can drop tuples immediately after reading ⟨userid,rating⟩ tuple.
husers	Netflix	Generates histogram for #reviews per user. Mappers read input movie data, ⟨movie,reviews⟩, and emit ⟨userid, 1⟩ tuples. Reducers accumulate all tuples of a rating. With approximation, Mappers can drop tuples immediately after reading ⟨userid,rating⟩ tuple.
invindex	Wikipedia	Generates pairs of word-to-document tuples from documents. Mappers read input documents and emit ⟨word, documentid⟩ tuples while reducers accumulate all occurences of unique documentid's per word. With approximation, Mappers can immediately drop words after reading them once.
classification	Netflix	Classifies movies, based on cosine similarity, into a set of pre-determined number of clusters (32) and generates a count per cluster Mappers read input movie data, perform clustering and emit ⟨clusterid,1⟩ tuples. Reducers accumulate all tuples of a clusterid. With approximation, Mappers can only drop tuples following the cluster processing.
kmeans	Netflix	A popular data mining algorithm that classifies movies into a set of k-clusters (32). Mappers read input movie data, and cluster movies into k-clusters and emit ⟨clusterid, (movie,reviews)⟩ tuples. Reducers accumulate all tuples of a clusterid and calculate new centroids. With approximation, Mappers can only drop tuples following the cluster processing. But large shuffle and reduce computations follow.

reservoirs and sends feedback between the Shuffles and the Mappers, but does not perform any sampling (i.e., regardless of the sampling probability, all tuples are output by the Mappers). This variant is labeled hadoop-with-overhead. We note that the overheads of MaRSOS are minimal, with an average speedup reduction of only 7%. This low overhead means that MaRSOS can begin to see performance improvements even with relatively high sampling rates.

Across the entire range of benchmarks, MaRSOS gives 39% speedup, on average, over Hadoop. Several of the benchmarks give speedups over 60%. Two of the benchmarks, *hmovies* and *classification*, show little speedup over Hadoop. This is because MaRSOS achieves its improvements through two mechanisms: (i) once the key for a tuple is determined, sampling can be performed and the map computation can be terminated; (ii) dropped tuples are not output by the Mapper, reducing Shuffle traffic and Reduce computation. For these two benchmarks, MaRSOS can only determine the key for the tuple late in the Map computation, so there is no computation to save in Map. Furthermore, both benchmarks have light Shuffle and Reduce phases, offering little opportunity.

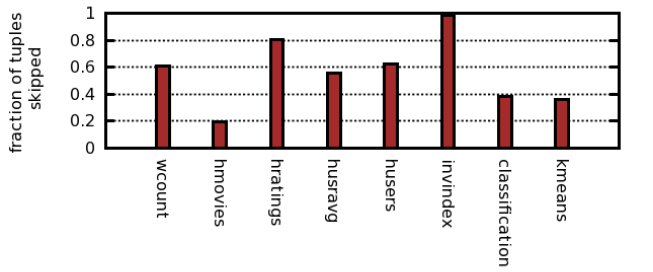

Figure 4: Fraction of tuples skipped by MaRSOS

The difference between *classification* (no speedup) and *kmeans* (72% speedup) is illustrative. In both cases, there is little opportunity in Map: both use the same computation to determine which cluster a tuple belongs to. However, *classification* merely counts the number of tuples per cluster, leading to nominal Shuffle and Reduce phases. In contrast, *kmeans* sends the entire volume of the movie dataset to the Shuffle, so that the Reduce phase can re-compute cluster centroids. As a result, *kmeans* has very heavy Shuffle and Reduce phases, leading to substantial performance gains from sampling.

Distribution of downsampled keys Figure 4 shows the fraction of tuples that are dropped due to sampling in each of our benchmarks. Even though all of our benchmarks use the same target number of samples, the effective overall sam-

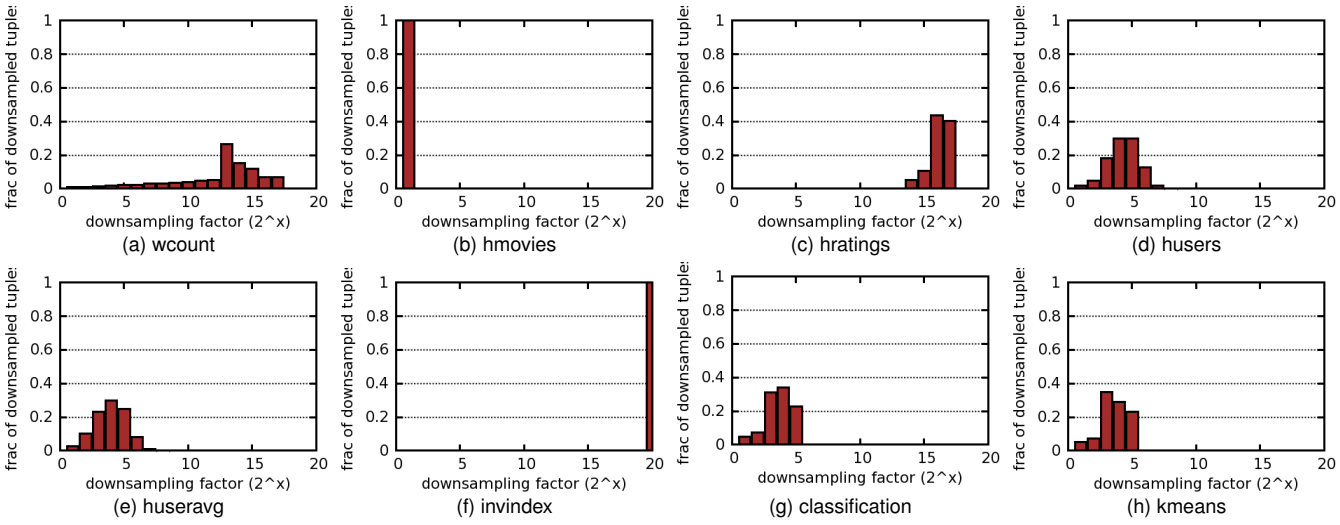

Figure 5: Downsampled tuple distribution for benchmarks

pling rate is dependent on the distribution of keys in the input, since only keys with more than n tuples will be sampled at all, and only keys with substantially more than n tuples will be sampled aggressively. We observe that across the benchmarks, MaRSOS is able to drop a significant fraction of tuples, with many benchmarks dropping more than half of their tuples. The percentage of tuples kept translates straightforwardly into an effective sampling rate; hence many benchmarks have global sampling rates of $< 50\%$.

Figure 5 examines the distribution of tuples. For the tuples that are subject to downsampling (i.e., that have keys popular enough to be sampled), we determine the rate at which that tuple's key is sampled. We then plot the distribution of these sampling rates, with the x-axis of Figure 5 showing the log of the inverse sampling rate (e.g., if a given tuple is subject to a sampling rate of $1/128$, it will be placed in the bin labeled 7). Essentially, tuples placed in farther-right bins have more popular keys. We observe that in many of our benchmarks, the bulk of the tuples are to the right of distribution, an indicator of the skewness of the inputs.

5.2. Errors in Approximation

This section studies the error behavior of MaRSOS. Figure 6 shows the distribution of errors for each of our benchmarks. The plots show histograms of per-key relative error rates. Benchmarks *kmeans, classification, hratings,* and *hmovies* all have a small number of keys, so we present histograms averaged over ten runs. Note that in all of our benchmarks, 95% of the keys have relative errors of less than $\pm 2\%$. MaRSOS is extremely effective at providing low per-key errors.

We next compare to a variant of MaRSOS that performs *uniform* sampling, rather than stratified sampling: all tuples are sampled at the same rate, rather than at different per-key rates. We call this variant UniformHadoop. We set UniformHadoop's global sampling rate such that the number of tuples dropped by UniformHadoop is equal to the total number of tuples dropped by MaRSOS (as in Figure 4). Hence, UniformHadoop and MaRSOS shuffle and reduce the same volume of data. Note that unlike ApproxHadoop, which performs multi-stage sampling [7], UniformHadoop performs simple random sampling. Hence, we would expect ApproxHadoop to have an error *at least as high* as UniformHadoop at the same overall sampling rate. Figure 7 shows the per-key error histograms of UniformHadoop.

The figures clearly show that while MaRSOS is to able to restrict most errors to $\leq 1\%$, UniformHadoop suffers from large variance in its error bounds. In general, most benchmarks have a large number of keys with $\geq 10\%$ error. Worse, benchmarks run on UniformHadoop have several keys with $\pm 100\%$ errors, because those keys are missed entirely during sampling. In contrast, MaRSOS, by design, never misses any keys. This is particularly well highlighted through *invindex*. Figures 7(f) and 6(f) show the benchmark's error with UniformHadoop and MaRSOS respectively. While UniformHadoop shows high errors, due to entirely dropping infrequent words in documents, MaRSOS does not miss any of the word-to-document associations and produces zero error. As mentioned in Section 2.2, such a design choice is highly desirable for applications where rare keys are objects of interest.

However, better error bounds come at the cost of limited performance gains. MaRSOS needs to look at every key before dropping it, and hence cannot randomly skip keys (as in UniformHadoop) or entire map tasks altogether (as in ApproxHadoop [7]). Accordingly, for applications that desire popular key metrics, the latter two can provide better performance. Nevertheless, we reemphasize that MaRSOS's performance cannot be directly compared to the performance of UniformHadoop or ApproxHadoop, as the first provides per-key guarantees, while the latter two do not.

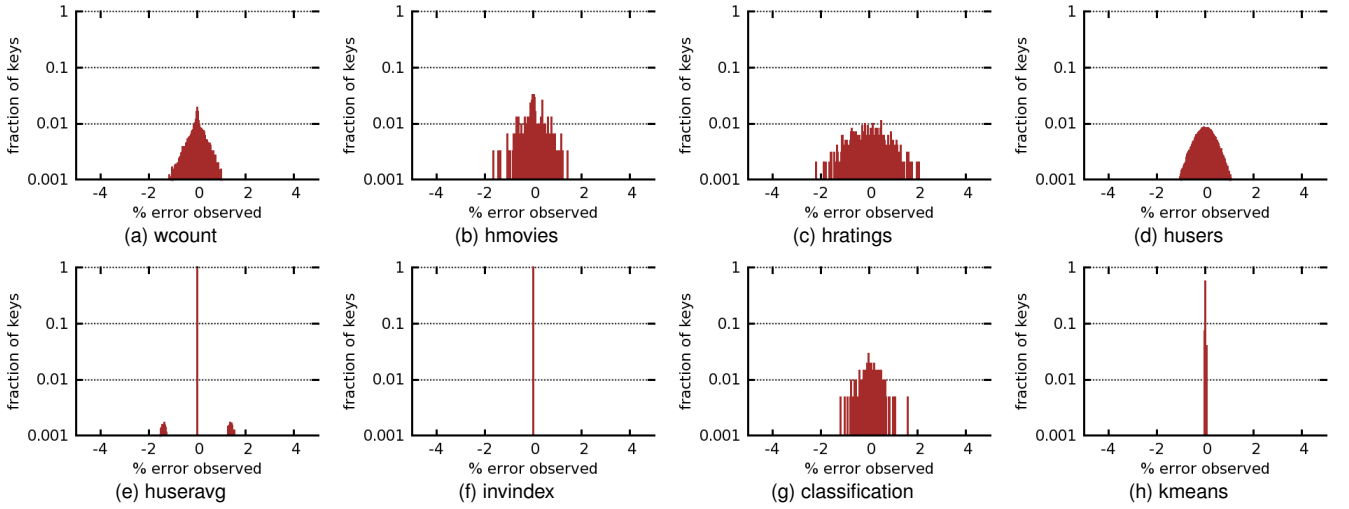

Figure 6: MaRSOS Errors for different benchmarks

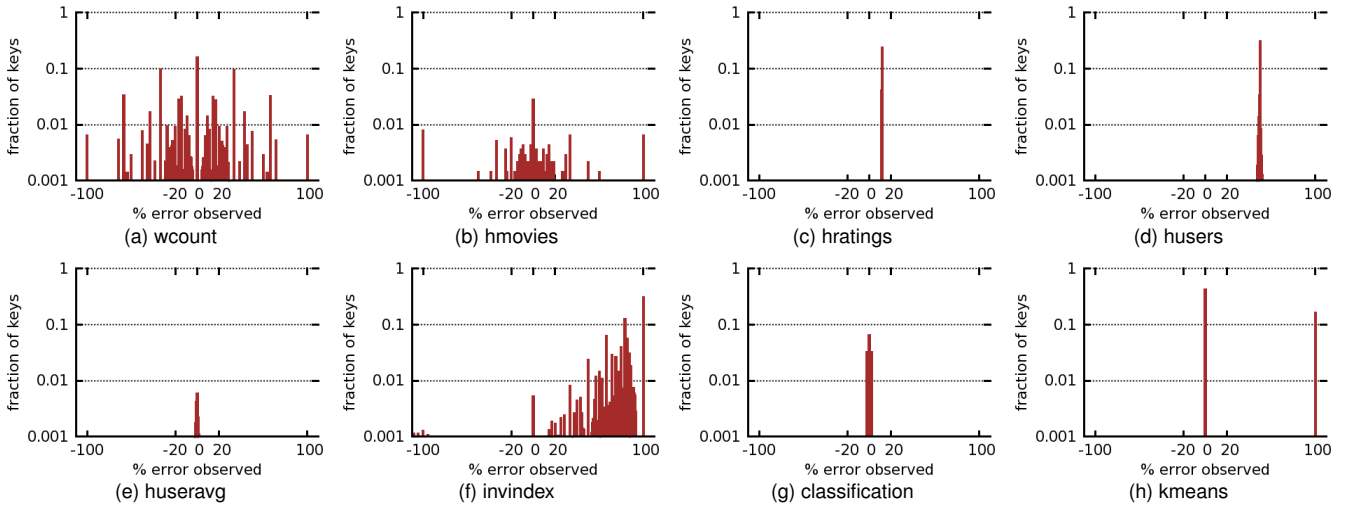

Figure 7: UniformHadoop Errors for different benchmarks

5.3. Impact of n on Performance and Error

Finally, we examine the sensitivity of MaRSOS's performance and accuracy to different choices of n, the target number of samples. Figure 8 shows the speedups over the baseline for different choices of n. As expected, lower ns have higher speedups, but, with the exception of *huseravg* and *husers*, there is not a substantial performance penalty to increasing n, as hypothesized in Section 3.5. The behavior of the two outlier benchmarks can be explained by their much broader distribution of key popularity, as seen in Figure 5—small changes in n can move large numbers of keys between sampling thresholds.

Figure 9 shows that the impact on error due to a change in n is more pronounced. The y-axis here shows the 95th percentile error for each benchmark and choice of n (i.e., only 5% of the keys have higher error).[3] Again, as expected, increasing n decreases the error, but there are diminishing returns.

[3]The benchmark *invindex* is not shown, as it has zero error for $n > 1$

6. Related Work

ApproxHadoop The closest related work to MaRSOS is ApproxHadoop [7], which was the first system to provide general sampling facilities in MapReduce. We first note that the two systems are not truly comparable, as they are designed for different application use cases. ApproxHadoop provides global sampling of the input space, and is best suited for problems where the metrics of interest are not key-specific; in contrast, MaRSOS provides per-key sampling to ensure that rare keys are not missed completely. Due to these different focuses, the results of the two systems are not comparable: ApproxHadoop is able to use multistage sampling to drop entire map tasks, providing potentially very large speedups, but at the cost of missing some keys entirely, and only providing global, not per-key, error bounds. MaRSOS, on the other hand, is bound by the dictates of stratified sampling to inspect each input tuple to at least determine its key; this necessarily limits the amount

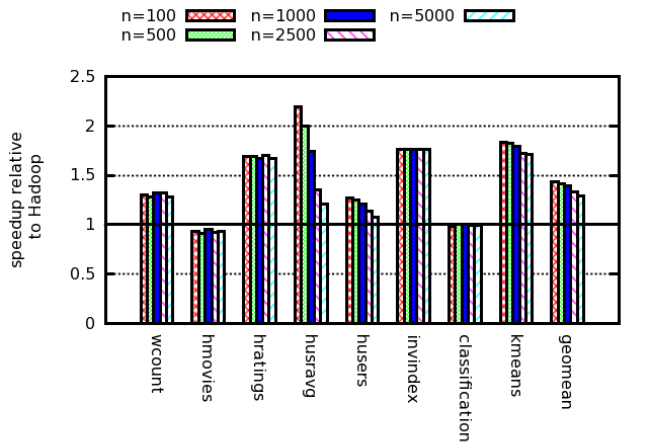

Figure 8: Performance vs *n* (downsampling threshold)

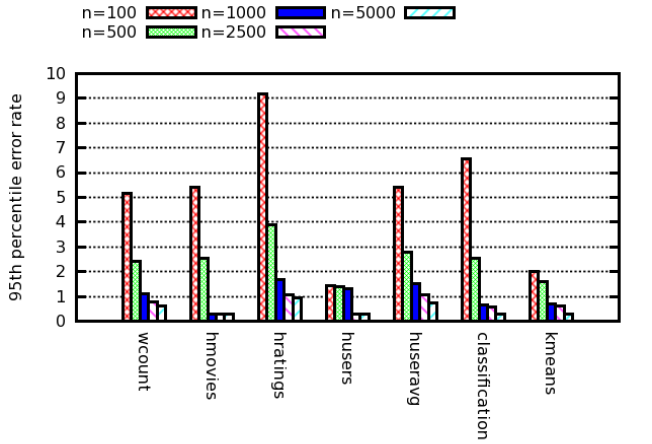

Figure 9: Error vs *n* (downsampling threshold)

of speedup that a MaRSOS program can achieve through sampling compared to an ApproxHadoop program, but MaRSOS provides much stronger quantitative (per-key error bounds) and qualitative (no missed keys) accuracy guarantees.

Unlike ApproxHadoop, MaRSOS does not provide the ability to adjust sampling rates in response to observed variance during execution. This is not a fundamental limitation. The techniques proposed in ApproxHadoop to modify sampling rates based on input characteristics can be applied in MaRSOS by changing the target reservoir size for a given key. As noted in Section 3.5, this type of feedback may not be valid for some problems or inputs. Indeed, many aspects of ApproxHadoop's design—for example, it drops all remaining map tasks if it determines that the error bound has been achieved—are based on strong assumptions about the input (in particular, an assumption that the data are independent and identically distributed) that may not hold in many real world situations.

Stratified sampling Database systems have provided stratified sampling-based queries for decades [1, 3, 5]. These systems inspired the design of MaRSOS, especially our sampling design, which targets a specific number of samples for each key, rather than worrying about particular sampling rates. In particular, Acharya *et al.* provide a proof that for a given number of total samples, the correct distribution of those samples across a set of keys to minimize the worst-case error of any key is to assign each key the same number of samples, regardless of the number of elements with that key [1]. Importantly, these systems rely on offline processing of inputs to build the samples, which are then used to perform queries. The challenges of building an online, distributed stratified sampling system were not addressed. Hellerstein *et al.* present an on-the-fly aggregation system [8] that performs the equivalent of online sampling for DSS. However, its sampling strategy requires randomly reading the input data (to account for, e.g., sorted data), which is far more expensive than the sequential reads of data performed by Hadoop and used by MaRSOS.

Other approximation techniques There has been a surge of interest in approximate computing over the last decade. There are a wide variety of ways of providing approximations, ranging from computing over uncertain inputs, to using approximate memories, to using approximate arithmetic. Perhaps the most closely related approximation techniques to MaRSOS are those that center on *task-dropping*: dropping all but a subset of tasks during execution [9, 11, 12, 13, 15]. The most common technique is *loop perforation*, which randomly (or systematically) skips iterations of loops [12]. Loop perforation implicitly has the effect of sampling, though it does not explicitly perform sampling. Indeed, the most common perforation techniques perform *periodic* sampling [11, 15], keeping every *n*th iteration, which does not have appropriate statistical properties in many cases. Zhu *et al.* look at analogous transformations that specifically target sampling the inputs to reductions, a very close match to the type of sampling that MaRSOS targets [17]. Because these techniques are applied as program transformations, they do not consider the online, distributed sampling problem that we solve.

7. Conclusions

This paper presented MaRSOS, the first MapReduce system that implements stratified sampling to provide disciplined, sound approximation. Unlike prior work in database systems for stratified sampling, MaRSOS implements a novel telescoping sampling algorithm performs stratified sampling online and in a distributed manner, making it suitable for large-scale MapReduce jobs. Unlike ApproxHadoop, earlier work on applying sampling in MapReduce, MaRSOS's use of stratified sampling guarantees that it will not miss keys, and can provide per-key, rather than global, error bounds, making it sound for computations that are sensitive to uncommon keys. MaRSOS is able to deliver substantial performance improvements across a wide variety of benchmarks, with little accuracy loss. Moreover, because its stratified sampling tailors sampling rates to each key, MaRSOS gives *substantially* less error than uniform sampling at the same rate.

References

[1] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 487–498, New York, NY, USA, 2000. ACM.

[2] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[4] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. Technical Report TR-ECE-12-11, Purdue, 2012.

[5] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2), June 2007.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 383–397, New York, NY, USA, 2015. ACM.

[8] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[9] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM.

[10] Sharon L. Lohr. *Sampling: Design and Analysis*. Duxbury Press, 2009.

[11] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[12] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM.

[13] Martin Rinard. Probabilistic accuracy bounds for perforated programs: A new foundation for program analysis and transformation. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 79–80, New York, NY, USA, 2011. ACM.

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[16] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

[17] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM.