Scalable Simple Random Sampling and Stratified Sampling

Xiangrui Meng XIMENG@LINKEDIN.COM
LinkedIn Corporation, 2029 Stierlin Court, Mountain View, CA 94043, USA

Abstract

Analyzing data sets of billions of records has now become a regular task in many companies and institutions. In the statistical analysis of those massive data sets, sampling generally plays a very important role. In this work, we describe a scalable simple random sampling algorithm, named ScaSRS, which uses probabilistic thresholds to decide on the fly whether to accept, reject, or wait-list an item independently of others. We prove, with high probability, it succeeds and needs only $O(\sqrt{k})$ storage, where k is the sample size. ScaSRS extends naturally to a scalable stratified sampling algorithm, which is favorable for heterogeneous data sets. The proposed algorithms, when implemented in MapReduce, can effectively reduce the size of intermediate output and greatly improve load balancing. Empirical evaluation on large-scale data sets clearly demonstrates their superiority.

1 Introduction

Sampling is an important technique in statistical analysis, which consists of selecting some part of a population in order to estimate or learn something from the population at low cost. *Simple random sampling (SRS)* is a basic type of sampling, which is often used as a sampling technique itself or as a building block for more complex sampling methods. However, SRS usually appears in the literature without a clear definition. The principle of SRS is that every possible sample has the same probability to be chosen, but the definition of "possible sample" may vary across different sampling designs. In this work, we consider specifically the simple random sampling without replacement:

Definition 1. *(Thompson, 2012) Simple random sampling is a sampling design in which k distinct items are selected from the n items in the population in such a way that every possible combination of k items is equally likely to be the sample selected.*

Sampling has become more and more important in the era of big data. The continuous increase in data size keeps challenging the design of algorithms. While there have been many efforts on designing and implementing scalable algorithms that can handle large-scale data sets directly, e.g, (Boyd et al., 2011) and (Owen et al., 2011), many traditional algorithms cannot be applied without reducing the data size to a moderate number. Sampling is a systematic and cost-effective way of reducing the data size while maintaining essential properties of the data set. There are many successful work combining traditional algorithms with sampling. For instance, (Dasgupta et al., 2009) show that with proper sampling, the solution to the sub-sampled problem of a linear regression problem is a good approximate solution to the original problem with some theoretical guarantee.

Interestingly, even the sampling algorithms themselves do not always scale well. For example, the reservoir sampling algorithm (Vitter, 1985) for SRS needs storage for k items and reads data sequentially. It may become infeasible for a large sample size, and it is not designed for running in parallel or distributed computing environments. In this work, we propose a scalable algorithm for SRS, which succeeds with high probability and only needs $\mathcal{O}(\sqrt{k})$ storage. It is embarrassingly parallel and hence can be efficiently implemented for distributed environments such as MapReduce.

Discussing the statistical properties of SRS and how SRS compares to other sampling designs is beyond the scope of the paper, we refer readers to (Thompson, 2012) for more details. For a comprehensive discussion of sampling algorithms, we refer readers to (Tillé, 2006). It is also worth noting that there are many work on the design of sampling algorithms at a large scale, for example, (Toivonen, 1996; Chaudhuri et al., 1999; Manku et al., 1999; Leskovec & Faloutsos, 2006). The rest of the paper will focus on the design of scalable SRS algorithms in particular. In Section 2, we briefly

Proceedings of the 30^{th} International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

review existing SRS algorithms and discuss their scalability. In Section 3, we present the proposed algorithm and analyze its properties. We extend it to stratified sampling in Section 4. A detailed empirical evaluation is provided in Section 5.

2 SRS Algorithms

Denote the item set by S which contains n items: $s_1, s_2, \ldots, s_n$. Given a sample size $k \leq n$, let $\mathcal{S}_k$ be the set of all k-subsets of S. By definition, a k-subset of S picked from $\mathcal{S}_k$ with equal probability is a simple random sample of S. Even for moderate-sized k and n, it is impractical to generate all $\binom{n}{k}$ elements of $\mathcal{S}_k$ and pick one with equal probability. In this section, we briefly review existing SRS algorithms, which can generate a simple random sample without enumerating $\mathcal{S}_k$, and discuss their scalability. The content of this section is mostly from (Tillé, 2006).

The naïve algorithm for SRS is the draw-by-draw algorithm. At each step, an item is selected from S with equal probability and then is removed from S. After k steps, we obtain a sample of size k, which is a simple random sample of S. The drawbacks of the draw-by-draw algorithm are obvious. The algorithm requires random access to S and random removal of an item from S, which may become impossible or very inefficient while dealing with large-scale data sets.

The selection-rejection algorithm (Fan et al., 1962), as in Algorithm 1, only needs a single sequential pass to the data and $\mathcal{O}(1)$ storage (not counting the output). This algorithm was later improved by many researchers, e.g., (Ahrens & Dieter, 1985; Vitter, 1987) in order to skip rejected items directly. Despite the improvement, the algorithm stays sequential: whether to select or reject one item would affect all later decisions. It is apparently not designed for parallel environments. Moreover, both k and n need to be explicitly specified. It is very common that data comes in as a stream, where only either k or the sampling probability $p = k/n$ is specified but n is not known until the end of the stream.

Algorithm 1 Selection-Rejection (Fan et al., 1962)

Set $i = 0$.
for j from 1 to n **do**
  With probability $\frac{k-i}{n-j+1}$, select s_j and let $i = i+1$.
end for

(Vitter, 1985) proposed a sampling algorithm with a reservoir, as in Algorithm 2, which needs storage (with random access) for k items but does not require explicit knowledge of n. Nevertheless, the reservoir algorithm is again a sequential algorithm. Like the selection-rejection algorithm, it reads data sequentially in order to get the item indexes correctly, which may take a very long time for large-scale data sets.

Algorithm 2 Reservoir (Vitter, 1985)

The first k items are stored into a reservoir R.
for j from $k+1$ to n **do**
  With probability $\frac{k}{j}$, replace an item from R with equal probability and let s_j take its place.
end for
Select the items in R.

A very simple random sort algorithm was proved by (Sunter, 1977) to be an SRS algorithm, as presented in Algorithm 3. It performs a random permutation of the items via a random sort and then pick the first k items. At the first look, this algorithm needs $\mathcal{O}(n \log n)$ time to perform a random permutation of the data, which is inferior to Algorithms 1 and 2. However, both associating items with independent random variables and sorting can be done efficiently in parallel. As demonstrated in (Czajkowski, 2008), sorting a petabyte-worth of 100-byte records on 4000 computers took just over 6 hours. Moreover, it is easy to see that a complete sort is not necessary for finding the smallest k items, which could be done in linear time using the selection algorithm (Blum et al., 1973). Though Algorithm 3 scales better than Algorithms 1 and 2 in theory, it needs an efficient implementation in order to perform better than Algorithms 1 and 2 in practice. As we will show in the following section, there exists much space for further improvement of this random sort algorithm, which leads to a faster and more scalable algorithm for SRS.

Algorithm 3 Random Sort (Sunter, 1977)

Associate each item of S with an independent variable X_i drawn from the uniform distribution $U(0, 1)$.
Sort S in ascending order.
Select the smallest k items.

3 ScaSRS: A Scalable SRS Algorithm

In this section, we present a fast and scalable algorithm for SRS, named ScaSRS, which gives the same result as the random sort algorithm (given the same sequence of X_i) but uses probabilistic thresholds to accept, reject, or wait-list an item on the fly to reduce the number of items that go into the sorting step. We prove that ScaSRS succeeds with high probability and analyze its theoretical properties. For the simplicity of the analysis, we present the algorithm assuming that

both k and n are given and hence the sampling probability $p = k/n$. Then, in Section 3.4, we consider the streaming case when n is not explicitly given.

3.1 Rejecting Items on the Fly

The sampling probability p plays a more important role than the sample size k in our analysis. Qualitatively speaking, in the random sort algorithm (Algorithm 3), if the random key X_j is "much larger" than the sampling probability p, the item s_j is "very unlikely" to be one of the smallest $k = pn$ items, i.e., to be included in the sample. In this section, we present a quantitative analysis and derive a probabilistic threshold to reject items on the fly. We need the following inequality from (Maurer, 2003):

Lemma 1. *(Maurer, 2003) Let $\{Y_j\}_{j=1}^n$ be independent random variables, $\mathbf{E}[Y_j^2] < \infty$, and $Y_j \geq 0$. Set $Y = \sum_j Y_i$ and let $t > 0$. Then,*

$$\log \mathbf{Pr}\{\mathbf{E}[Y] - Y \geq t\} \leq -\frac{t^2}{2\sum_j \mathbf{E}[Y_j^2]}.$$

Theorem 1. *In Algorithm 3, if we reject items whose associated random keys are greater than*

$$q_1 = \min(1, p+\gamma_1+\sqrt{\gamma_1^2+2\gamma_1 p}), \text{ where } \gamma_1 = -\frac{\log\delta}{n},$$

for some $\delta > 0$. The resulting algorithm is still correct with probability at least $1-\delta$.

Proof. Fix a $q_1 \in [0,1]$ and let $Y_j = \mathbf{1}_{X_j<q_1}$. $\{Y_j\}_{j=1}^n$ are independent random variables, and it is easy to verify that $\mathbf{E}[Y_j] = q_1$ and $\mathbf{E}[Y_j^2] = q_1$. Set $Y = \sum_j Y_j$, which is the number of items whose associated random keys are less than q_1. We have $\mathbf{E}[Y] = \sum_j \mathbf{E}[Y_j] = q_1 n$. Apply Lemma 1 with $t = (q_1 - p)n$,

$$\log \mathbf{Pr}\{Y \leq pn\} \leq -\frac{(q_1-p)^2 n}{2q_1}. \tag{1}$$

We want to choose a $q_1 \in (0,1)$ such that we can reject item s_j immediately if $X_j \geq q_1$, $j = 1,\ldots,n$, and with high probability doing this will not affect the sampling result, i.e., $Y \geq pn$. Given (1), in order to have a failure rate of at most δ for some $\delta > 0$, we need

$$-\frac{(q_1-p)^2 n}{2q_1} \leq \log\delta,$$

which gives

$$q_1 \geq p + \gamma_1 + \sqrt{\gamma_1^2+2\gamma_1 p}, \text{ where } \gamma_1 = -\frac{\log\delta}{n}.$$

This completes the proof. □

By applying the probabilistic threshold q_1, we can reduce the number of items that go into the sorting step.

3.2 Accepting Items on the Fly

Given the analysis in Section 3.1, it is natural to think of the other side: if the random key X_j is "much smaller" than the sampling probability p, then the item s_j is "very likely" to be included in the sample. For a quantitative analysis, this time we need Bernstein's inequality (Bernstein, 1927):

Lemma 2. *(Bernstein, 1927) Let $\{Z_j\}_{j=1}^n$ be independent random variables with $Z_j - E[Z_j] \leq M$ for all $j \in \{1,\ldots,n\}$. Let $Z = \sum_j Z_j$ and $t > 0$. Then with $\sigma_j^2 = \mathbf{E}[Z_j^2] - \mathbf{E}[Z_j]^2$ we have*

$$\log \mathbf{Pr}\{Z - \mathbf{E}[Z] \geq t\} \leq -\frac{t^2}{2\sum_j \sigma_j^2 + 2Mt/3}.$$

Theorem 2. *In Algorithm 3, if we accept items whose associated random keys are less than*

$$q_2 = \max(0, p+\gamma_2-\sqrt{\gamma_2^2+3\gamma_2 p}), \text{ where } \gamma_2 = -\frac{2\log\delta}{3n},$$

for some $\delta > 0$. The resulting algorithm is still correct with probability at least $1-\delta$.

Proof. Fix a $q_2 \in [0,1]$ and let $Z_j = \mathbf{1}_{X_j<q_2}$. $\{Z_j\}_{j=1}^n$ are independent random variables. It is easy to verify that $\mathbf{E}[Z_j] = q_2$, $Z_j - \mathbf{E}[Z_j] \leq 1 - q_2 \leq 1$, and $\sigma_j^2 = \mathbf{E}[Z_j^2] - \mathbf{E}[Z_j]^2 \leq \mathbf{E}[Z_j^2] = q_2$. Consider $Z = \sum_j Z_j$, which is the number of items whose associated random keys are less than q_2. We have $\mathbf{E}[Y] = \sum_j \mathbf{E}[Y_j] = q_2 n$. Applying Lemma 2 with $t = (p-q_2)n$, we get

$$\log \mathbf{Pr}[Z \geq pn] \leq -\frac{3(p-q_2)^2 n}{4q_2+2p}.$$

The proof is done by similar arguments as in the proof of Theorem 1. □

By applying the probabilistic threshold q_2 together with q_1 from Section 3.1, we can further reduce the number of items that go into the sorting step.

3.3 The Algorithm

The scalable SRS algorithm we propose, referred to as ScaSRS, is simply the random sort algorithm plus the probabilistic thresholds introduced in Theorems 1 and 2. We describe ScaSRS in Algorithm 4.

Theorem 3. *ScaSRS (Algorithm 4) succeeds with probability at least $1-2\delta$. Moreover, for a fixed δ and with high probability, it only needs $\mathcal{O}(\sqrt{k})$ storage (not counting the output) and runs in $\mathcal{O}(n)$ time.*

Proof. Adopt the notation from Sections 3.1 and 3.2. The output of ScaSRS will not be affected by setting

Algorithm 4 ScaSRS: Scalable SRS

Choose a small $\delta > 0$ which controls the failure rate.
Compute q_1 and q_2 based on Theorems 1 and 2.
Let $l = 0$, and $W = \emptyset$ be the waiting list.
for each item $s_j \in S$ **do**
  Draw a key X_j independently from $U(0, 1)$.
  if $X_j < q_2$ **then**
    Select s_j and let $l := l + 1$.
  else if $X_j < q_1$ **then**
    Associate s_j with X_j and add it into W.
  end if
end for
Sort W's items in the ascending order of the key.
Select the smallest $pn - l$ items from W.

the probabilistic thresholds if we have both $Y \geq pn$ and $Z \leq pn$, i.e., we do not reject more than $n - k$ items and we do not select more than k items during the scan. Then, by Theorems 1 and 2, we know that ScaSRS succeeds with probability $1 - 2\delta$. Let $w = Y - Z$ be the final size of W. It is easy to see that the storage requirement of the algorithm is $\mathcal{O}(w)$. Remember that a complete sort is not necessary to find the smallest $k - l$ items from W. Instead, we can use the selection algorithm that takes linear time. Therefore, the total running time is $\mathcal{O}(n + w) = \mathcal{O}(n)$. In the following, we prove that, for a fixed δ and with high probability, $w = \mathcal{O}(\sqrt{k})$. We have

$$\begin{aligned}\mathbf{E}[Y] &= q_1 n = pn - \log\delta + \sqrt{\log^2\delta - 2\log\delta \cdot pn} \\ &= k + \mathcal{O}(\sqrt{k}), \\ \mathbf{E}[Z] &= q_2 n = pn - \frac{2}{3}\log\delta - \sqrt{\frac{4}{9}\log^2\delta - 2\log\delta \cdot pn} \\ &= k - \mathcal{O}(\sqrt{k}).\end{aligned}$$

Choose a small positive number θ. Apply Lemma 1 to $Z = \sum_j Z_j$ with $t = \sqrt{2\log\theta \cdot q_2 n}$,

$$\mathbf{Pr}\{Z \leq q_2 n - \sqrt{2\log\theta \cdot q_2 n}\} \leq \theta.$$

Since θ only appears in the log term of the bound, with high probability we have $Z = k - \mathcal{O}(\sqrt{k})$. Similarly, by applying Lemma 2 to $Y = \sum_j Y_j$, we can get $Y = k + \mathcal{O}(\sqrt{k})$ with high probability, and hence $w = Y - Z = \mathcal{O}(\sqrt{k})$ with high probability. □

To illustrate the result, let $n = 10^6$, $k = 50000$, and hence $p = 0.05$, and then compute q_1 and q_2 based on n, p, and $\delta = 0.00005$. We plot the probability density functions (pdf) of Y and Z in Figure 1. We see that with high probability, $Y \geq k$, $Z \leq k$, and $Y - Z = \mathcal{O}(\sqrt{k})$.

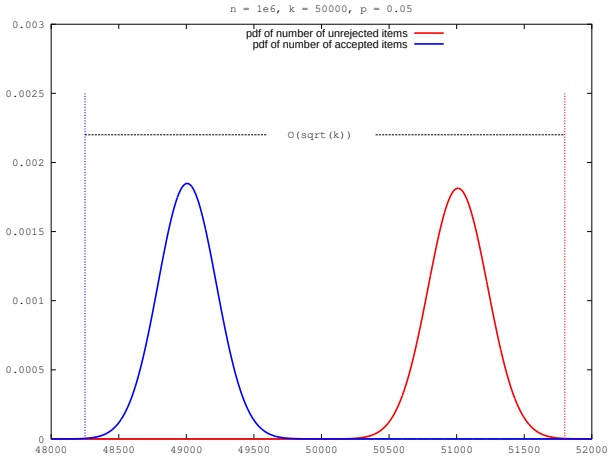

Figure 1. The probability density functions of Y, number of unrejected items, and Z, number of accepted items. With high probability, $Y \geq k$, $Z \leq k$, and $Y - Z = \mathcal{O}(\sqrt{k})$.

From the proof, we see that, with high probability, $n - k - \mathcal{O}(\sqrt{k})$ items are rejected and $k - \mathcal{O}(\sqrt{k})$ items are accepted right after the corresponding random keys $\{X_j\}$ are generated. Only $\mathcal{O}(\sqrt{k})$ items stay in the waiting list and go into the sorting step, instead of n items for the random sort algorithm. The improvement is significant. For example, let us consider the task of generating a simple random sample of size 10^6 from a collection of 10^9 items. The random sort algorithm needs to find the smallest 10^6 items according to the random keys out of 10^9, while ScaSRS only needs to find approximately the smallest a few thousand items out of a slightly larger pool that is still at the order of a few thousand, which is a trivial task.

In ScaSRS, the decision of whether to accept, reject, or wait-list an item s_j is made solely on the value of X_j, independent of others. Therefore, it is embarrassingly parallel, which is a huge advantange over the sequential Algorithms 1 and 2 and makes ScaSRS more favorable for large-scale data sets in distributed environments. Even in the single-threaded case, ScaSRS only loses to Algorithm 1 by $\mathcal{O}(\sqrt{k})$ storage for the waiting-list and $\mathcal{O}(\sqrt{k})$ time for the sorting step. Similar to the improvement that has been made to Algorithm 1, we can skip items in ScaSRS. If $X_j < q_1$, let T_j be the smallest number greater than j such that $X_{T_j} < q_1$. We know that $T_j - j$ is a random number follows the geometric distribution. Therefore, we can generate $T_j - j$ directly and skip/reject all items between s_j and s_{T_j}. It should give a performance boost provided random access to the data.

ScaSRS is a randomized algorithm with a certain chance of failure. However, since δ only appears in

the log terms of the thresholds q_1 and q_2, we can easily control the failure rate without sacrificing the performance much. In practice, we set $\delta = 0.00005$ and hence the failure rate is controlled at 0.01%, which makes ScaSRS a quite reliable algorithm. With this setting, we have not encountered any failures yet during our experiments and daily use.

3.4 ScaSRS in Streaming Environments

For large-scale data sets, data usually comes in as a stream and the total number of items is unknown until we reach the end of the stream. Algorithm 1 does not apply to the streaming case. Algorithm 2 only needs the sample size k to work at the cost of a reservoir of k items. In this section, we discuss how ScaSRS works in streaming environments. We consider two scenarios: 1) when the sampling probability p is given, 2) when the sample size k is given.

If only p is given, we can update the thresholds q_1 and q_2 on the fly by replacing n by the number of items seen so far. Instead of fixed thresholds, we set

$$q_{1,j} = \min(1, p + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}p}), \quad (2)$$

$$q_{2,j} = \max(0, p + \gamma_{2,j} - \sqrt{\gamma_{2,j}^2 + 3\gamma_{2,j}p}), \quad (3)$$

where

$$\gamma_{1,j} = -\frac{\log \delta}{j},\ \gamma_{2,j} = -\frac{2\log \delta}{3j}, \quad j = 1, \ldots, n.$$

It can only loosen the thresholds and therefore the failure rate of the algorithm will not increase. However, this will increase the size of the waiting list. For simplicity, we present the theoretical analysis for the single-threaded case and then show how to get a near-optimal performance in parallel environments. We summarize the modified algorithm in Algorithm 5 and analyze its properties in Theorem 4.

Algorithm 5 ScaSRS (when only p is given)

```
Choose a small δ > 0 which controls the failure rate.
Let l = 0, and W = ∅ be the waiting list.
for j = 1 to n do
    Compute q_{1,j} and q_{q,2} based on (2) and (3).
    Draw a key X_j independently from U(0, 1).
    if X_j < q_{2,j} then
        Select s_j and let l := l + 1.
    else if X_j < q_{1,j} then
        Associate s_j with X_j and add it into W.
    end if
end for
Sort W's items in the ascending order of the key.
Select the first pn − l items from W.
```

Theorem 4. *Algorithm 5 succeeds with probability at least $1 - 2\delta$. Moreover, for a fixed δ and with high probability, it only needs $\mathcal{O}(\log n + \sqrt{k \log n})$ storage.*

Proof. Redefine random variables $Y_j = \mathbf{1}_{X_j < q_{1,j}}$, $j = 1, \ldots, n$ and $Y = \sum_j Y_j$. Hence, $\mathbf{E}[Y_j] = \mathbf{E}[Y_j^2] = q_{1,j}$ and $\mathbf{E}[Y] = \sum_j \mathbf{E}[Y_j] = \sum_j q_{1,j}$. Note that

$$\sum_{j=1}^{n} \frac{1}{j} \le \log n + 1, \quad \forall n \in \mathbb{N}$$

and

$$\sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}p} \le \gamma_{1,j} + \sqrt{2\gamma_{1,j}p}, \quad j = 1, \ldots, n.$$

With a fixed δ, we get the following bound of $\mathbf{E}[Y]$:

$$\begin{aligned}
\mathbf{E}[Y] &\le \sum_j \left(p + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}p}\right) \\
&\le pn + \sum_j \gamma_{1,j} + \sum_j (\gamma_{1,j} + \sqrt{2\gamma_{1,j}p}) \\
&\le k + 2\sum_j \gamma_{1,j} + \sqrt{2pn \sum_j \gamma_{1,j}} \\
&\le k - 2\log \delta(\log n + 1) + \sqrt{-2k \log \delta(\log n + 1)} \\
&= k + \mathcal{O}(\log n + \sqrt{k \log n}).
\end{aligned}$$

Then applying Lemma 2 to Y, we know that $Y = k + \mathcal{O}(\log n + \sqrt{k \log n})$ with high probability. By similar arguments, we can obtain $Z = k - \mathcal{O}(\log n + \sqrt{k \log n})$ with high probability. So the size of W is $\mathcal{O}(\log n + \sqrt{k \log n})$ with high probability, which is slightly larger than the size of the waiting list when n is given. □

When we process the input data in parallel, it may be too expensive to know in real time the exact j, the number of items that have been processed. Fortunately, the algorithm does not require the exact j to work correctly but just a good lower bound. So we can simply replace j by the local count on each process or a global count updated less frequently to reduce communication cost. The former approach works very well in practice, even with hundreds of concurrent processes.

If only k is given, we can no longer accept items on the fly because the sampling probability could be arbitrarily small. However, we can still reject items on the fly based on k and j, the number of items that have been processed. The following threshold can be used:

$$q_{1,j} = \min(1, \frac{k}{j} + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}\frac{k}{j}}), \quad (4)$$

where

$$\gamma_{1,j} = -\frac{\log \delta}{j}, \quad j = 1, \ldots, n.$$

Similar to the previous case, we redefine random variables $Y_j = \mathbf{1}_{X_j<q_{1,j}}$, $j = 1,\ldots,n$, and $Y = \sum_j Y_j$. Then compute an upper bound of $\mathbf{E}[Y]$:

$$\begin{aligned}\mathbf{E}[Y] &\le \sum_j \frac{k}{j} + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}\frac{k}{j}} \\ &\le k(\log n + 1) + \mathcal{O}(\sqrt{k}\log n).\end{aligned}$$

By Lemma 2, we know that with high probability, $Y = k(\log n+1)+\mathcal{O}(\sqrt{k}\log n)$. We omit the detailed proof and present the modified algorithm in Algorithm 6 and the theoretical result in Theorem 5.

Algorithm 6 ScaSRS (when only k is given)

```
Choose a small δ > 0 which controls the failure rate.
Let l = 0, and W = ∅ be the waiting list.
for j = 1 to n do
    Compute q_{1,j} based on (4).
    Draw a key X_j independently from U(0,1).
    if X_j < q_{1,j} then
        Associate s_j with X_j and add it into W.
    end if
end for
Sort W's items in the ascending order of the key.
Select the first k items from W.
```

Theorem 5. *Algorithm 6 succeeds with probability at least $1-\delta$. Moreover, for a fixed δ and with high probability, it needs $k(\log n + 1) + \mathcal{O}(\sqrt{k}\log n)$ storage.*

Compared with Algorithm 2, Algorithm 6 needs more storage. However, it does not need random access to the storage during the scan, so the items in the waiting list can be distributively stored. Moreover, the algorithm can be implemented in parallel provided a way to obtain a good lower bound of the global item count j. This can be done by setting up a global counter, and each process reports its local count and fetches the global count every, e.g., 1000 items.

4 Stratified Sampling

If the item set S is heterogeneous, which is common for large-scale data sets, it may be possible to partition it into several non-overlapping homogeneous subsets, called *strata*, denoted by $S_1,\ldots,S_m$. By "homogeneous" we mean the items within a stratum are similar to each other. For example, a training set can be partitioned into positives and negatives, or web users' activities can be partitioned based on the days of the week. Given strata of S, applying SRS within each stratum is preferred to applying SRS to the entire set for better representativeness of S. This approach is called *stratified sampling*. See (Thompson, 2012) for its theoretical properties. In this work, we consider stratified sampling with proportional allocation, in which case the sample size of each stratum is proportional to the size of the stratum.

Let p be the sampling probability and n_i be the size of S_i, $i = 1,\ldots,m$. We want to generate a simple random sample of size pn_i from S_i for each i. For simplicity, we assume that pn_i is an integer for all i. Extending ScaSRS to stratified sampling is straightforward, since stratified sampling is equivalent to applying SRS to $S_1,\ldots,S_m$ with sampling probability p.

Theorem 6. *Let S be an item set of size n, which is partitioned into m strata $S_1,\ldots,S_m$. Assume that the size of S_i is given, denoted by n_i, $i = 1,\ldots,m$. Given a sampling probability p, if ScaSRS is applied to each stratum of S to compute a stratified sample of S, it succeeds with probability at least $1-2m\delta$, and for a fixed δ and with high probability, it needs at most $\mathcal{O}(\sqrt{mpn})$ storage.*

Proof. To generate a stratified sample of S, we need to apply ScaSRS m times. Therefore, the failure rate is at most $2m\delta$. The total size for the waiting lists is

$$\sum_i \mathcal{O}(\sqrt{pn_i}) \le \mathcal{O}(\sqrt{m}\sqrt{p\cdot\sum_i n_i}) = \mathcal{O}(\sqrt{mpn}),$$

with high probability. □

In Theorem 6, we assume that n is explicitly given. In practice, it is common that only p is specified, but n_i, $i = 1,\ldots,m$, and n are all unknown or we need to make a pass to the data to get those numbers. Using Algorithm 5, we can handle this streaming case quite efficiently as well:

Theorem 7. *If only the sampling probability p is given and Algorithm 5 is applied to each stratum of S to compute a stratified sample of S, it succeeds with probability at least $1-2m\delta$, and for a fixed δ and with high probability, it needs $\mathcal{O}(m\log n + \sqrt{mpn\log n})$ storage.*

We omit the proof because it is very similar to the proof of Theorem 6. Similarly, using Algorithm 6, we can handle the other streaming case when only k is specified. However, since we do not know $k_i = kn_i/n$ in advance, $i = 1,\ldots,m$, we cannot apply Algorithm 6 directly to each stratum. Suppose we process items from S sequentially. Instead of (4), we can use the following threshold for s_j:

$$q_{1,j} = \min(1, \frac{k}{j} + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}\frac{k}{j}}), \quad (5)$$

where

$$\gamma_{1,j} = -\frac{\log \delta}{j_{i(j)}}, \quad j = 1, \ldots, n,$$

$i(j)$ is the index of the stratum s_j belongs to, and j_i is the number of items seen from S_i at step j. So we use the global count to compute an upper bound of p and use the local count as a lower bound of n_i. It leads to the following theorem:

Theorem 8. *If only the sample size k is given and a modified Algorithm 6 using the thresholds from* (5) *is applied to S, it succeeds with probability at least $1-m\delta$, and for a fixed δ and with high probability, it needs at most $(k+m)(\log n + 1) + \mathcal{O}(\sqrt{km}\log n)$ storage.*

Proof. Note that the threshold from (5) is always larger than the one from Theorem 1. Therefore, for each stratum, the failure rate is at most δ and hence the overall failure rate is at most $m\delta$. It is easy to derive the following bound:

$$\sum_j \frac{1}{j_{i(j)}} \le m \sum_j \frac{1}{j} \le m(\log n + 1).$$

Similar to the proof of Theorem 5, we have

$$\begin{aligned}\mathbf{E}[Y] &\le \sum_j \frac{k}{j} + \gamma_{1,j} + \sqrt{\gamma_{1,j}^2 + 2\gamma_{1,j}\frac{k}{j}} \\ &\le (k+m)(\log n + 1) + \mathcal{O}(\sqrt{km}\log n).\end{aligned}$$

Even if k_i is given and hence we can directly apply Algorithm 6 to each stratum, by Theorem 5, the bound on storage we can obtain is

$$\begin{aligned}&\sum_i k_i(\log n_i + 1) + \mathcal{O}(\sqrt{k_i}\log n_i) \\ &\le k(\log n + 1) + \mathcal{O}(\sqrt{km}\log n).\end{aligned}$$

Usually, m is a small number. Therefore, the overhead introduced by not knowing k_i is quite small. □

An efficient parallel implementation of the stratified sampling algorithm using ScaSRS (when only k is given) would need a global counter for each stratum. The counters do not need to be exact. Good lower bounds of the exact numbers should work well.

5 Implementation and Evaluation

We implemented ScaSRS (Algorithm 4) and its variant Algorithm 5 using Apache Hadoop[1], which is an open-source implementation of Google's MapReduce framework (Dean & Ghemawat, 2008). We did not implement Algorithm 6 due to the lack of efficient near real-time global counters in Hadoop and it is also because that in most practical tasks p is specified instead of k, especially for stratified sampling. The MapReduce framework consists of three phases: map, sort, and reduce. In the map phase, input data is processed by concurrent mappers, which generate key-value pairs. In the sort phase, the key-value pairs are ordered by the key. Then, in the reduce phase, the values associated with the same key will be processed by a reducer, where multiple reducers can be used to accelerate the process if we have multiple keys. The reducers' output is the final output. See (White, 2012) for more details.

ScaSRS fits the MapReduce framework very well. In the map phase, for each input item we generate a random key and decide whether to accept, reject, or wait-list the item. If we decide to put the item onto the waiting list, we let the mapper emit the random key and the item as a key-value pair. Then in the sort phase, the items in the waiting list are sorted by their associated keys. Finally, the sorted items are processed by a reducer, which selects the first $k - l$ items into the sample where l is the number of accepted items in the map phase. It is technically possible to let mappers output accepted items directly (using MultipleOutputs), and the result we report here is based on this approach. In our implementation, we also have an option to let accepted items go through the sort and the reduce phases in order to control the number of final output files and the size of each file, which is a practical workaround to prevent file system fragmentation. If this option is enabled, we associate the accepted items with a special key such that the reducers know that those items have been already accepted, and we also use a random key partitioner which assigns an accepted item to a random reducer in order to help load balancing. If only p is given, we use local item counts to update the thresholds on the fly. We set $\delta = 0.00005$, which controls the failure rate at 0.01% and hence leads to a reliable algorithm.

For comparison purpose, we also implemented Algorithms 1 (referred to as SR), 2 (referred to as R) and 3 (referred to as RS) using Hadoop. For SR and R, we use identity mappers that do nothing but copy the input data. The actual work is done on a single reducer. So, to be fair, we only use the time of the reduce phase as the running time of the algorithm. Note that we cannot use multiple reducers because the algorithms are sequential. This single reducer has to go through the entire data set to generate a simple random sample, which clearly becomes the bottleneck. For RS, mappers associate input items with random keys. After the sort, a single reducer output the first k items as the sample. This is a naïve implementation of RS. We did not implement the selection algorithm but simply

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

use the sort capability from Hadoop. One drawback of conforming to the MapReduce framework is that the entire data set will be fed to the reducer though we know that only the first k items are neccessary. Instead of implementing a distributed selection algorithm and plugging it into the MapReduce framework, using ScaSRS is apparently a better choice. Recall that ScaSRS, if it runs successfully, outputs the same result as RS given the same sequence of random keys.

The test data sets we use are user-generated events from LinkedIn's website, which scales from 60 million records to 1.5 billion. Table 1 lists the test problems. By default, the number of mappers is proportional to

	P_1	P_2	P_3	P_4	P_5	P_6
n	6.0e7	6.0e7	3.0e8	3.0e8	1.5e9	1.5e9
p	0.01	0.1	0.01	0.1	0.01	0.1
k	6.0e5	6.0e6	3.0e6	3.0e7	1.5e7	1.5e8

Table 1. Test problems.

the input data size. We use this default setting. For problems P_1 and P_2 we use 17 mappers, for P_3 and P_4 we use 85 mappers, and for P_5 and P_6 we use 425 mappers. Only one reducer is needed to process the waiting list. Note that the number of mappers does not affect the performance of SR and R, which can only rely on a single reducer.

First, we compare the running times of SR, RS, ScaSRS with n given, R with only k given, and ScaSRS with only p given (referred to as ScaSRS$_p$). Recall that we only measure the running time of the reduce phase for SR and R for fairness. Table 2 shows the running times in seconds. The running time of each test is

	P_1	P_2	P_3	P_4	P_5	P_6
SR	281	355	1371	1475	>3600	>3600
R	288	299	1285	1571	>3600	>3600
RS	513	581	1629	2344	>3600	>3600
ScaSRS	96	103	126	127	140	158
ScaSRS$_p$	98	144	109	139	162	214

Table 2. Running times (in seconds). SR and R are sequential algorithms with linear time complexity and hence their running times are approximately proportional to the input size. RS could be scalable if we implemented a distributed selection algorithm. However, it is easier to use ScaSRS instead, which scales very well across all tests.

based on a single run, and we terminate the job if the running time is longer than one hour. The running time of a Hadoop job can be affected by many factors, for example, I/O performance, concurrent jobs, etc. We did not take the average out of several runs, because the numbers are adequate to demonstrate the superiority of ScaSRS, whose running time grows very slowly as the problem scale increases.

Next, we verify our claim in Theorems 3 and 4 about the storage requirement, i.e., the size of the waiting list, denoted by w for ScaSRS and by w_p for ScaSRS$_p$. We list the result in Table 3. It is easy to verfiy

	P_1	P_2	P_3	P_4	P_5	P_6
k	6.0e5	6.0e6	3.0e6	3.0e7	1.5e7	1.5e8
w	6.9e3	2.2e4	1.6e4	4.9e4	3.4e4	1.1e5
w_p	5.8e4	1.8e5	2.9e5	9.1e5	1.5e6	4.5e6

Table 3. Waiting list sizes. The waiting list sizes of ScaSRS confirm our theory in Theorem 3: $w = \mathcal{O}(\sqrt{k})$. ScaSRS$_p$ requires more (but still tractable) storage than ScaSRS.

that $w < 10\sqrt{k}$, which confirms our theory in Theorem 3. We implement ScaSRS$_p$ using local counts instead of a global count, which introduces an overhead on the storage requirement. Nevertheless, the storage requirement of ScaSRS$_p$ is still moderate even for P_6, which needs a sample of size $1.5e8$.

Finally, we report a stratified sampling task we ran on a large data set (about 7 terabytes), which contains 23.25 billion user-generated events. The stratum of an event is determined by a field in the record. The data set contains 8 strata. The ratio between the size of the largest strata and that of the smallest strata is approximately 15000. We set the sampling probability $p = 0.01$ and use approximately 3000 mappers and 5 reducers for the sampling task. The job finished successfully in 509 seconds. The total size of the waiting lists is 4.3e7. Within the waiting list, the ratio between the size of the largest strata and that of the smallest strata is 861.2, which helps the load balancing.

6 Conclusion

We developed and implemented a scalable simple random sampling algorithm, named ScaSRS, which uses probabilistic thresholds to accept, reject, or wait-list items on the fly. We presented a theoretical analysis of its success rate and storage requirement, and discussed its variants for streaming environments. ScaSRS is very easy to implement in the MapReduce framework and it extends naturally to stratified sampling. Empirical evaluation on large-scale data sets showed that ScaSRS is reliable and has very good scalability.

Acknowledgments

The author would like to thank Paul Ogilvie and Anmol Bhasin for their valuable feedback on an earlier draft. The author also wishes to thank the anonymous reviewers for their comments and suggestions that helped improve the presentation of the paper.

References

Ahrens, J. H. and Dieter, U. Sequential random sampling. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):157–169, 1985.

Bernstein, S. *Theory of Probability*. Moscow, 1927.

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

Chaudhuri, S., Motwani, R., and Narasayya, V. On random sampling over joins. *ACM SIGMOD Record*, 28(2):263–274, 1999.

Czajkowski, G. Sorting 1PB with MapReduce, 2008. http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

Dasgupta, A., Drineas, P., Harb, B., Kumar, R., and Mahoney, M. W. Sampling algorithms and coresets for ℓ_p regression. *SIAM Journal on Computing*, 38 (5):2060–2078, 2009.

Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Fan, C. T., Muller, M. E., and Rezucha, I. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57 (298):387–402, 1962.

Leskovec, J. and Faloutsos, C. Sampling from large graphs. In *Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining*, pp. 631–636. ACM, 2006.

Manku, G. S., Rajagopalan, S., and Lindsay, B. G. Random sampling techniques for space efficient online computation of order statistics of large datasets. *ACM SIGMOD Record*, 28(2):251–262, 1999.

Maurer, A. A bound on the deviation probability for sums of non-negative random variables. *J. Inequalities in Pure and Applied Mathematics*, 4, 2003.

Owen, S., Anil, R., Dunning, T., and Friedman, E. *Mahout in Action*. Manning Publications Co., 2011.

Sunter, A. B. List sequential sampling with equal or unequal probabilities without replacement. *Applied Statistics*, pp. 261–268, 1977.

Thompson, S. K. *Sampling*. Wiley, 3 edition, 2012.

Tillé, Y. *Sampling Algorithms*. Springer, 2006.

Toivonen, H. Sampling large databases for association rules. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pp. 134–145. Morgan Kaufmann Publishers Inc., 1996.

Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

Vitter, J. S. An efficient algorithm for sequential random sampling. *ACM transactions on mathematical software (TOMS)*, 13(1):58–67, 1987.

White, T. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012.