KSample: Dynamic Sampling Over Unbounded Data Streams

Tiago Rodrigo Kepe[1], Eduardo Cunha de Almeida[1], Thomas Cerqueus[2]

[1] Universidade Federal do Paraná, Brazil
[trkepe,eduardo]@inf.ufpr.br
[2] Université de Lyon, CNRS, INSA-Lyon, LIRIS, France
thomas.cerqueus@insa-lyon.fr

Abstract.

Data sampling over data streams is common practice to allow the analysis of data in real-time. However, sampling over data streams becomes complex when the stream does not fit in memory, and worse yet, when the length of the stream is unknown. A well-known technique for sampling data streams is the Reservoir Sampling. It requires a fixed-size reservoir that corresponds to the resulting sample size. But, defining the reservoir size is challenging: huge samples may waste computing resources and may not fit in memory; whereas tiny samples may be inadequate and prevent from drawing meaningful conclusions. This article presents KSample, a novel data sampling algorithm over unbounded data streams. It does not require to know the length of the stream or the size of the sample. The key idea of KSample is based on an invariant that keeps the percentage of the stream regardless of its length. That is the reservoir invariably represents at least the target percentage of the stream. KSample eliminates the problem of memory space by defining the concept of distributed mini-reservoirs grounded on the same invariant. Experiments show that KSample is substantially faster than the Reservoir Sampling algorithm to generate samples. Finally, KSample was put in practice to speed up data analytics over MapReduce jobs, reducing their response times by up to a factor of 20.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous; H.3 [**Information Storage and Retrieval**]: Miscellaneous

Keywords: Data Analytics, Data Sampling, Data Stream

1. INTRODUCTION

Data stream is an inherent property of different data-centric systems, including Internet databases [Chen et al. 2000], continuous query systems [Madden et al. 2002] [Arasu et al. 2003], sensor networks [Madden and Franklin 2002], mobile devices [Boutsis et al. 2013], social networks [Diaz-Aviles et al. 2012] and data mining [Papini et al. 2014a; 2014b]. In many companies extracting information from data streams is crucial to infer business trends and, consequently, focus efforts on right markets by applying prompt actions and take advantage of business opportunities. For instance, on stock markets, investors have to make quick decisions based on data streams of the stock exchange. Delayed decisions on new entrants might lead to costly consequences.

Besides time constraints, other problems arise when memory space is insufficient for building analytics over data streams. A well-accepted solution lies in computing data samples. A state-of-the-art algorithm to sample data streams is the Reservoir Sampling (RS) [Vitter 1985; Bonin et al. 2014]. It has been successfully applied in many contemporary systems, such as real-time social streams (e.g., Twitter) [Chen et al. 2013], text streams [Bonin et al. 2014] and Web-based stream-data systems[1] [2]. The RS computes samples as fresh data arrives from the stream and keeps them in a memory buffer called reservoir. The RS also replaces legacy data in the reservoir to keep the sample fresh. However, the RS requires a pre-defined fixed size of the sample that bounds the size of the reservoir. Defining the optimal size is challenging: huge samples may waste computing resources and may not fit in memory; whereas tiny samples may be inadequate and compromise drawing meaningful conclusions [Cochran 1977].

Al-Kateb et al. [2007] proposed an adaptive-size reservoir sampling over data streams. In their work, the size of the reservoir is not fixed, it is adjusted according to the characteristics of the data stream: if there is any change in the data characteristics, then the reservoir is resized. However, memory space is not taken into account and overfilling reservoirs is prohibited. Alternatively, other approaches focus on preserving memory space by using sliding windows over unbounded data streams [Braverman et al. 2009; Srivastava and Widom 2004], but the size of the window must be externally provided.

This article presents KSample, a new sampling technique to address two problems of the state-of-the-art techniques: (i) the size of the reservoir, and (ii) the space of main memory. The first one is addressed by introducing an invariant: at any iteration of KSample the resulting sample invariably represents at least the target percentage of the stream regardless of its length. The second one is addressed by distributing the reservoir as mini-reservoirs, also called slots (each slot respects the invariant). Several elements from a given data stream may compete for the same slot, but only statistically chosen elements are kept in the slot. When the slot reaches the target percentage, a new empty slot is created to be disputed by the incoming fresh elements. In time, slots that are no longer disputed are flushed away to secondary storage to save main memory space. Experiments show that KSample is substantially faster than the Reservoir Sampling algorithm to generate samples. KSample was also put in practice to speed up data analytics over MapReduce jobs, reducing their response times by a factor of 20.

The remainder of this article is structured as follows. Section 2 presents the related work. Section 3 presents the Reservoir Sampling algorithm, on which KSample is based. KSample is introduced in Section 4, along with the proof of its invariant. Section 5 evaluates KSample. We first show the accuracy and efficiency of KSample in terms of response time (Sections 5.1 and 5.2). Then, we present two practical use cases of KSample on MapReduce processing (Sections 5.3 and 5.4). Section 6 concludes this article.

2. RELATED WORK

Sampling over data streams is a strategy applied to rapidly infer information without prior knowledge of the stream length, especially for continuous query systems [Madden et al. 2002] [Arasu et al. 2003]. The Reservoir Sampling [Vitter 1985] is a well-known and widely used algorithm to sample data streams [Gama 2010]. However, approaches based on it [Chen et al. 2013; Bonin et al. 2014] require a fixed-size reservoir.

To deal with the static characteristic of the Reservoir Sampling, Al-Kateb et al. [2007] proposed a dynamic adjustment of the reservoir size while sampling is still in progress. They focused on two main factors: the reservoir size and the sample uniformity. The reservoir size is adjusted based on runtime data characteristics or application behavior. While they address the problem of the reservoir size, a new enlarged reservoir may burst memory. Furthermore, adjusting the size of the reservoir while sampling leads to an additional cost corresponding to (I) the analysis of data stream, (II) the detection of changes in data characteristics, and (III) the computation of a new reservoir size. Recently, the

[1]Algorithms Every Data Scientist Should Know: Reservoir Sampling
`http://blog.cloudera.com/blog/2013/04/hadoop-stratified-randosampling-algorithm`
[2]Reservoir Sampling - Sampling from a stream of elements
`http://gregable.com/2007/10/reservoir-sampling.html`

same authors proposed a new adaptive stratified reservoir sampling over multiple sub-streams, but their algorithm still runs with fixed-size reservoir [Al-Kateb and Lee 2014].

Some works focus on the frequency of items to perform sampling of items at any time from the stream [Manku and Motwani 2002; Gibbons 2001]. For instance, the Hybrid-Streaming [Guo et al. 2011] considers the frequency of items on multiple data streams, maintaining approximate histograms for all data streams coming from different report sources. Frequency summaries are gathered by the algorithms and stored in internal structures consuming extra memory space.

Other approaches present moving/sliding windows over data streams, where timeline is meaningful [Gemulla and Lehner 2008; Babcock et al. 2002; Braverman et al. 2009]. These approaches consider that outdated data is expired in a time interval t (timestamp-based sample) or a group of the n most recent elements (sequence-based sample). A sample of size k is created by replacing the most recent elements in a window. However, they require defining a fixed-size sample (k) and extra information (e.g. t or n), leading to use $O(k)$ memory space and $O(k\ log\ n)$ for bursty windows [Braverman et al. 2009].

Other works use different strategies based on stratified sampling and resampling techniques. For instance, Levin and Kanza [2014] designed a distributed MapReduce algorithm based on the stratified sample technique. This algorithm aims to create multi-survey stratified sampling over on-line social networks, considering specific constraints and costs to share individuals among surveys. Elements are selected from strata created by Map and Reduce functions. However, the algorithm requires the population size from which the stratum is extracted. Another approach is the EARL [Laptev et al. 2012] framework to run queries on samples constrained by user thresholds in order to reduce response time. EARL creates samples following an error threshold. If the current sample achieves a high error, then a new sample is created with an increased sample size (resampling).

Compared to the described approaches, our main contribution lies in creating a new data sampling algorithm that does not require of a fixed-size reservoir while solving the acute problem of memory space.

3. RESERVOIR SAMPLING ALGORITHM

This section presents the Reservoir Sampling algorithm, as it represents the basis of KSample. The Reservoir Algorithm, initially known as Algorithm R, was designed by Alan Waterman, and later categorized by Vitter [1985]. Our first contribution draws on the Reservoir Sampling algorithm to generalize the proof for a reservoir of any size.

The Reservoir Sampling algorithm uses a fixed-size reservoir to hold elements added by the sampling process. The reservoir size (denoted by k) is the sample size. But defining k is challenging, as it has to be representative without filling up the whole main memory. The Reservoir Sampling aims to process a stream of elements whose length is large and unknown, and build a reservoir that can fit in main memory.

The Reservoir Sampling algorithm is presented in Algorithm 1: It receives as input k and the data stream. Initially, the reservoir is assigned with the first k elements. Then the algorithm calculates the probability of the i^{th} element to be inserted into the reservoir starting from the $(k+1)^{th}$, which has a probability equal to $P(i^{th}) = \frac{k}{i}$. Next, a random number (denoted by $rand$) uniformly distributed between 0 and 1 is chosen. If $rand < P(i^{th})$, then the i^{th} element randomly replaces another one in the reservoir.

Let us sketch the execution of the algorithm assuming a reservoir sampling of size $k = 1$, and that all elements have the same probability to be chosen. Initially (in round R_1), the first element E_1 comes in with the probability to be sampled as $P(E_1)^{R_1} = 1$. At this point, E_1 is chosen and the length of the stream is 1. When the next element E_2 comes in, the algorithm should decide to keep

Algorithm 1: Reservoir Sample Algorithm

```
Input  : $k$, sample size
Input  : $stream$, data stream of undefined length
Output: $arraySample[k]$
for $i = 1 \to k$ do
    $arraySample[i] \leftarrow stream[i]$
$numElements \leftarrow k$
while $stream$ != $EOF$ do
    $numElements \leftarrow numElements + 1$
    $probability \leftarrow k/numElements$
    $rand \leftarrow Random(0,1)$
    if $rand < probability$ then
        $pos \leftarrow Random(1,k)$
        $arraySample[pos] \leftarrow stream[numElements]$
return $arraySample$
```

E_1 or to replace it by E_2. The probability to choose E_2 is $P(E_2) = \frac{1}{2}$, as at this point the stream length is 2. The probability to keep E_1 $(P(E_1)^{R_2})$ is equal to the probability to choose it in the last round multiplied by the probability of not choosing E_2, which is:

$$\begin{aligned} P(E_1)^{R_2} &= P(E_1)^{R_1} \times \overline{P(E_2)} \\ &= 1 \times (1 - P(E_2)) \\ &= 1 \times \left(1 - \tfrac{1}{2}\right) = 1 \times \tfrac{1}{2} = \tfrac{1}{2} \end{aligned}$$

Thus, the probabilities of E_1 and E_2 in the second round are the same, i.e., $\frac{1}{2}$.

Likewise in the next round (R_3), when E_3 comes in, the algorithm has to decide between the element chosen in R_2 and E_3. The probability to choose E_3 is $P(E_3) = \frac{1}{3}$, as the stream length at this point is 3. Now, the algorithm computes the probability to keep the element chosen in R_2. Let us consider that this element is E_1. Its probability in the current round R_3 is its probability in R_2 multiplied by the probability of not choosing E_3, as follows:

$$\begin{aligned} P(E_1)^{R_3} &= P(E_1)^{R_2} \times \overline{P(E_3)} \\ &= \tfrac{1}{2} \times (1 - P(E_3)) \\ &= \tfrac{1}{2} \times \left(1 - \tfrac{1}{3}\right) = \tfrac{1}{2} \times \tfrac{2}{3} = \tfrac{1}{3} \end{aligned}$$

So, the probabilities of E_1, E_2 and E_3 in R_3 are equal to $\frac{1}{3}$. By induction, in round R_n the probability of each element is $\frac{1}{N}$.

Proof 3.1. Reservoir Sampling's proof.

Let us prove that the probability of any element to be present in the reservoir after N rounds is $\frac{k}{N}$.

—*Base Case*: $N = k$.
The probability of the k^{st} elements is $P(k^{st}) = \frac{k}{N} = 1$.

—*Induction Hypothesis (I.H.)*:
Let us assume that the probability of the N^{st} elements in the round N^{th} is $P(N^{st}) = \frac{k}{N}$, N > k.

—*Induction Step*: round $N + 1$.
$P((N+1)^{th}) = \frac{k}{N+1}$ is the probability to choose the $(N+1)^{th}$ element to put into the reservoir, as the reservoir size is k and the stream length is $N + 1$ at this point.

The probability to remove an element (E_{remove}) already inserted in the reservoir is:

$$P(E_{remove}) = \text{(the probability to choose the } (N+1)^{th} \text{ element)}$$
$$\times \text{ (the probability to remove an element of the reservoir)}$$
$$P(E_{remove}) = P((N+1)^{th}) \times \frac{1}{k} = \frac{k}{N+1} \times \frac{1}{k} = \frac{1}{N+1}$$

Hence, the probability of any element kept in the reservoir is:

$$P(E_{keep}) = 1 - P(E_{remove}) = 1 - \tfrac{1}{N+1} = \tfrac{N}{N+1}$$

Thus, the probability of any previous element to be in the reservoir after the round $(N+1)^{th}$ is:

$$P(E_{being}) = \text{(probability of any previous element to be chosen in the last } N^{th} \text{ rounds)}$$
$$\times P(E_{keep})$$
$$P(E_{being}) = \frac{k}{N}\text{(by the I.H.)} \times \frac{N}{N+1} = \frac{k}{N+1}$$

□

4. KSAMPLE DYNAMIC RESERVOIR SAMPLING ALGORITHM

In this section, we present KSample, a sampling algorithm for the challenging task of defining the reservoir size on unbounded data streams. The key idea to drive the data sampling is to define an invariant: at any iteration of KSample the resulting sample invariably represents at least the target percentage of the stream regardless of its length. KSample receives as parameter a target percentage ($\rho \in\ ]0,1]$), and, whenever the stream or KSample stops, the resulting sample contains at least $P\%$ of elements from the stream (where $P = \rho \times 100$). Let us assume that $\rho = 0.1$ is received as input, then KSample will dynamically scale the reservoir to hold at least 10% of the input data stream.

KSample refines the reservoir sample technique to work with an undefined reservoir size. Hence, we define the concept of slot, which is a memory space where a single element is inserted into the reservoir. Several elements from a given data stream may compete for the same slot, but only statistically chosen elements are inserted. New slots can be dynamically created in the reservoir in order to ensure the invariant. When the slot reaches the target percentage, a new empty slot is created to be disputed by the newcomer elements. In time, slots that are no longer disputed are flushed away to secondary storage to save main memory space.

Algorithm 2 depicts KSample. It receives as input the target percentage to build the sample and the data stream. It starts with an empty reservoir. New slots are created in the sample reservoir, thus the reservoir bulks up dynamically. If the reservoir size is lower than ρ multiplied by the stream length (denoted by $sLength$), then a new $slot_i$ is created in the reservoir to keep newcomer elements. Then, KSample considers this slot as a reservoir of size equal to 1, and uses it to store newcomer elements until a new slot $slot_{i+1}$ is created (see Algorithm 1). Elements stored in old slots can be flushed away to secondary storage to save memory space.

Figure 1 illustrates KSample running with a target percentage equal to 30%, i.e., $\rho = 0.3$. Initially, when the first element ($E1$) comes in, $slot_1$ is created and $E1$ is inserted with probability $P(E_1) = 1$. In the second round, $E2$ replaces $E1$ in the reservoir with probability $P(E_2) = \frac{1}{2}$, but no additional slots are required as the reservoir still adheres to the target percentage ($\geq 30\%$). In the third round, the newcomer element $E3$ does not replace $E2$ and is discarded with probability $P(E_3) = \frac{1}{3}$. When $E4$ comes in the fourth round, KSample requires creating $slot_2$, otherwise the reservoir would not contain at least 30% of the population (from this moment $slot_1$ can be flushed away). Thus, $E4$ is inserted into $slot_2$. The algorithm loops until the stream ends.

Algorithm 2: KSample Algorithm

```
Input  : $\rho$, percentage for sampling ($\rho \in ]0, 1]$)
Input  : $stream$, data stream of undefined length
Output: $reservoir[\,]$
$sLength \leftarrow 0$
$slotRound \leftarrow 0$
while $stream$ != $EOF$ do
    $sLength^{++}$
    if $reservoir.size() < (\rho \times sLength)$ then
        $reservoir.flush()$
        $reservoir.newSlot()$
        $slotRound \leftarrow 0$
    $slotRound^{++}$
    $probability \leftarrow \frac{1}{slotRound}$
    $rand \leftarrow Random(0, 1)$
    if $rand \leq probability$ then
        $reservoir.currentSlot \leftarrow stream[sLength]$
return $reservoir$
```

Although the newest element seems to have the least probability to be inserted in the slot, the probability of old elements also change according to the arrival of the newest one, which is a property inherited from the Reservoir Sampling algorithm. For example when E_3 arrives in Figure 1, the probability of E_1 and E_2 to keep in the reservoir is the same than inserting E_3, i.e., $P(E_{1,2})$ = (the probability of E_1 and E_2 before the arrival of E_3) multiplied by (the probability of not choosing E_3), that is $\frac{1}{2} \times (1 - \frac{1}{3}) = \frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$. The cumulative probability of the Reservoir Sampling is shown by the induction in Section 3. It means that the probability of old items to be kept in the reservoir equals

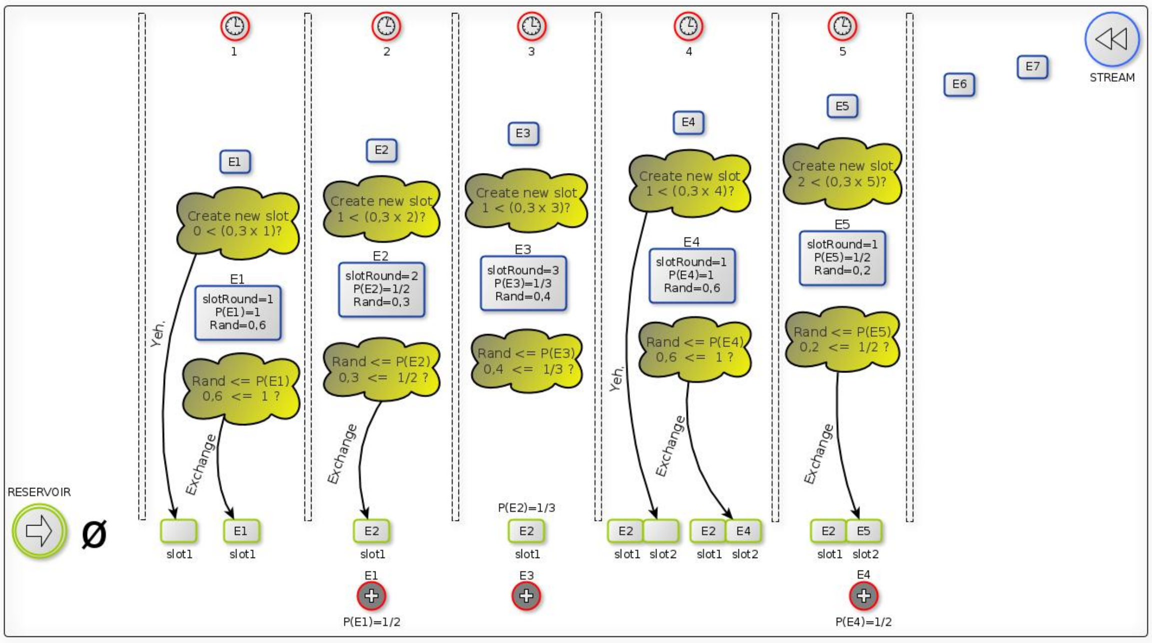
1
2
3
4
5
STREAM
E7
E6
E1
Create new slot 0 < (0,3 x 1)?
E1 slotRound=1 P(E1)=1 Rand=0,6
Rand <= P(E1) 0,6 <= 1 ?
Yeh.
Exchange
E2
Create new slot 1 < (0,3 x 2)?
E2 slotRound=2 P(E2)=1/2 Rand=0,3
Rand <= P(E2) 0,3 <= 1/2 ?
Exchange
E3
Create new slot 1 < (0,3 x 3)?
E3 slotRound=3 P(E3)=1/3 Rand=0,4
Rand <= P(E3) 0,4 <= 1/3 ?
E4
Create new slot 1 < (0,3 x 4)?
E4 slotRound=1 P(E4)=1 Rand=0,6
Rand <= P(E4) 0,6 <= 1 ?
Yeh.
Exchange
E5
Create new slot 2 < (0,3 x 5)?
E5 slotRound=1 P(E5)=1/2 Rand=0,2
Rand <= P(E5) 0,2 <= 1/2 ?
Exchange
RESERVOIR
Ø
slot1
E1 slot1
E2 slot1
E1
P(E1)=1/2
P(E2)=1/3
E2 slot1
E3
E2 slot1 slot2
E2 E4 slot1 slot2
E2 E5 slot1 slot2
E4
P(E4)=1/2

Fig. 1. The execution of KSample with $\rho = 0.3$ and with the conditional decisions to create a new slot and replace the elements inserted in the reservoir.

(the probability to choose them before the arrival of the newest one)^1 multiplied by (the probability of not choosing the newest one)^2.

Proof 4.1. Let us consider the following mathematical demonstration:

—*Notations:*
$Reservoir\ size = 1$ (for KSample one specific slot is a reservoir of size 1).
n: Slot round.
$Olds$: (the probability to choose old items before the arrival of the newest one)^1.
$NotNewest$: (the probability of not choosing the newest one)^2.

—$Olds$ = 1/n → that is the probability of the old items before the arrival of the newest one, according to the Reservoir Sampling's proof.

—$NotNewest$ = the complement of the probability to choose the newest one, i.e., 1 - (the probability to choose the newest one = $\frac{1}{n+1}$), then $NotNewest = 1 - \frac{1}{n+1}$.

—The probability of old items to be kept in the reservoir after the arrival of the newest one is: $Olds \times NotNewest = \frac{1}{n} \times 1 - \frac{1}{n+1} = \frac{1}{n} \times \frac{(n+1-1)}{n+1} = \frac{1}{n} \times \frac{n}{n+1} = \frac{1}{n+1}$.

—Thus, the probability of the old items to stay in the reservoir when the newest one arrives is the same, i.e., $\frac{1}{n+1}$, that is the probability of the newest one.

□

KSample ensures that the reservoir always holds at least the target percentage of elements (30% in the previous example) regardless of the length of the current stream. The creation of new slots is based on this invariant. That is, if KSample does not create a new slot, then the reservoir cannot hold at least P% of elements from the stream.

Proof 4.2. KSample's invariance proof by induction.

—*Notations*:
R: reservoir size.
ρ: target percentage of elements from the stream, $\rho \in]0,1]$.
P: $\rho \times 100$.
L: stream length.

—*Base Case*: When E_1 comes in.
$R = 0$, the algorithm determines whether to create a new slot by checking the condition $(R < (\rho \times L))$. As $\rho \in]0,1]$ and $L = 1$, then $(\rho \times L) \in]0,1]$. Consequently, $R < (\rho \times L)$ is true and the algorithm determines to create a new slot, thus the reservoir keeps E_1 and holds at least P% of the elements from the stream.

—*Induction Hypothesis (I.H.)*:
Suppose in the N^{th} step, after the arrival of the $(E_n)^{th}$ element, the reservoir holds at least P% of elements from the stream.

—*Induction Step*: step $(N+1)$.
We have to prove two cases:
(1) *Create a new slot*:
In this case the condition: $R < (\rho \times L)$ has to be true. By the I.H., in the last step N, the reservoir contains at least P% of the elements from the stream. Due to the incoming of E_{n+1}, KSample creates a new slot to hold this new element. Certainly, adding a new slot to the reservoir increases its size, thus keeping up with at least P% of elements of the stream.
(2) *Do not create a new slot*:
In this case the condition: $R \geq (\rho \times L)$ has to be true. This means that the reservoir holds P% or more elements from the stream.

Therefore, after the step $(N+1)$ the reservoir size, with at least $P\%$ of the elements of the stream, is kept in any round of KSample.

□

KSample processes any slot as a reservoir of size equal to 1. Moreover, elements competing for the same slot have the same probability to be inserted. One may argue that an exception arises when considering the last slot. For instance, if the stream length is odd and $\rho = 0.5$, the last element will be inserted in the last slot with a probability equal to 1. All other previous elements are inserted into the reservoir with probability equal to $\frac{1}{2}$. However, this exception cannot arise for sampling over unbounded data streams, because the stream is continuous.

5. EVALUATION

In this section, we present the empirical evaluation of KSample. We evaluate the efficiency of KSample against an adapted Reservoir Sampling on MapReduce. In addition, we show two practical use cases of KSample for boosting data processing of the Apache Hadoop system.

All experiments were carried out in a Hadoop[3] cluster composed of three machine nodes: one machine acts as the master node and the two others act as slave nodes. The machines run with Linux Mint 13 Maya SO, 3GB of ram memory, SATA Disk with at least 250GB, at a LAN of 100Mbs. The cluster was populated using the job *randomtextwriter* bundled with Apache Hadoop. This job generates 10GB of random text on each slave machine, totaling 20GB of data stored in the cluster. We used Hadoop version 0.20.203.0 in all experiments, as it is a requirement of the third party software (i.e., Starfish[4]). All experiments were run using the Hadoop streaming[5] in order to simulate the input data set as a stream of elements.

We implemented KSample in MapReduce for Hadoop to take advantage of its architecture for distributed computing and storage. We used each generated row as the atomic unit for data sampling. However, KSample is flexible enough to handle other atomic units, including files, blocks, bytes, and images. The MapReduce implementation of KSample is given in APPENDIX A.

Next, sections 5.1 and 5.2 show comparisons of KSample against a Random Sampling algorithm and an adaptation of the Reservoir Sampling algorithm on MapReduce. Sections 5.3 and 5.4 illustrate potential use cases of KSample for Hadoop data processing. In Section 5.3, KSample and the Starfish framework were used to tune Hadoop jobs. Section 5.4 shows the benefit of using KSample in the context of a typical Hadoop data analytics application.

5.1 KSample's Invariant

In this section, we compare KSample with a Random Sampling algorithm based on the Simple Random Sampling[6]. The Random Sampling algorithm is presented in APPENDIX C. It receives a percentage as parameter and the input population are text lines coming from the standard input. For each line a random float number is generated, a line is added to the survey if this random number is lower than the given percentage.

[3]Apache Hadoop a standardized MapReduce framework for Big Data processing.
`http://hadoop.apache.org/docs/r1.2.1/`
[4]Starfish - Self-tuning system on Hadoop. `http://www.cs.duke.edu/starfish/`
[5]Hadoop Streaming. `http://hadoop.apache.org/docs/r1.2.1/streaming.html`
[6]Simple Random Sampling. `http://www.ph.ucla.edu/epi/rapidsurveys/RScourse/RSbook_ch3.pdf`
`http://www.randomsampling.org/Simple-Random-Sampling.html`

	KSample		Random	
Round	Lines	Percentage	Lines	Percentage
1	106613	20,02%	106341	19.97%
2	106613	20,02%	106649	20.03%
3	106613	20,02%	106890	20.07%
4	106613	20,02%	106343	19.97%
5	106613	20,02%	105839	19.87%
6	106613	20,02%	106345	19.97%
7	106613	20,02%	106691	20.03%
8	106613	20,02%	106262	19.95%
9	106613	20,02%	107065	20.10%
10	106613	20,02%	106667	20.03%

Table I. KSample X Random Sampling.

We ran KSample and the Random Sampling algorithm 10 times with 20% as input parameter of a 20GB population. The size of this population corresponds to 532,414 lines of text. Table I shows the number of resulting line and the corresponding percentage for both algorithms.

Based on Table I, we observe that KSample does not have any deviation related to its invariant, i.e., all samples generated contain at least 20% of elements. KSample variance and standard deviation are both zero, which empirically corroborates with the KSample's invariant. On the other side, the Random Sampling algorithm generates different sizes of samples at each round with standard deviation of 335 number of lines. For applications constrained by accuracy, KSample is the best solution.

5.2 KSample Execution Time

In this section, we present a performance comparison in terms of execution time to generate samples between KSample and a MapReduce algorithm derived from the Reservoir Sampling [7]. In this derived algorithm, the Reservoir Sampling was adapted to support very large data sets, considering scenarios where the desired sample does not fit in memory. The Map function randomly fills buckets (that are allocated memory spaces to accommodate some items) by choosing items from the input data set. Then the Reduce function selects the k first items from these buckets.

We generate 9 samples using KSample with several percentage numbers as parameter (10%, 20%, ..., 90%) recording the execution times for each percentage. Meanwhile, we did the same with Reservoir Sampling[7], but provided the k parameter to generate the same quantity of samples. To compute 9 different sizes of reservoir, we ran another Hadoop job, called LineCounter, that counts the number of lines on the whole data set. Given the total number of lines computed by LineCounter (i.e., 532,414), we manually computed the number of lines that represent 10%, 20%, ..., 90% of the population. After, using each number of lines we ran the Reservoir Sampling to compute the samples.

Figure 2 depicts the execution time to generate the samples without considering the execution time of the LineCounter job. KSample was faster than the Reservoir Sampling in six data samplings (the samples: 10%, 20%, 30%, 40%, 50% and 60%). For large samples (70%, 80% and 90%) the Reservoir Sampling gets better execution time. But in this approach, users have to provide the reservoir size (i.e., the parameter k) to the algorithm, and finding a proper k may be an expensive task. In terms of efficiency, KSample shows better response times to generate samples smaller than 70% of the population.

[7]Reservoir Sampling in MapReduce.
`http://had00b.blogspot.ie/2013/07/random-subset-in-mapreduce.html`

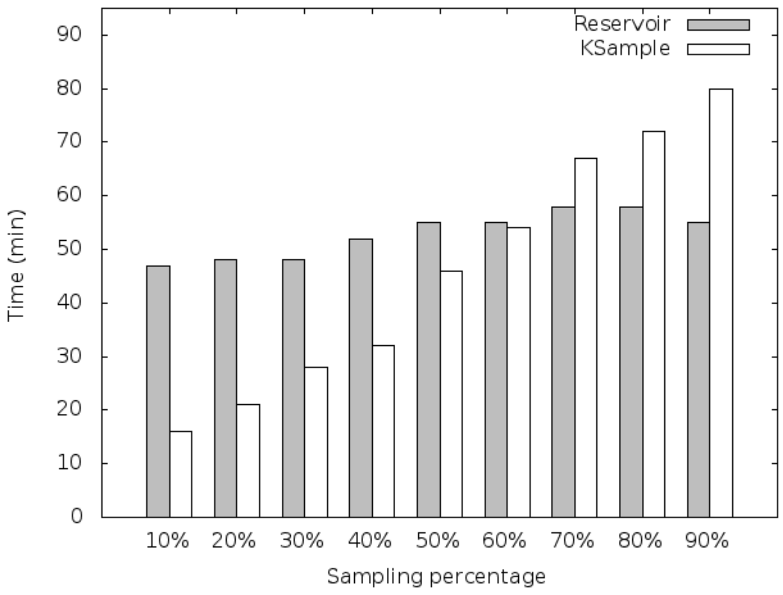

Fig. 2. Execution time to generate samples.

5.3 Using KSample to Tune Hadoop Jobs

In this section, we present a use case for KSample for boosting the costly process of tuning Hadoop jobs. Essentially, the cost is related to the large number of parameters involved. Indeed, every Hadoop job has more than 200 parameters to be properly tuned in order to obtain good performance, such as: memory allocation, I/O controllers and network timeouts [Filho et al. 2014]. Those parameters are bound to the available resources (e.g., input data, online machines and network bandwidth). In addition, Hadoop jobs are expected to process massive data sets, which can be the main barrier to finding a good configuration [Chen et al. 2012]. Thus, some tools have been developed to tune Hadoop jobs, and Starfish [Herodotou et al. 2011] is the state-of-the-art tool. Therefore, data sampling can be useful to generate testing data inputs for tuning Hadoop jobs.

We used Starfish v0.3.0 to find configurations for two jobs bundled with the Hadoop package: WordCount and Grep. First, we used the distributed KSample to generate 9 samples corresponding to 10%, 20%, ..., 90% of the data stored in the cluster. Second, in those samples we ran the Starfish's optimizer to get the configurations for both Hadoop jobs. We also ran the optimizer to get the configuration for the whole data set (100% of data). Finally, we configured the jobs with each computed configuration to run the jobs on the whole data set.

Tables II and III present the response times for 5 runs of each job. The columns labeled "10%" to "90%" show the execution time of the jobs tuned with the configurations provided by Starfish over the

#	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	Default
1	40:58	38:40	39:42	37:00	37:54	37:04	36:36	36:40	37:29	37:37	39:39
2	39:24	36:05	36:27	36:22	37:26	37:11	37:10	36:48	38:11	39:06	40:56
3	36:13	35:54	36:45	36:49	36:49	37:02	36:16	36:36	36:40	34:01	43:10
4	33:10	36:11	42:05	48:20	40:27	42:33	36:03	37:21	35:51	34:39	43:07
5	36:56	36:36	36:56	39:06	42:09	43:12	36:45	38:00	37:25	34:56	45:20
Avg	36:32	36:29	37:35	39:19	38:33	39:12	36:22	36:29	36:31	35:27	42:26

Table II. The execution times in minutes for KSample generating samples for the WordCount job.

#	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	Default
1	29:14	29:12	29:11	28:35	28:24	28:44	28:59	29:17	28:31	28:11	33:48
2	29:34	29:18	29:17	29:32	29:20	28:59	29:02	29:29	29:25	29:52	31:10
3	32:13	39:56	42:47	43:24	43:40	44:1	44:43	44:52	44:07	36:41	31:16
4	29:18	28:55	28:58	29:18	29:28	29:1	29:26	29:38	29:19	27:38	31:33
5	29:29	29:13	29:28	29:23	29:13	29:15	29:14	29:28	29:24	28:51	31:36
Avg	29:21	30:30	31:32	31:26	31:25	31:24	31:28	32:32	31:21	29:38	31:52

Table III. The execution times in minutes for KSample generating samples for the Grep job.

samples. The column labeled "100%" shows the execution time of the jobs tuned using the whole data set. The column labeled "Default" shows the execution time of jobs using the default configuration of Hadoop on the whole data set.

Figure 3 presents the average times in minutes with the best execution time for the WordCount provided by Starfish for the whole data set (100%). The second best performance was the sample of 20% of the data. However, getting this configuration takes 10 minutes and 9 seconds, and it takes 21 minutes to generate the sample. In total, it takes more than 31 minutes to tune the WordCount job. Whilst, the time needed to get the configuration from the whole data set was 56 minutes and 57 seconds. The difference of execution times is close to 1 minute, but to reach the configuration from a sample of 20% was around 25 minutes faster. If the WordCount job is executed once a day, after 25 days the configuration from the whole data set would be advantageous. If after 25 days the data set changed, thus it should be needed to compute a new job configuration.

For the Grep job the best performance was reached using the configuration from the sample of 10%. The second best time was reached using the configuration from the whole data set. But, the difference between them is 17 seconds. However, the configuration from the sample of 10% was computed in 5 minutes and 12 seconds, and the time to generate that sample was 16 minutes and 43 seconds. So the total time to get a configuration was 21 minutes and 55 seconds. Whilst, the time spent to get the configuration from the whole data set was 54 minutes and 43 seconds. Therefore, with a sample of 10% the execution time slightly decreases, but with half of the cost.

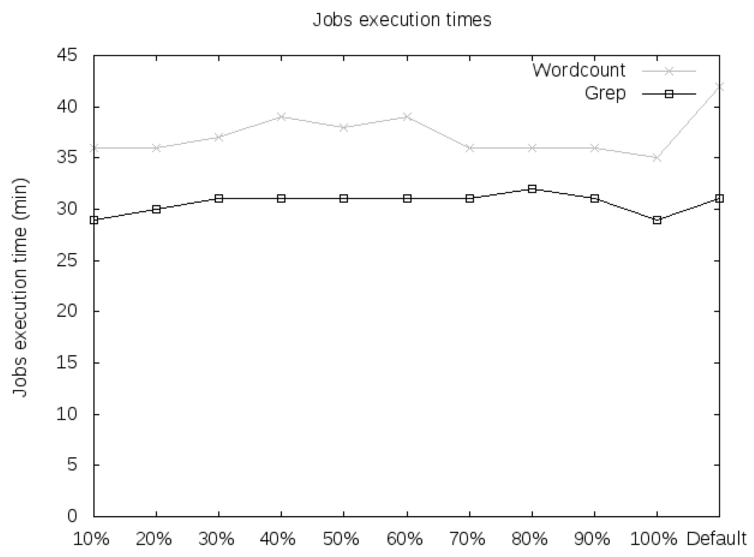

Fig. 3. MapReduce Jobs tuned by Starfish using samples provided by KSample.

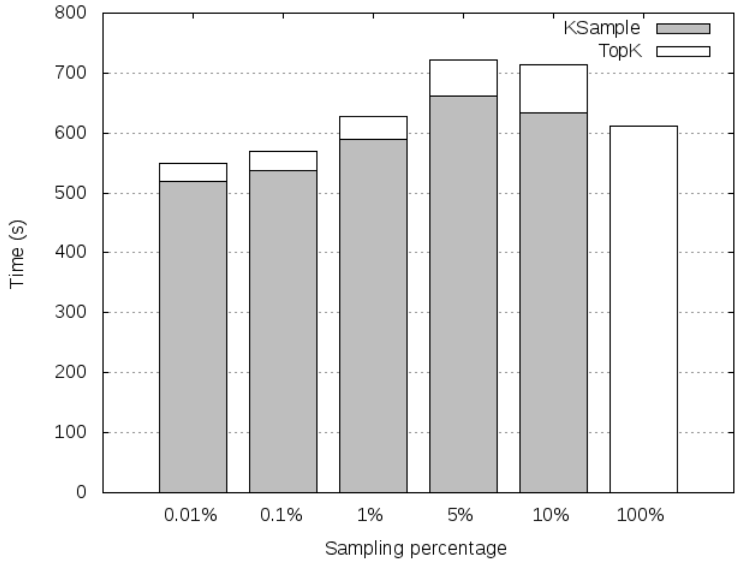

Fig. 4. Time to retrieve the top 10 products on samples.

5.4 Top-k Products

This section demonstrates the benefits of KSample when retrieving the top-k products in a sales list[8]. Top-k is a common task on continuous queries systems over data streams [Mouratidis et al. 2006]. KSample was applied to generate product lines as atomic units. The data set was generated synthetically with $300,000,000$ lines with 100 different products, each line represents the sale of a product, and contains *product id, product name, sales time stamp and amount of product sold.*

We implemented a MapReduce version of top-k for Hadoop to get the top-10 products from a data set of sold products. Top-k runs over each sample to seek the top-10 products. The Map function of the top-k job keeps apart the *product id* and the *amount of product sold* as the intermediate key/value. In the shuffle phase, top-k groups different amount of sales belonging to the same product, e.g., $\langle product\_id, [11, 20, 3, 5]\rangle$ (this process is similar to the WordCount job, but the *amount of sales* is assigned to the value instead of the occurrence of words). The Reduce function sums up the total of product sold, and in the end the output is ordered according to this number. Our implementation is given in APPENDIX D.

Figure 4 presents the time needed to generate the data samples (KSample bar), and the response time to retrieve the top-10 products on samples (TopK bar). Only the last bar (100%) does not have the time needed to run KSample, as it represents the whole data set. We used the *diff* program to check the top-10 products reached using those data samples against the top-10 products reached on the whole data set. We checked that the results were identical, i.e., 100% of precision. On the sample of 0,01% the top-k response time was 30 seconds. Whilst, the top-k response time on the whole data set was 612 seconds. If we consider just the top-k response time, KSample leads to an improvement of 20 times. In addition, KSample generated the sample in 520 seconds, totaling 540 seconds to run the top-k on a sample of 0,01%. This represents an improvement of 12% in the response time. In real-world analytics, the time to generate the sample can be diluted if reusing it to extract other information, such as the top-k products sold in a certain period.

[8]The variable k is different from the one introduced in Section 3.

As a concluding remark, computing the top-k products over a sample improved the response time in 20 times in contrast with the same computing over the whole data set, without losing precision.

6. CONCLUSION

We presented KSample, an efficient data sampling algorithm for data streams. A well-known technique for sampling data streams is the Reservoir Sampling, but it requires a fixed-size reservoir that corresponds to the resulting sample size. Defining the reservoir size is challenging, as big reservoirs may waste computing resources, whilst small ones may hold inaccurate samples. KSample differs from the state-of-the-art Reservoir Sampling algorithm, as it does not require to know the length of the stream or the size of the sample. The key idea of KSample is based on an invariant that keeps the percentage of the stream regardless of its length. KSample introduces the concept of mini-reservoirs to tackle the acute problem of memory space. The mini-reservoirs allow distributing the sampling over large machine setups.

KSample is also faster than the standard Reservoir Sampling to generate samples, which makes it appealing for data analytics tools over data streams. Empirical experiments show that KSample reached better performance to generate samples smaller than 70% of the population against the standard Reservoir Sampling. For instance, to generate a sampling of 10% from the data set, KSample is approximately 2.8 times faster, even ignoring the extra time to compute the reservoir sizes required by the Reservoir Sampling.

KSample showed good performance for data sampling in a distributed fashion on a cluster of Apache Hadoop machines and flexibility to process large and unbounded data streams. We obtained interesting results after exposing KSample to two practical use cases on Hadoop and continuous query systems. First, samples can be used to test performance tuning setups of Hadoop jobs. As presented in Section 5.3, the *Grep* job tuned over a sample of 10% reached a better response time than the whole data set counterpart, but with less than half of the cost. Second, KSample was applied to retrieve the top-k products from a stream of products containing 300, 000, 000 elements. The response time was 20 times better than executing top-k over the whole data set, whilst maintaining 100% accuracy. Future work includes seeking the best target percentage as the current KSample algorithm still needs it as an input from developers.

ACKNOWLEDGMENT

This work has been partially funded by the Brazilian Innovation Agency (FINEP) and CNPq grant 441944/2014-0.

REFERENCES

Al-Kateb, M. and Lee, B. S. Adaptive Stratified Reservoir Sampling over Heterogeneous Data Streams. *Information Systems* 39 (0306-4379): 199–216, 2014.

Al-Kateb, M., Lee, B. S., and Wang, X. S. Adaptive-Size Reservoir Sampling over Data Streams. In *Proceedings of the International Conference on Scientific and Statistical Databases Management*. Washington, DC, USA, pp. 22–34, 2007.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, pp. 665–665, 2003.

Babcock, B., Datar, M., and Motwani, R. Sampling from a Moving Window over Streaming Data. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA, pp. 633–634, 2002.

Bonin, R., Marcacini, R. M., and Rezende, S. O. Unsupervised Instance Selection from Text Streams. *Journal of Information and Data Management* 5 (1): 114–123, 2014.

Boutsis, I., Kalogeraki, V., and Gunopulos, D. Efficient Event Detection by Exploiting Crowds. In *Proceedings of the ACM International Conference on Distributed Event-based Systems*. New York, NY, USA, pp. 123–134, 2013.

Braverman, V., Ostrovsky, R., and Zaniolo, C. Optimal Sampling from Sliding Windows. In *Proceedings of the ACM Symposium on Principles of Database Systems*. New York, NY, USA, pp. 147–156, 2009.

Chen, C., Yin, H., Yao, J., and Cui, B. TeRec: a temporal recommender system over tweet stream. *Proceedings of the VLDB Endowment* 6 (12): 1254–1257, 2013.

Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. NiagaraCQ: a scalable continuous query system for internet databases. *SIGMOD Record* 29 (2): 379–390, 2000.

Chen, Y., Alspaugh, S., and Katz, R. Interactive Analytical Processing in Big Data Systems: a cross-industry study of MapReduce workloads. *Proceedings of the VLDB Endowment* 5 (12): 1802–1813, 2012.

Cochran, W. G. The Estimation of Sample Size. In J. Wiley (Ed.), *Sampling Techniques*. John Wiley Sons, USA, pp. 72–86, 1977.

Diaz-Aviles, E., Drumond, L., Gantner, Z., Schmidt-Thieme, L., and Nejdl, W. What is Happening Right Now ... That Interests Me?: online topic discovery and recommendation in Twitter. In *Proceedings of the ACM International Conference on Information and Knowledge Management*. New York, NY, USA, pp. 1592–1596, 2012.

Filho, E. R. L., de Almeida, E. C., and Traon, Y. L. Intra-query Adaptivity for MapReduce Query Processing Systems. In *Proceedings of the 18th International Database Engineering & Applications Symposium*. New York, NY, USA, pp. 380–381, 2014.

Gama, J. *Knowledge Discovery from Data Streams*. CRC Press, Boca Raton, USA, 2010.

Gemulla, R. and Lehner, W. Sampling Time-based Sliding Windows in Bounded Space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, pp. 379–392, 2008.

Gibbons, P. B. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, CA, USA, pp. 541–550, 2001.

Guo, J., Zhang, P., Tan, J., and Guo, L. Mining Frequent Patterns Across Multiple Data Streams. In *Proceedings of the ACM International Conference on Information and Knowledge Management*. New York, NY, USA, pp. 2325–2328, 2011.

Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. Starfish: a self-tuning system for big data analytics. In *Conference on Innovative Data Systems Research*. California, USA, pp. 261–272, 2011.

Laptev, N., Zeng, K., and Zaniolo, C. Early Accurate Results for Advanced Analytics on MapReduce. *Proceedings of the VLDB Endowment* 5 (10): 1028–1039, 2012.

Levin, R. and Kanza, Y. Stratified-sampling over Social Networks Using Mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, pp. 863–874, 2014.

Madden, S. and Franklin, M. J. Fjording the Stream: an architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering, ICDE*. San Jose, CA, USA, pp. 555–566, 2002.

Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, pp. 49–60, 2002.

Manku, G. S. and Motwani, R. Approximate Frequency Counts over Data Streams. In *Proceedings of the International Conference on Very Large Data Bases*. Hong Kong, China, pp. 346–357, 2002.

Mouratidis, K., Bakiras, S., and Papadias, D. Continuous Monitoring of Top-k Queries over Sliding Windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, pp. 635–646, 2006.

Papini, J. A. J., de Amo, S., and Soares, A. K. S. FPSMining: a fast algorithm for mining user preferences in data streams. *Journal of Information and Data Management* 5 (1): 4–15, 2014a.

Papini, J. A. J., de Amo, S., and Soares, A. K. S. Strategies for Mining User Preferences in a Data Stream Setting. *Journal of Information and Data Management* 5 (1): 64–73, 2014b.

Srivastava, U. and Widom, J. Memory-limited Execution of Windowed Stream Joins. In *Proceedings of the International Conference on Very Large Data Bases*. Toronto, Canada, pp. 324–335, 2004.

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

APPENDIX A. KSAMPLE IN PYTHON

```python
#!/usr/bin/python

import sys, random
from reservoir import Reservoir

perc = float(sys.argv[1])
slength, slotRound = 0, 0
```

```
res = Reservoir()

for elem in sys.stdin:
    slength += 1
    if (res.size() < (perc * slength)):
        if slotRound != 0:
            res.flush()
        res.newSlot()
        slotRound = 0

    slotRound += 1
    prob = 1.0/slotRound
    r = random.random()
    if (r <= prob):
        res.setCurrentSlot(elem)

print '%s' % (res.getCurrentElement()),
```

The Map Python script imports the Reservoir class presented in APPENDIX B. This class was designed to support a reservoir of size 1. The target percentage number is passed as a parameter to the program. Hadoop provides an embedded package to perform streaming processing, which transforms files in the HDFS (Hadoop Distributed File System) into streaming to the standard input (stdin). Then the Python Map script reads each line from the stdin and runs KSample. The intermediate key/value pairs are sent before a new slot is create in order to avoid leak memory on Hadoop.

We decided to implement KSample in the map phase due to the number of files scanned in this phase. The sample is created while files are scanned. The Reduce phase simply writes the lines sampled on Map phase into the HDFS.

APPENDIX B. RESERVOIR PYTHON CLASS

```
class Reservoir:
    def __init__(self):
        self.currentSlot = 0
        self.currentElement = ''

    def newSlot(self):
        self.currentSlot += 1

    def setCurrentSlot(self, newElement):
        self.currentElement = newElement

    def getCurrentElement(self):
        return self.currentElement

    def size(self):
        return self.currentSlot

    def flush(self):
        print '%s' % (self.getCurrentElement()),
```

APPENDIX C. RANDOM SAMPLING IN PYTHON

```
#!/usr/bin/python

import sys, random

perc = float(sys.argv[1])

for elem in sys.stdin:
    r = random.random()
    if r < perc:
        print '%s' % (elem),
```

APPENDIX D. TOP-K MR JOB IN PYTHON

APPENDIX D.1 TOP-K MAP FUNCTION

```
#!/usr/bin/python
import sys

products={}
for line in sys.stdin:
    product=line.split('|')
    id=product[0]
    sold=int(product[3])
    if id not in products:
        products[id] = sold
    else:
        products[id] += sold

for k,v in products.items():
    print "%s|%s" % (k, v)
```

APPENDIX D.2 TOP-K REDUCE FUNCTION

```
#!/usr/bin/python
import sys, random

topK=[]
current_product = None
current_count=0

for elem in sys.stdin:
    elem=elem.rstrip()
    product, count = elem.split("|");

    try:
        count = int(count)
    except ValueError:
        continue

    if current_product == product:
        current_count += count
    else:
        if current_product:
            topK.append([int(current_product), int(current_count)])
        current_product = product
        current_count = count

if current_product == product:
    topK.append([int(current_product), int(current_count)])

topK.sort(key=lambda x: x[1], reverse=True)

for k,v in topK:
    print "%s|%s" % (k, v)
```