Online Appendix: Temporally-Biased Sampling Schemes for Online Model Management

BRIAN HENTSCHEL, Harvard University, USA
PETER J. HAAS, University of Massachusetts Amherst, USA
YUANYUAN TIAN, IBM Research – Almaden, USA

This online appendix contains supplementary material for the paper, "Temporally-Biased Sampling Schemes for Online Model Management", by Hentschel, Haas, and Tian.

ACM Reference Format:
Brian Hentschel, Peter J. Haas, and Yuanyuan Tian. 2018. Online Appendix: Temporally-Biased Sampling Schemes for Online Model Management. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2018), 6 pages. https://doi.org/0000001.0000001

1 ANALYSIS FOR SECTION 2

Bernoulli downsampling. We first show that downsampling using the binomial distribution, as in Algorithm 1, is statistically equivalent to simple sequential downsampling via Bernoulli coin flips. Consider a set S and a subset $S' \subseteq S$ with $|S| = n$ and $|S'| = k$ (with $k \leq n$). The probability of producing S' from S via n coin flips with retention probability p is $P_1(S') = p^k(1-p)^{n-k}$. Now consider the probability $P_2(S')$ of producing S' from S by first generating a binomial number M of items to retain and then uniformly selecting M specific items uniformly from S. The probability of selecting $M = k$ items is $\binom{n}{k}p^k(1-p)^{n-k}$ and the probability of selecting the specific set S' of k elements, given that $M = k$, is $\binom{n}{k}^{-1}$. Thus the overall probability $P_2(S')$ is the product of these terms, which equals $P_1(S')$. Thus, for any subset S', both sampling schemes produce S' with the same probability, and hence the schemes are statistically identical.

Batch reservoir sampling. We now prove that Algorithm 2 does in fact produce uniform samples. As before, for $k \geq 1$, let $\mathcal{U}_k = \bigcup_{j=1}^{k} \mathcal{B}_j$ be the set of items arriving up through time t_k and set $W_k = |\mathcal{U}_k|$; we take $W_0 = 0$. Also write $B_k = |\mathcal{B}_k|$. Observe that $\{W_k\}_{k\geq 1}$ is nondecreasing and set $K = \min\{\, k \geq 1 : W_k > n \,\}$. We first show that S_k is a uniform sample from $\mathcal{U}_k$ for $k \in [1..K]$. For $k < K$, we have $W_{k-1} \leq n$ and $W_k = W_{k-1} + B_k \leq n$. In this case, $S_{k-1} = \mathcal{U}_{k-1}$ and $M = B_k$ with probability 1, since M is hypergeometric$(B_k + W_{k-1}, B_k, W_{k-1})$, so that $S_k = S_{k-1} \cup \mathcal{B}_k = \mathcal{U}_k$ and hence is trivially a uniform sample from $\mathcal{U}_k$. For $k = K$, we have that $W_{k-1} \leq n$ and $W_{k-1} + B_k > n$. Again, $S_{k-1} = \mathcal{U}_{k-1}$. Fix $m \in [n - W_{k-1}..n]$ and consider a set $S = B \cup R$, where $B \subseteq \mathcal{B}_k$ with $|B| = m$ and $R \subseteq \mathcal{U}_{k-1}$ with $|R| = n - m$. In this case, we will have $S_k = S$ if (i) $M = m$, where M is hypergeometric(n, B_k, W_{k-1}), (ii) the set of m items accepted into the sample is exactly the set

Authors' addresses: Brian Hentschel, Harvard University, Cambridge, Massachusetts, USA, bhentschel@g.harvard.edu; Peter J. Haas, University of Massachusetts Amherst, Amherst, Massachusetts, USA, phaas@cs.umass.edu; Yuanyuan Tian, IBM Research – Almaden, San Jose, California, USA, ytian@us.ibm.com.

0362-5915/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

B, and (iii) the set of $m - (n - W_{k-1})$ items chosen to be overwritten is exactly the set $\mathcal{U}_{k-1} - R$. Multiplying the probabilities of these three events together, we find that

$$\Pr[S_k = S] = \frac{\binom{B_k}{m}\binom{W_{k-1}}{n-m}}{\binom{W_{k-1}+B_k}{n}} \cdot \frac{1}{\binom{B_k}{m}} \cdot \frac{1}{\binom{W_{k-1}}{m-n+W_{k-1}}} = \frac{1}{\binom{W_{k-1}+B_k}{n}} = \frac{1}{\binom{W_k}{n}}.$$

Since m and S were chosen arbitrarily, and there are $\binom{W_k}{n}$ possible choices for S, it follows that S_K is a uniform sample of $\mathcal{U}_K$. We establish the desired result for $k > K$ by induction. Suppose that S_j is a uniform random sample of $\mathcal{U}_j$ for $j \le k-1$. Since $k > K$, we have that $W_{k-1} > n$. Consider a set $S = B \cup R$ as above, but with $m \in [0..\min(n, B_k)]$. Also let $\mathcal{E}$ denote the set of all subsets of $\mathcal{U}_{k-1} - R$ of size m. Thus $\mathcal{E}$ contains all possible sets of items that might be overwritten when accepting m items from $\mathcal{B}_k$ into the sample. Fix $E \in \mathcal{E}$ and consider the case where $S_{k-1} = E \cup R$. Then we will have $S_k = S$ if (i) $M = m$, where M is hypergeometric(n, B_k, W_{k-1}), (ii) the set of m items accepted into the sample is exactly the set B, and (iii) the set of m overwritten items is exactly the set E. Also, by induction, we have $\Pr[S_{k-1} = E \cup R] = 1/\binom{W_{k-1}}{n}$ for all $E \in \mathcal{E}$. Putting everything together, we have

$$\begin{aligned}
\Pr[S_k = S] &= \sum_{E \in \mathcal{E}} \Pr[S_{k-1} = E \cup R] \Pr[S_k = S \mid S_{k-1} = E \cup R] \\
&= \sum_{E \in \mathcal{E}} \frac{1}{\binom{W_{k-1}}{n}} \cdot \frac{\binom{B_k}{m}\binom{W_{k-1}}{n-m}}{\binom{W_{k-1}+B_k}{n}} \cdot \frac{1}{\binom{B_k}{m}} \cdot \frac{1}{\binom{n}{m}} = |\mathcal{E}| \frac{1}{\binom{W_{k-1}}{n}} \cdot \frac{\binom{B_k}{m}\binom{W_{k-1}}{n-m}}{\binom{W_{k-1}+B_k}{n}} \cdot \frac{1}{\binom{B_k}{m}} \cdot \frac{1}{\binom{n}{m}} \\
&= \binom{W_{k-1}-(n-m)}{m} \cdot \frac{1}{\binom{W_{k-1}}{n}} \cdot \frac{\binom{B_k}{m}\binom{W_{k-1}}{n-m}}{\binom{W_{k-1}+B_k}{n}} \cdot \frac{1}{\binom{B_k}{m}} \cdot \frac{1}{\binom{n}{m}} = \frac{1}{\binom{W_k}{n}}.
\end{aligned}$$

Again, since m and S are arbitrary, the desired result follows.

2 CHAO'S ALGORITHM

In this section, we provide pseudocode for a batch-oriented, time-decayed version of Chao's algorithm [1] for maintaining a weighted reservoir sample of n items, which we call B-Chao. Recall that the goal of time-biased sampling is to enforce the relationship

$$\Pr[x \in S_k]/\Pr[y \in S_k] = f(\alpha_{i,k})/f(\alpha_{j,k}) \tag{1}$$

for arbitrary batch arrival times $t_i \le t_j \le t_k$ and arbitrary items $x \in \mathcal{B}_i$ and $y \in \mathcal{B}_j$, where f is the decay function and $\alpha_{i,k} = t_k - t_i$ the age at time t_k of an item belonging to batch $\mathcal{B}_i$. For simplicity, we focus on exponential decay functions.

The pesudocode is given as Algorithm 1. In the algorithm, the function GET1(x, A) randomly chooses an item i in a set A, and then sets $x \leftarrow i$ and $A \leftarrow A \setminus \{x\}$. We explain the function NORMALIZE below.

Note that the sample size increases to n and remains there, regardless of the decay rate. During the initial period in which the sample size is less than n, arriving items are included with probability 1 (line 13); if more than one batch arrives before the sample fills up, then clearly the relative inclusion property in (1) will be violated since all items will appear with the same probability even though the later items should be more likely to appear. Put another way, the weights on the first n items are all forced to equal 1.

After the sample fills up, B-Chao encounters additional technical issues due to "overweight" items. In more detail, observe that $\mathrm{E}[|S|] = \sum_{i \in S} \pi_i$, where $\pi_i = P[i \in S]$. At any given moment we require that $\mathrm{E}[|S|] = \sum_{i \in S} \pi_i = n$. If we also require for each i that $\pi_i \propto w_i$, then we must have $\pi_i = n w_i / W$, where $W = \sum_{i \in S} w_i$. It is possible, however, that $w_i/W > 1/n$, and hence $\pi_i > 1$, for one or more items $i \in S$. Such items are called *overweight*. As in [1], B-Chao handles this by

ALGORITHM 1: Batched version of Chao's scheme (B-Chao)

```
λ: decay factor (≥ 0)
n: reservoir size
//Initialize
S ← ∅
W ← 0                                          //W = agg. weight of non-overweight items
V ← ∅                                          //V holds overweight items
A ← ∅                                          //A hold newly non-overweight items
//Process batches
for i ← 1, 2, ... do
    //update weights
    W ← e^{-λ} W
    for (z, w_z) ∈ V do w_z ← e^{-λ} w_z
    //Process items in batch
    for j ← 1, 2, ..., |B_t| do
        GET1(x, B_i)                           //get new item to process
        if |S| < n then                        //reservoir not full yet
            S ← S ∪ {x}; W ← W + 1
        else                                   //reservoir is full
            NORMALIZE(x, V, A, W, π_x)         //categorize items
            if UNIFORM() ≤ π_x then
                //accept x and choose victim to eject
                α = 0; y ← null; U ← UNIFORM()
                for (z, w_z) ∈ A do            //attempt to choose from A...
                    α ← α + (1 − ((n−|V|)w_z)/W)/π_x
                    if U ≤ α then
                        A ← A \ {(z, w_z)}; y ← z; break
                if y == null then GET1(y, S)   //... else remove victim from S
                if (x, 1) ∉ V then S ← S ∪ {x} //Add new item to sample if not overweight
            S ← S ∪ {z : (z, w_z) ∈ A}; A ← ∅  //if no longer overweight, stop tracking
    output S ∪ {z : (z, w_z) ∈ V}
```

retaining the most overweight item, say i, in the sample with probability 1. The algorithm then looks at the reduced sample of size $n-1$ and weight $W-w_i$, and identifies the item, say j, having the largest weight w_j. If item j is overweight in that the modified relative weight $w_j/(W-w_i)$ exceeds $1/(n-1)$, then it is included in the sample with probability 1 and the sample is again reduced. This process continues until there are no more overweight items, and can be viewed as a method for categorizing items as overweight or not, as well as normalizing the appearance probabilities to all be less than 1. The NORMALIZE function in Algorithm 2 carries out this procedure; Algorithm 2 gives the pseudocode. In Algorithm 2, the function GETMAX(V) returns the pair $(z, w_z) \in V$ having the maximum value of w_z and also sets $V \leftarrow V \setminus \{(z, w_z)\}$; ties are broken arbitrarily. An efficient implementation would represent V as a priority queue.

When overweight items are present, it is impossible to both maintain a sample size equal to n and to maintain the property in (1). Thus, as discussed in Section 2.1 of [1], the algorithm only enforces the relationship in (1) for items that are not overweight. When the decay rate λ is high, newly arriving items are typically overweight, and transform into non-overweight items over time due to the arrival of subsequent items. In this setting, recently-arrived items are overrepresented.

ALGORITHM 2: Normalization of appearance probabilities

```
x: newly arrived item (has weight = 1)
V: set of items that remain overweight (and their weights)
A: set of items that become non-overweight (and their weights)
W: aggregate weight of non-overweight items
π_x: inclusion probability for x
n: reservoir size
W ← W + 1 + Σ_{(z,w_z)∈V} w_z                          //agg. wt. of new & sample items
if n/W ≤ 1 then                                          //x is not overweight
    A ← V; V ← ∅                                      //no item is now overweight
    π_x ← n/W
else                                                     //x is overweight
    π_x ← 1; W ← W − 1
    D ← {(x, 1)}                                  //D = set of overweight items so far
    repeat
        (z, w_z) ← GetMax(V)
        if (n − |D|)w_z/W > 1 then                       //z remains overweight
            D ← D ∪ {(z, w_z)}; W ← W − w_z
        else                                          //z no longer overweight
            A ← A ∪ {(z, w_z)}
    until (n − |D|)w_z/W ≤ 1                          //first non-overweight item
    A ← A ∪ V; V ← D                                  //no more overweight items in V
```

The R-TBS algorithm, by allowing the sample size to decrease, avoids the overweight-item problem, and thus the violation of the relative inclusion property (1), as well as the complexity arising from the need to track overweight items and their individual weights (as is done in the pseudocode via V). We note that prior published descriptions of Chao's algorithm tend to mask the complexity and cost incurred by the handling of overweight items.

3 IMPLEMENTATION OF D-R-TBS ON SPARK

In this section we discuss aspects of our implementation that are specific to Spark. Spark is a natural platform for implementing D-R-TBS because it supports streaming, machine learning, and efficient distributed data processing, and is widely used. Efficient implementation is relatively straightforward for T-TBS but decidedly nontrivial for R-TBS because of both Spark's idiosyncrasies and the coordination needed between nodes.

3.1 Spark Overview

Spark is a general-purpose distributed processing framework based on a functional programming paradigm. Spark provides a distributed memory abstraction called a Resilient Distributed Dataset (RDD). An RDD is divided into partitions that are then distributed across the cluster for parallel processing. RDDs can either reside in the aggregate main memory of the cluster or in efficiently serialized disk blocks. An RDD is immutable and cannot be modified, but a new RDD can be constructed by transforming an existing RDD. Spark utilizes both lineage tracking and checkpointing of RDDs for fault tolerance. A Spark program consists of a single driver and many executors. The driver of a Spark program orchestrates the control flow of an application, while the executors perform operations on the RDDs, creating new RDDs.

3.2 Distributed Data Structures

We leverage Spark Streaming to ingest batches of arriving data, thereby supporting input sources such as HDFS, Kafka, Flume, and so on. Each incoming batch $\mathcal{B}_k$ is thus naturally stored as an RDD. We can store the reservoir using either a key-value store or a co-partitioned reservoir—see Section 5.2—but prefer using a co-partitioned reservoir because it has lower overhead (since incoming batch partitions align with local reservoir partitions). We would like use Spark's distributed fault-tolerant RDD data structure to implement the co-partitioned reservoir. A problem arises, however, if we try to store the reservoir as a vanilla RDD: because RDDs are immutable, the large numbers of reservoir inserts and deletes at each time point would trigger the constant creation of new RDDs, quickly saturating memory. We therefore augment the RDD with with the in-place update technique proposed by Xie, et al. [2]. The key idea is to share objects across different RDDs. In particular, we store the reservoir as an RDD, each partition of which contains only one object, a (mutable) vector containing the items in the corresponding reservoir partition. A new RDD created from an old RDD via a batch of inserts and deletes references the same vector objects as the old RDD. We keep the lineage of RDDs intact by notifying Spark of changes to old RDDs by calling the Unpersist function. In case of failure, old RDDs (with old samples) can be recovered from checkpoints, and Spark's recovery mechanism based on lineage will regenerate the sample at the point of failure.

3.3 Choosing Items to Delete and Insert

Section 5.3 has detailed the centralized and distributed decision mechanisms for choosing items to delete and insert. Here, we add some Spark-related details for centralized decisions.

All of the transient large data structures are stored as RDDs in Spark; these include the set of item locations for the insert items $\mathcal{Q}$, the set of retrieved insert items $\mathcal{S}$, and the set of item locations for the delete items $\mathcal{R}$. To ensure the co-partitioning of these RDDs with the incoming batch RDD—and the reservoir RDD when the co-partitioned reservoir is used—we use a customized partitioner. For the join operations between RDDs, we use by default Spark's standard repartition-based join. When RDDs are co-partitioned and co-located, however, we implement a customized join algorithm that performs only local joins on corresponding partitions.

3.4 Fault Tolerance of Distributed Implementations

We rely primarily on Spark's lineage tracking and checkpointing mechanisms to ensure the fault tolerance of our distributed algorithms. Spark Streaming's checkpointing mechanism is used to ensure the resiliency of the incoming batches. If the co-partitioned reservoir approach is used, we simply leverage Spark's built-in lineage and checkpointing mechanisms for the reservoir RDD. If the key-value store approach is used, then, because such stores are non-native to Spark, we need to do our own checkpointing, writing the reservoir content to the distributed file system; this adds more implementation overhead to the distributed algorithms. We also have to do our own checkpointing for any variables not stored in the foregoing distributed data structures, such as the current total weight and the current sample weight, for the distributed algorithms.

Finally, we distinguish between checkpointing the reservoir for fault tolerance and materializing the reservoir for the use of external ML applications. Because failure doesn't happen very often, checkpointing occurs much less frequently than the arrival of incoming batches. On the other hand, no matter how the reservoir is implemented, its content needs to be materialized in a consumable format after processing each incoming batch, thereby enabling an external ML application to access the sample for model retraining. Because the changes to the reservoir between subsequent incoming

batches are usually small, the system can write a small delta for each new batch, and write full snapshots periodically.

REFERENCES

[1] M. T. Chao. 1982. A general purpose unequal probability sampling plan. *Biometrika* (1982), 653–656.
[2] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J. Haas. 2015. Dynamic interaction graphs with probabilistic edge decay. In *ICDE*. 1143–1154.