Reservoir Sampling over Memory-Limited Stream Joins

Mohammed Al-Kateb Byung Suk Lee X. Sean Wang

Department of Computer Science
The University of Vermont
Burlington VT 05405, USA
{malkateb,bslee,xywang}@cs.uvm.edu

Abstract

In stream join processing with limited memory, uniform random sampling is useful for approximate query evaluation. In this paper, we address the problem of reservoir sampling over memory-limited stream joins. We present two sampling algorithms, Reservoir Join-Sampling (RJS) and Progressive Reservoir Join-Sampling (PRJS). RJS is designed straightforwardly by using a fixed-size reservoir sampling on a join-sample (i.e., random sample of a join output stream). Anytime the sample in the reservoir is used, RJS always gives a uniform random sample of the original join output stream. With limited memory, however, the available memory may not be large enough even for the join buffer, thereby severely limiting the reservoir size. PRJS alleviates this problem by increasing the reservoir size during the join-sampling [1]*. This increasing is possible since the memory requirement by the join-sampling algorithm decreases over time. A larger reservoir provides a closer representation of the original join output stream. However, it comes with a negative impact on the probability of the sample being uniform. Through experiments we examine the tradeoffs and compare the two algorithms in terms of the aggregation error on the reservoir sample.*

1. Introduction

Uniform random sampling has been known for its usefulness and efficiency for generating consistent and unbiased estimates of an underlying population. It has been extensively used in the database community for evaluating queries approximately [2] [10] [15] [16] [27] [36]. This approximate query evaluation may be necessary due to limited system resources like memory space or computation power. Two types of queries have been mainly considered: aggregation queries [2] and join queries [2] [10]. Between the two types, it is far more challenging for join queries because uniform random sampling of join inputs does not guarantee a uniform random sample of the join output [2] [10]. This paper concerns join queries.

In the context of data stream processing, Srivastava et al. [31] addressed that challenge with a focus on *streaming out* (without retaining) a uniform random sample of the result of a sliding-window join query with limited memory. There are, however, many data stream applications for which such a continuous streaming out is not practical.

One example is the applications that need a block of tuples (instead of a stream of tuples) to perform some statistical analysis like median, variance, etc. For these applications, there should be a way of retaining a uniform random sample of the join output stream.

Another example comes from the applications that collect results of join queries from wireless sensor networks using a mobile sink. Data collection applications have been extensively addressed in research literature [8] [9] [11] [19] [21] [29] [30] [32] [35]. In these applications, a mobile sink traverses the network and collects data from sensors. Thus, each sensor needs to retain a uniform random sample of the join output, instead of streaming out the sample tuples toward the sink.

A natural solution to keep a uniform random sample of the join output stream is to use *reservoir sampling* [25] [33]. Reservoir sampling selects a uniform random sample of a fixed size from an input stream of an unknown size. However, keeping a reservoir sample over stream joins is not trivial since streaming applications can be limited in memory size [5] [17] [26].

In this paper we address the problem of reservoir sampling over memory-limited stream joins. To our knowledge, this problem has not been addressed in any previous work. We present two algorithms that perform reservoir sampling on the join result: *Reservoir Join-Sampling (RJS)* algorithm and *Progressive Reservoir Join-Sampling (PRJS)* algorithm. The RJS algorithm is straightforward. We simply apply the conventional reservoir sampling to join-sample tuples streaming out from a join operator. The reservoir size is fixed. Naturally, the sample in the reservoir is always a

[1] Uniform random sampling on a join result is called *join-sampling* in [10].

uniform random sample of the join result. Therefore, RJS fits those applications that may use the sample in the reservoir at any time (e.g., continuous queries). This algorithm, however, may not accommodate a memory-limited situation in which the available memory may be too small even for storing tuples in the join buffer. In such a situation, it may be infeasible to allocate the already limited memory to a reservoir with an adequately large size.

The PRJS algorithm is designed to alleviate this problem by increasing the reservoir size during the sampling process. For this, we modify the conventional reservoir sampling technique to what we call the *progressive reservoir sampling* [3]. The key idea of PRJS is to exploit the property of reservoir sampling that the sampling probability keeps decreasing for each subsequent tuple. Based on this property, the memory required by the join buffer keeps decreasing during the join-sampling (details in Section 4.3). Therefore, PRJS releases the join buffer memory not needed anymore and allocates it to the reservoir.

Evidently, a larger reservoir sample represents the original join result more closely. It, however, comes at a cost in terms of the uniformity of the sample. Once the reservoir size is increased, the sample's uniformity is damaged. Besides, even after the enlarged reservoir is filled again with new tuples, the sample's uniformity is still not guaranteed, that is, the sample's uniformity confidence stays below 100% [3] (details in Section 2). There is a tradeoff that a larger increase of reservoir size leads to lower uniformity confidence after the reservoir is filled again. Therefore, PRJS is suitable for those applications that can be tolerant in terms of the uniformity of the sample. Specifically, it fits those applications that use the sample at a predetermined time (such as applications of data collection over wireless sensor networks). Given such a tradeoff, PRJS is designed so that it determines how much the reservoir can be increased given a sample-use time and a uniformity confidence threshold.

We have done extensive experiments to evaluate the two algorithms with respect to the two competing factors (size and uniformity of sample). We have also compared the two algorithms in terms of the aggregation error resulting from applying AVG on the join result. The experimental results confirm our understanding of the tradeoffs.

In this paper, we make the following contributions:

1. We identify a new problem of reservoir sampling on the output of a memory-limited stream join, and present two algorithms as solutions to the problem: one with a fixed reservoir size (called Reservoir Join-Sampling (RJS)) and the other with an increasing reservoir size (called Progressive Reservoir Join-Sampling (PRJS)).
2. For the PRJS algorithm we introduce an algorithm, called progressive reservoir sampling, and formally discuss a property that the uniformity of sample is not guaranteed if the reservoir size is increased in the middle of sampling.
3. We evaluate the two presented algorithms with respect to their performance tradeoffs between the size and uniformity of a reservoir sample as well as aggregation results on the reservoir sample.

The rest of the paper is organized as follows. Section 2 introduces our progressive reservoir sampling algorithm. Section 3 outlines the join-sampling processing model. Section 4 presents RJS and PRJS algorithms. Section 5 evaluates the two algorithms through experiments. Section 6 discusses related work. Section 7 concludes the paper and outlines future work.

2. Progressive Reservoir Sampling

In this section, we review the conventional reservoir sampling algorithm. Then, we conduct a theoretical study on the effects of increasing a reservoir size in the middle of sampling, and propose our progressive reservoir sampling algorithm. Results of this theoretical study and the origin of the proposed algorithm appear in [3].

2.1. Reservoir sampling

The conventional reservoir sampling [25] selects a uniform random sample of a fixed size, without replacement, from an input stream of an unknown size (see Algorithm 1).

Algorithm 1 *Conventional Reservoir Sampling*

```
1: input: $r$ {reservoir size}
2: $k = 0$
3: for each tuple arriving from the input stream do
4:     $k = k + 1$
5:     if $k \leq r$ then
6:         add the tuple to the reservoir
7:     else
8:         sample the tuple with the probability $\frac{r}{k}$ and replace a
           randomly selected tuple in the reservoir with the sampled tuple
9:     end if
10: end for
```

Initially, the algorithm places all tuples in the reservoir until the reservoir (of size r tuples) becomes full. After that, each k^{th} tuple is sampled with the probability $\frac{r}{k}$. A sampled tuple replaces a randomly selected tuple in the reservoir. This way, the reservoir always holds a uniform random sample of all the tuples seen from the beginning [25].

2.2. To increase the size of a reservoir

A sample is a uniform random sample if it is produced using a sampling scheme in which all statistically possible samples of the same size are equally likely to be selected.

Table 1. Notations used in this paper.

Symbol	Description
S_i	Data stream i ($i = 1, 2$)
λ_i	Rate of stream S_i
s_i	Tuple arriving in stream S_i
W_i	Sliding window on stream S_i
A	Join attribute (common to S_1 and S_2)
S_i-probe	Join tuple produced by $s_i \in W_i$
$n_i(s_i)$	Number of S_i-probe join tuples produced by a tuple $s_i \in S_i$ before it expires from W_i
$\mathbb{S}$	Sample in a reservoir
r	Initial reservoir size
δ	Increment of a reservoir size
k	Number of tuples seen so far in an input stream
l	Number of tuples that would be generated without join-sampling by the time the reservoir sample will be used (or collected)
RC	Reservoir refill confidence
ξ	Reservoir refill confidence threshold
UC	Uniformity confidence in a reservoir sample
ζ	Uniformity confidence threshold
m	Uniformity confidence recovery tuple count, i.e., number of tuples to be seen in an input stream of the progressive reservoir sampling until UC for the enlarged reservoir reaches ζ
x	Number of tuples to be selected from k after increasing the reservoir size
y	Number of tuples to be selected from m after increasing the reservoir size
p_1	Join sampling probability in the first phase of the algorithms *RJS* and *PRJS*
p_2	Reservoir sampling probability in the second phase of the algorithms *RJS* and *PRJS*

In this case, we say the uniformity confidence in the sampling algorithm equals 100%. In contrast, if some statistically possible samples cannot be selected using a certain sampling algorithm, then we say the uniformity confidence in the sampling algorithm is below 100%. Thus, we define uniformity confidence as follows.

$$\frac{\text{the number of different samples of the same size possible with the algorithm}}{\text{the number of different samples of the same size possible statistically}} \times 100 \qquad (1)$$

For reservoir sampling, the uniformity confidence in a reservoir sampling algorithm which produces a sample $\mathbb{S}$ of size r (denoted as $\mathbb{S}_{[r]}$) is defined as the probability that $\mathbb{S}_{[r]}$ is a uniform random sample of *all the tuples seen so far*. That is, if k tuples have been seen so far, then the uniformity confidence is 100% if and only if every statistically possible $\mathbb{S}_{[r]}$ has an equal probability to be selected from the k tuples. As we show below, if the reservoir size is increased from r to r+δ ($\delta > 0$), then some statistically possible $\mathbb{S}_{[r+\delta]}$'s cannot be selected.

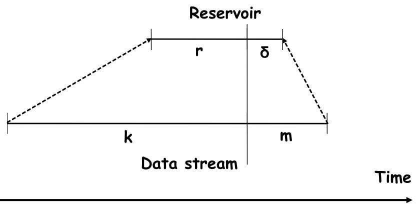


Figure 1. Increasing the reservoir size.

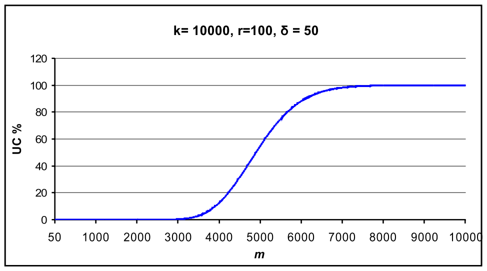


Figure 2. $UC(\mathbb{S}_{r+\delta})$ with respect to m (Equation 2).

Suppose the size of a reservoir is increased from r to r+δ ($\delta > 0$) immediately after the k^{th} tuple arrives (see Figure 1). Then, the reservoir has room for δ additional tuples. Clearly, there is no way to fill this room from sampling the k tuples as they have already passed by. We can only use incoming tuples to fill the room. The number of incoming tuples used to fill the enlarged reservoir is denoted as m and called the *uniformity confidence recovery tuple count*.

For the sake of better uniformity, we allow some of the r existing tuples to be evicted probabilistically and replaced by some of the incoming m tuples. In our work, we *randomly* pick the number of tuples evicted (or equivalently, the number of tuples retained). Clearly, the number of tuples that are retained, x, can be no more than r. Besides, x should not be less than $(r + \delta) - m$ if $m < r + \delta$ (because otherwise, with all m incoming tuples the enlarged reservoir cannot be refilled), and no less than 0 otherwise. Hence, we can have x tuples, where $x \in [\max\{0, (r + \delta) - m\}, r]$, from the k tuples and the other $r + \delta - x$ tuples from the m tuples. This eviction scheme allows for $\binom{k}{x}\binom{m}{r+\delta-x}$ different $\mathbb{S}_{[r+\delta]}$'s for each x in the range $[\max\{0, (r+\delta)-m\}, r]$. On the other hand, the number of different samples of size $r + \delta$ that should be statistically possible from sampling $k + m$ tuples is $\binom{k+m}{r+\delta}$. Hence, with the eviction in place, the uniformity confidence is expressed as follows:

$$UC(k, r, \delta, m) = \frac{\sum_{x=\max\{0,(r+\delta)-m\}}^{r} \binom{k}{x}\binom{m}{r+\delta-x}}{\binom{k+m}{r+\delta}} \times 100$$

where $m \geq \delta$. (2)

Examining this formula shows that the uniformity confidence increases monotonously and saturates as m increases. Figure 2 shows this pattern given one setting of k, r, and δ.

Note that the uniformity confidence *never* reaches 100%, as exemplified by Figure 3 which magnifies the uniformity confidence curve of Figure 2 for $m \geq 9000$.

The following theorem summarizes the uniformity confidence property of reservoir sampling in the event of increasing the reservoir size in the middle of sampling.

Theorem 1 *If the size of a reservoir is increased from r to $r + \delta$ ($\delta > 0$) while sampling from an input stream is in progress (after seeing more than r tuples), it is not possible to maintain the sample in the enlarged reservoir with a 100% uniformity confidence.*

Proof Let x be the number of tuples that can be selected in the enlarged reservoir (of size $r + \delta$) from the k tuples seen so far in the input stream. Then, the uniformity confidence is equal to 100% if and only if x can be any value in the range of $[0, r + \delta]$. However, x cannot be more than r since we have only r tuples from the k tuples seen so far. From this we conclude that the uniformity confidence cannot reach 100%.

2.3. Progressive reservoir sampling algorithm

Algorithm 2 *Progressive Reservoir Sampling*

```
Inputs: $r$ {reservoir size}
        $k$ {number of tuples seen so far}
        $\zeta$ {uniformity confidence threshold}
1: while true do
2:     while reservoir size does not increase do
3:         conventional reservoir sampling (Algorithm 1).
4:     end while
5:     Find the minimum value of $m$ (using Equation 2 with the
       current values of $k, r, \delta$) that causes the $UC$ to exceed $\zeta$.
6:     flip a biased coin to decide on the number, $x$, of tuples to
       retain among $r$ tuples already in the reservoir (Equation 3).
7:     randomly evict $r - x$ tuples from the reservoir.
8:     select $r + \delta - x$ tuples from the incoming $m$ tuples using
       conventional reservoir sampling (Algorithm 1).
9: end while
```

Based on the above discussion, our progressive reservoir sampling works as shown in Algorithm 2. As long as the size of the reservoir does not increase, it uses the conventional reservoir sampling to sample the input stream (Line 3). Once the reservoir size increases by δ, the algorithm computes the minimum value of m (using Equation 2) that causes the UC to exceed a given threshold (ζ) (Line 5). Then, the algorithm flips a biased coin to decide on the number of tuples (x) to retain among the r tuples already in the reservoir (Line 6). The probability of choosing the value x is defined as:

$$p(x) = \frac{\binom{k}{x}\binom{m}{r+\delta-x}}{\binom{k+m}{r+\delta}} \qquad (3)$$

where $\max\{0, (r + \delta) - m\} \leq x \leq r$.

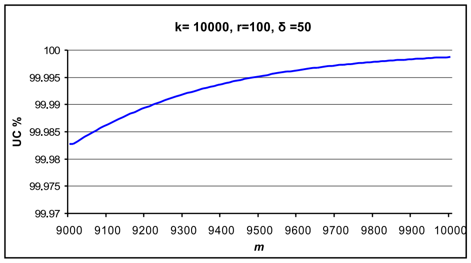


Figure 3. Figure 2 magnified for $m \geq 9000$.

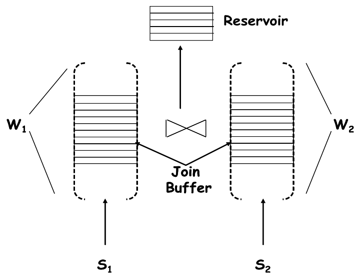


Figure 4. Join-sampling processing model.

After that, the algorithm randomly evicts $r - x$ tuples from the reservoir (Line 7) and refills the remaining reservoir space with $r + \delta - x$ tuples from the arriving m tuples using the conventional reservoir sampling (Line 8). Eventually, the algorithm continues sampling the input stream using the conventional reservoir sampling (Line 3) as if the sample in the enlarged reservoir were a uninform random sample of the $k + m$ tuples.

We will use these results to design the *Progressive Reservoir Join-Sampling* algorithm presented in Section 4.3.

3. Join-Sampling Processing Model

Figure 4 illustrates the processing model of join-sampling, i.e., uniform random sampling over a (two-way) join output stream. Tuples in the two sliding windows (W_1 and W_2) on the input data streams (S_1 and S_2) are equi-joined, i.e., with $S_1 \bowtie_{S_1.A=S_2.A} S_2$. ($A$ is the join attribute.) A sliding window is either time-based or tuple-based. If time-based, at any time τ a window W_i ($i = 1, 2$) contains the tuples of S_i whose timestamp t is in the range $[\tau - |W_i|, \tau]$, where $|W_i|$ denotes the size of W_i. If tuple-based, W_i contains the $|W_i|$ tuples that arrived on S_i most recently. The result of a join query is a stream of tuples $s_1||s_2$ (i.e., concatenation of s_1 and s_2) where s_1 is a tuple in S_1, s_2 is a tuple in S_2, and $s_1.A = s_2.A$.

We adopt the following notions from the join-sampling processing model in [31]. Every join-result tuple is classified as either an S_1-probe join tuple or an S_2-probe join tuple. When a new tuple s_1 arrives on S_1 and is joined with a tuple $s_2 \in W_2$, s_1 is said to produce an S_2-probe join tuple. An S_1-probe join tuple is defined symmetrically. A tuple $s_1 \in S_1$ may first produce S_2-probe join tuples when it arrives. Then, before it expires from W_1, it may produce S_1-probe join tuples with tuples newly arriving on S_2. $n_1(s_1)$ is a function which returns the number of S_1-probe join tuples produced by a tuple $s_1 \in S_1$ before it expires from W_1. $n_2(s_2)$ is defined symmetrically.

Tuples arrive in a data stream in a monotonically increasing order of the timestamp. In other words, there is no out of order arrival. The available memory M is limited, and insufficient for the join buffer to hold all tuples of the current sliding windows. We assume the initial reservoir size, r, is given. Determining the initial reservoir size, that is, allocating the memory M between the reservoir and the join buffer, is an interesting future research problem.

Under this join-sampling processing model, we have observed that as time passes we can lower memory requirement on the join buffer and transfer memory from the join buffer to the reservoir. This makes our results on progressive reservoir sampling applicable to this processing model.

4. Join-Sampling Algorithms

In this section, we first give an overview of the Uniform Join-Sampling (UNIFORM) algorithm [31], and then present our proposed algorithms: *Reservoir Join-Sampling (RJS)* and *Progressive Reservoir Join-Sampling (PRJS)*. UNIFORM is used as a building block in RJS and PRJS. Each of the proposed algorithms has two phases: join sampling phase and reservoir sampling phase. We denote the sampling probabilities used in the first phase as p_1 and the sampling probability used in the second phase as p_2.

4.1. Uniform join-sampling

Algorithm 3 *Uniform Join-Sampling (UNIFORM)*

```
1: for each $s_2$ in $W_2$ where $s_2.A = s_1.A$ do
2:     $s_2.num = s_2.num + 1$
3:     if $s_2.num = s_2.next$ then
4:         output $s_1 \| s_2$
5:         decide on the next $s_1$ to join with $s_2$
6:     end if
7: end for
8: pick $X \sim G(p_1)$ {geometric distribution}
9: $s_1.next = s_1.num + X$
10: if $s_1.next > n_1(s_1)$ then
11:     discard $s_1$
12: end if
```

UNIFORM [31] streams out a uniform random sample of the result of a sliding-window join query in a memory-limited stream environment. Algorithm 3 outlines the steps of the algorithm for one-way join from S_2 to S_1. (Join in the opposite, from S_1 to S_2, is symmetric.) The basic assumption of the algorithm is that $n_i(s_i)$ (see Table 1) ($i = 1, 2$) is known. The algorithm works with two prediction models that provide $n_i(s_i)$: frequency-based model and age-based model. Frequency-based model assumes that, given a domain D of the join attribute A, for each value $v \in D$ a fixed fraction $f_1(v)$ of the tuples arriving on S_1 and a fixed fraction $f_2(v)$ of the tuples arriving on S_2 have value v of the attribute A. Age-based model assumes that for a tuple $s_1 \in S_1$ the S_1-probe join tuples produced by s_1 satisfies the conditions that (1) the number of S_1-probe join tuples produced by s_1 is a constant independent of s_1 and (2) out of the $n_1(s_1)$ S_1-probe join tuples of s_1, a certain number of tuples is produced when s_1 is between the age $g-1$ and g. These definitions are symmetric for a tuple $s_2 \in S_2$. The choice of a predictive model is irrelevant to our work; thus, without loss of generality, we use the frequency-based model in the rest of the paper.

For the frequency-based model, $n_1(s_1) = \lambda_2 \times W_1 \times f_2(s_1.A)$. The join sampling probability p_1 is computed by first obtaining the expected memory usage (i.e., the expected number of tuples retained in the join buffer) in terms of p_1 and, then, equate this to the amount of memory available for performing the join and solving it for p_1. The expected memory usage of W_1 thus obtained as [31]:

$$\lambda_1 W_1 \sum_{v \in D} f_1(v)\left(1 - \frac{(1-p_1)(1-(1-p_1)^{\lambda_2 W_1 f_2(v)})}{p_1 \lambda_2 W_1 f_2(v)}\right) \tag{4}$$

A symmetric expression holds for the expected memory usage of W_2, assuming the same sampling probability p_1 for the S_2-probe join tuples. That is,

$$\lambda_2 W_2 \sum_{v \in D} f_2(v)\left(1 - \frac{(1-p_1)(1-(1-p_1)^{\lambda_1 W_2 f_1(v)})}{p_1 \lambda_1 W_2 f_1(v)}\right) \tag{5}$$

Summation of these two expressions gives the total memory usage for $W_1 \bowtie W_2$.

Given p_1, the algorithm proceeds as shown in Algorithm 3. When a tuple s_1 arrives on S_1, UNIFORM looks for every $s_2 \in W_2$ such that $s_1.A = s_2.A$ (Line 1). Then, it outputs $s_1 \| s_2$ if this s_1 is the tuple s_2 is waiting for to output the next sample tuple (Line 4), and then decides on the next s_1 for s_2 (Line 5). Moreover, once s_1 arrives on S_1, UNIFORM flips a coin with bias p_1 to decide the next S_1-probe join tuple of s_1 (Line 8-9). To do that, UNIFORM picks X at random from the geometric distribution with parameter p_1, $G(p_1)$. If all remaining $S1$-probe join tuples of s_1 are rejected in the coin flips, s_1 is discarded (Line 10-12).

4.2. Reservoir join-sampling

RJS applies the conventional reservoir sampling on the output of UNIFORM. Thus, it uses a fixed size reservoir, and always holds a uniform random sample in the reservoir. Algorithm 4 outlines the steps of RJS. Given a fixed reservoir of size r, the first r join-sample tuples produced by UNIFORM are directly placed in the reservoir (Line 3-4). After that, each join-sample tuple is re-sampled using reservoir sampling with a probability p_2 so that a $p_1 \times p_2 = r/(k+1)$, that is, $p_2 = (r/(k+1)/p_1$ (Line 6).

Algorithm 4 *Reservoir Join-Sampling (RJS)*

```
$k = 0$
for each tuple output by UNIFORM do
    if $k \leq r$ then
        add the tuple to the reservoir
    else
        sample the tuple with the probability
        $p_2 = (r/(k+1))/p_1$
    end if
    $k = k + (1/p_1)$
end for
```

Algorithm 5 *Progressive Reservoir Join-Sampling (PRJS)*

```
$k = 0$
{Initially, the memory utilization of the join buffer is 100%.}
while the memory utilization of the join buffer does not decrease do
    for each tuple output by UNIFORM do
        if $k \leq r$ then
            add the tuple to the reservoir
        else
            sample the tuple with a probability $p_2 = (r/(k+1))/p_1$
        end if
        $k = k + (1/p_1)$
        set $p_1 = r/(k+1)$ {for the next incoming tuple}
        recompute the memory utilization of the join buffer using Equations 4 and 5
    end for
end while
while $(RC(m) \geq \xi)$
      and $(UC(\mathbb{S}_{r+\delta}) \geq \zeta)$
      and $(m \geq (x+y) - (p_1(k+1)))$ do
    decrease $p_1$ by a specified constant value
    recompute the memory utilization of the join buffer using Equations 4 and 5
    increase $\delta$ by the amount of unused memory
end while
while $(RC(m) < \xi)$
      or $(UC(\mathbb{S}_{r+\delta}) < \zeta)$
      or $(m < (x+y) - (p_1(k+1)))$ do
    $\delta = \delta - 1$
    if $\delta = 0$ then
        return
    end if
end while
release $\delta$ memory units from the join buffer and allocate the released memory to the reservoir.
flip a biased coin to decide on $x$ and $y$ (Equation 3)
randomly evict $r - x$ sample tuples from the reservoir
get $y$ sample tuples out of $m$ using Algorithm 1
continue sampling the input stream using Algorithm 1
```

k is an index of the *original* join output tuples that would be generated from the join. Since join-sampling selects only a portion of them, the value of k should be *estimated*. This estimation is done as follows. When a tuple s_1 produces an S_2-probe join tuple, $1/p_1$tuples would be generated on average from the exact join since the algorithm samples a join result tuple with probability p_1. Therefore, $k = k + (1/p_1)$ (Line 8). This estimation process is symmetric for S_1-probe join tuples.

4.3. Progressive reservoir join-sampling

The key idea behind PRJS is to utilize the property of reservoir sampling that the sampling probability keeps decreasing for each subsequent tuple (see Algorithm 1). This property allows to release memory from the join buffer and transfer it to the reservoir. However, the benefit of increasing a reservoir size comes at a cost on the uniformity of the sample, as stated in Theorem 1.

PRJS needs to know the values of m (uniformity confidence recovery tuple count) and ζ (uniformity confidence threshold). Given the time left until the sample-use (or collection) time (denoted as T), the number of tuples (denoted as l) that would be generated during T if there were no join-sampling is computed as follows:

$$l = \mathrm{T}\lambda_1\lambda_2(W_1 + W_2)\sum_{v\in D} f_1(v)f_2(v) \tag{6}$$

PRJS proceeds in two phases: join-sampling phase and reservoir-sampling phase. Tuples in the join-sampling phase are sampled with a probability p_1. Therefore, the expected number of tuples to be seen by the reservoir-sampling phase (m) is:

$$m = lp_1 \tag{7}$$

Given m and ζ, PRJS works as shown in Algorithm 5. There are four main steps in the algorithm. The first step (Lines 2-13) concerns the memory transfer mechanism of PRJS. Initially there is no memory that can be transferred, since the memory utilization of the join buffer is 100%. As long as this is the case, PRJS works in the same way as RJS does (Algorithm 4) except that, for each new tuple s_i arriving on join input stream S_i, p_1 is decreased to $r/(k+1)$ and, accordingly, PRJS recomputes memory utilization of the join buffer. The reason for assigning this particular value to p_1 is that all S_i-*probe* join tuples to be produced by s_i while $s_i \in W_i$ should be sampled with effectively a probability of no more than $r/(k+1)$. In other words, this is the smallest possible value that can be assigned to p_1. PRJS keeps decreasing p_1 and recomputing the memory utilization until it finds that some memory can be released from the join buffer and transferred to the reservoir.

In the second step (Lines 14-18) and in the third step (Lines 19-24), PRJS finds the largest amount of memory (δ) that can be released from the join buffer and transferred to the reservoir, considering the following constraints:

- *Refill confidence:* The refill confidence, RC, is defined as the probability that m is at least the same as the enlarged reservoir size. That is given r and δ:
$$RC(m) = probability(m >= r + \delta) \qquad (8)$$
Unlike the progressive reservoir sampling (Section 2), PRJS cannot guarantee that the enlarged reservoir will be filled out of m tuples since m is only an *expected* number of tuples on the outcome of the join-sampling phase (see Equation 7). That is, the value of m is an expected value rather than an exact value. This means that actual value of m may be less than $r + \delta$, and this implies that $\delta \leq y \leq \min(m, r + \delta)$. ($y$ is the number of tuples to be selected from the m tuples). Therefore, the algorithm has to make sure that y falls in that range with a confidence no less than a given threshold ξ.
- *Uniformity confidence:* $UC \geq \zeta$. (See Equation 2.) That is, the uniformity confidence should be no less than ζ after the enlarged reservoir is filled.
- *Uniformity-recovery tuple count:* $m \geq (x + y) - (p_1(k + 1))$. The rationale for this constraint is as follows. PRJS assumes the reservoir sample (of $x + y$ tuples) will be used (or collected) after it will have seen m tuples. But if the sample-use does not happen, then it will have to continue with the conventional reservoir sampling on the join-sample tuples as if the sample in the reservoir were a uniform random sample of all join result tuples seen so far. In this case, $(x + y)/((k + (m/p_1)) + 1) \leq p_1$. Hence, $m \geq (x + y) - (p_1(k + 1))$.

If all these three constraints are satisfied, then in the second step PRJS keeps decreasing p_1 and increasing δ until one or more of them are not satisfied anymore. The more p_1 is decreased, the larger δ can be. Therefore, PRJS finds the smallest possible p_1 that makes the three constraints satisfied. This ensures to find the largest possible memory (δ) to be transferred to the reservoir.

When PRJS enters the third step, δ has been set too large to satisfy one or more of the three constraint. So, PRJS decreases δ until the constraints are satisfied or δ becomes 0. The latter case means that the reservoir size cannot be increased.

Once δ (> 0) is determined, in the fourth step (Line 25-29) PRJS releases δ memory units from the join buffer and allocates the released memory to the reservoir. Then, PRJS works in the same way as in the progressive reservoir sampling (see Lines 6-8 of Algorithm 2) to refill the reservoir.

5. Performance Evaluations

As mentioned in the Introduction, there is a tradeoff between the two presented algorithms: *Reservoir Join-Sampling (RJS)* and *Progressive Reservoir Join-Sampling (PRJS)*. Thus, in our evaluation we aim to compare these two algorithms in terms of the two traded factors: the achieved reservoir sample size and the achieved (recovered) uniformity of the sample. In addition, we conduct another set of experiments to put our evaluations in the database context. Specifically, we do an aggregation (AVG) on the reservoir sample, and compare the aggregation errors between the two algorithms.

The experimental results confirm the following:

- *Size of reservoir sample:* Regardless of the initial reservoir size, PRJS eventually results in a reservoir larger than the fixed-size reservoir of RJS.
- *Uniformity of reservoir sample:* Naturally, RJS's sample uniformity is always no lower than PRJS's sample uniformity. For PRJS, the uniformity is degraded when the reservoir size is increased but starts recovering promptly and approaches toward 100% as additional join-sample tuples are generated.
- *Aggregation on a reservoir sample:* For all the experimental settings used, we have observed from the results of aggregation errors on the reservoir sample that the benefit of gaining reservoir size is larger than the cost of losing sample uniformity. Naturally, PRJS achieves smaller aggregation errors than RJS unless the initial reservoir size is too large for PRJS to have room for increasing the size.

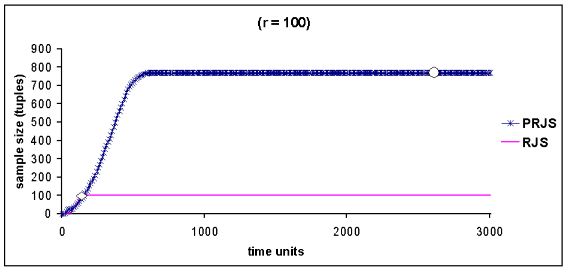
5.1. Setup

Algorithm setup: Both window sizes (W_1 and W_2) are set to 500 time units, and the two stream rates (λ_1 and λ_2) are set to 1 tuple per time unit and 5 tuples per time unit, respectively. These settings of sliding window sizes and stream rates are the same as those used in UNIFORM [31] which provides a basis of RJS and PRJS. Memory allocated to join buffer is 50% of the memory required for an exact result. The initial size of reservoir is 100 (i.e., $r = 100$ tuples) which represents 6% of the total available memory. We set both the uniformity confidence threshold ζ and the refill confidence threshold ξ to 0.90. We believe this value is sufficiently large to constrain the increase of reservoir size in PRJS. Unless stated otherwise, the results we report are obtained as an average of the results of 50 runs.

Data streams setup We have generated stream data sets each of which contains tuples amounting to 10000 time units. Values of join attribute in the input stream tuples are generated assuming the frequency-based model as indicated in Section 4. The values are drawn from a normal distribution with mean $\mu = 1000$ and variance $\sigma^2 = 1000$. Values of aggregate attribute are drawn from a normal distribution with mean $\mu = 1000$ and variance $\sigma^2 = 10000$.

5.2. Reservoir sample size

The objective of this experiment is to observe how the size of a sample in the reservoir changes size over time. Figure 5 shows the average sample size over time, at the interval of 10 time units, for both PRJS and RJS. For PRJS, the



Reservoir size increase time is marked with a diamond and sample-use time is marked with a circle.

Figure 5. Average sample size over time.

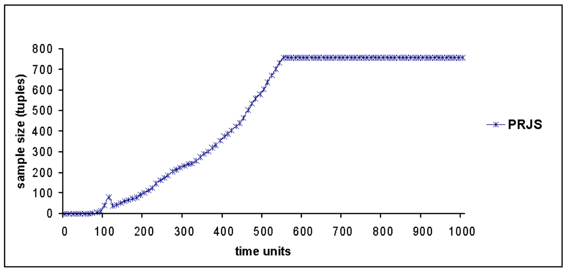


Figure 6. Change in sample size over time.

sample size increases linearly until the enlarged reservoir is filled, and then the increase saturates. The same happens for RJS, but sample size does not ever exceed the initial reservoir size.

Figure 6 shows the sample size over the first 1000 time units for a single run. Note that the sample size decreases initially because some sample tuples are evicted from the reservoir after x and y are decided. This is recovered quickly after that.

Figure 7 shows the effect of PRJS on the reservoir size for varying l. We use l instead of m because the value of m is an expected value for a given l (see Equation 7). The figure shows that the increase of size is larger for larger values of l. The effect saturates for relatively large values of l.

5.3. Reservoir sample uniformity

The purpose of this set of experiments is to test the uniformity of the sample in the reservoir. We use χ^2 statistic [18] [23] as a metric of the sample uniformity. Higher χ^2 indicates lower uniformity and vice versa. χ^2 statistic measures, for each value v in a domain D, the relative difference between the observed number of tuples ($o(v)$) and the expected number of tuples ($e(v)$) that contain the value. That is:

$$\chi^2 = \sum_{\forall v \in D} \frac{(e(v) - o(v))^2}{e(v)} \tag{9}$$

Figure 8 shows χ^2 statistic over time for both algorithms, at the interval of 100 time units. The underlying assumption is that the input stream is randomly sorted on the join attribute value. The results in the figure show that for PRJS the uniformity is decreased after the reservoir size

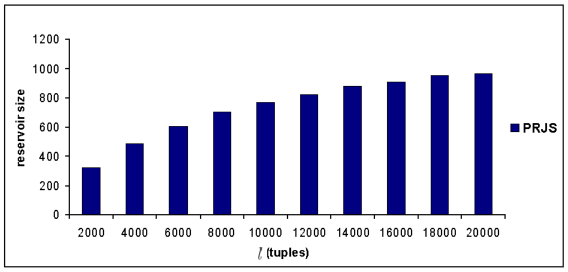
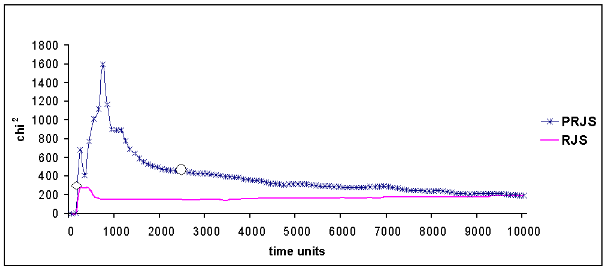


Figure 7. Effect of l on the reservoir size.



Reservoir size increase time is marked with a diamond and sample-use time is marked with a circle.

Figure 8. Change in sample uniformity over time.

is increased, but it starts recovering before the sample-use time. As expected, the sample uniformity for RJS is better and is almost stable over time.

Since PRJS evicts some tuples from the reservoir in order to refill the reservoir with the incoming tuples, the uniformity can be damaged more if there is some sort of dependence in the arrival of join attribute values on the input streams. Therefore, we conduct an experiment to test the effect of the ordering of tuples in the input streams by the join attribute. For this, we generate partially sorted streams. This is done by configuring the values in the domain of the attribute into a tree structure. In the tree, the value in a parent node has a precedence in appearing in the input stream over the values in the children nodes. Between siblings there is no precedence conditions. The number of children of each node is fixed and is parameterized as $fanout$. As the value of $fanout$ decreases, the stream becomes more sorted. That is, if $fanout = 1$, the stream is totally sorted. We set the value of $fanout$ to 2, 3, and 4 as shown in Figure 9. The figure shows that, for PRJS, there is more damage on the uniformity when the degree of the input stream ordering is higher. On the other hand, RJS is not sensitive for any kind of ordering in the input stream. This is evident for RJS and, thus, we omit the graph.

5.4. Aggregation on a reservoir sample

In this set of experiments, we compare RJS and PRJS in terms of the accuracy of aggregation (AVG) query results. We report the average absolute error (AE), at the interval

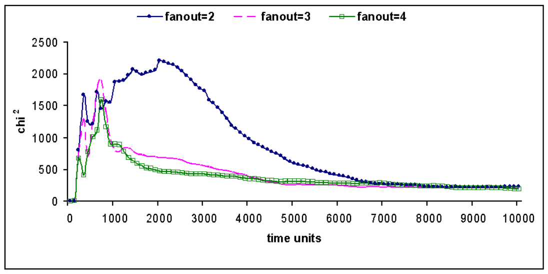


Figure 9. Sample Uniformity for partially-sorted streams.

of 500 time units, for each algorithm. Absolute error (AE) is defined as follows:

$$AE = \sum_{i=1}^{n} \frac{\mid \hat{A}_i - A_i \mid}{n} \tag{10}$$

where A_i ($i = 1, 2, ..., n$) is the exact aggregation result computed from the original join result and $\hat{A}_i$ is the aggregation result computed from a sample in the reservoir, and n is number of runs.

The results shown in Figure 10 demonstrate that right after the reservoir size increases, PRJS gives a larger aggregation error but, after that, as the sample size increases the aggregation errors decreases. The curve of PRJS crosses over the curve of RJS even before reaching the sample-use time (marked as a circle on the PRJS curve). This happens because the benefit of the enlarged reservoir size dominates over the damage in the uniformity. As the uniformity recovers more, the aggregation error decreases more.

6. Related Work

The problem addressed in this paper combines three main research topics: sampling from data streams, uniform random sampling over joins, and memory-limited stream joins. We briefly review related work to each topic and then point out current research combining them.

In addition to the reservoir sampling [25] [33], there are special-purpose algorithms for sampling over data streams, such as *heavy hitters* [24] and *distinct counts* [14]. Heavy hitters find the elements that appear in a data stream for at least a certain fraction of all tuples, and distinct counts estimate the number of distinct values for a given target attribute over an input stream. In [7] Babcock et al. present memory-efficient algorithms for maintaining a uniform random sample of a specified size from a moving window over a data stream. In [20], Johnson et al. abstract the process of sampling from a stream and design a generic sampling operator which can be specialized to implement a wide variety of stream sampling algorithms. In [3], we theoretically study the effects of increasing and decreasing the reservoir size in the middle of sampling on the sample uniformity. We present an adaptive-size reservoir sampling algorithm and apply it to an application in which samples are collected from wireless sensor networks using a mobile sink.

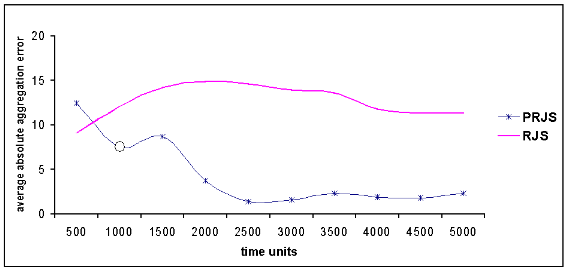


Figure 10. Absolute error in AVG aggregation query over time.

Uniform random sampling over joins has been addressed in the context of relational databases. Olken et al. [28] [27] assume that the relations being operated on should be base relations and have indexes. They consider both the case in which the join attribute is a key in one or more base relations and the case in which the join attribute is not a key in any of the base relations. Chaudhuri et al. [10] improve on Olken's results by considering more general settings in which intermediate relations in a query tree are typically not materialized and indexed. Acharya et al. [2] focus on computing approximate aggregation on multi-way joins by making use of pre-computed samples (known as join-synopses).

Memory-limited stream processing [26] has been considered in the context of stream joins. In [12], [22], and [34] this problem has been addressed by dropping tuples from input streams randomly [22] or based on the tuple content [12]. Xie et al. [34] present a stochastic process for load shedding by observing and exploiting statistical properties of input streams. Sketching techniques are proposed by Dobra et al. in [13] and by Alon et al. [4] for evaluating multi-join aggregate queries over data streams with limited memory.

There are a few research works combining some of these topics. With the of objective of minimizing answers inaccuracy in sliding-window aggregation queries where processing power is limited, Babcock et al. [6] propose a load shedding technique which includes random sampling operators in a query plan. Their work is different from ours as they assume a streaming environment in which the processing power, not the memory, is limited. In [31], the problems of sampling from data streams, uniform random sampling over joins, and limited memory-limited stream joins are combined. However, our work is different from theirs as we have the objective of maintaining the sample in a reservoir, instead of streaming out the sample tuples. This makes the problem we are addressing distinct and challenging.

7. Conclusion

In this paper, we studied reservoir sampling over memory-limited stream joins, and presented two algorithms for this problem. One algorithm uses fixed-size reservoir and always maintains a uniform sample. The other algorithm allows the reservoir size to increase during the join-sampling by releasing memory from the join buffer and allocating it to the reservoir. We show theoretically that such an increase comes with a negative impact on the probability of the sample being uniform. We reported experimental evaluation comparing the two presented algorithms.

Several issues are still open for future work. It is interesting to extend the presented algorithms to support *group-by* queries, where a reservoir may naturally keep tuples from multiple groups. In this case, we may need to *bias* the sampling in order to collect as equal number of tuples as possible from each group, which results in an accurate approximation of the query results [1]. It is also challenging to study the addressed problem when resources are shared among *multiple queries*. In this case, the developed algorithm should optimally allocate the memory between reservoirs and join buffers of all participating queries.

Acknowledgment

This research is based upon work supported by the National Science Foundation under Grant No. IIS-0415023

References

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD '00*, pages 487–498.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD '99*, pages 275–286.

[3] M. Al-Kateb, B. S. Lee, and X. S. Wang. Adaptive-size reservoir sampling over data streams. In *SSDBM '07*.

[4] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *PODS '99*, pages 10–20.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS '02*, pages 1–16.

[6] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04*, pages 350–361.

[7] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA '02*, pages 633–634.

[8] M. M. Cecilia Mascolo and B. Pasztor. Data collection in delay tolerant mobile sensor networks using SCAR. In *SenSys '06*.

[9] I. Chatzigiannakis, A. Kinalis, and S. Nikoletseas. Sink mobility protocols for data collection in wireless sensor networks. In *MobiWac '06*, pages 52–59.

[10] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD '99*, pages 263–274.

[11] A. S. D. Jea and M. Srivastava. Multiple controlled mobile elements (data mules) for data collection in sensor networks. In *DCOSS '05*, pages pages 244–257.

[12] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD '03*, pages 40–51.

[13] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD '02*, pages 61–72.

[14] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB '01*, pages 541–550.

[15] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD '98*, pages 331–342.

[16] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB '97*, pages 466–475.

[17] L. Golab and M. T. Ozsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[18] N. M. Greenwood, P.E. *A Guide to Chi-Squared Testing.* John Wiley and sons, New York., 1996.

[19] S. Jain, R. C. Shah, W. Brunette, G. Borriello, and S. Roy. Exploiting mobility for energy efficient data collection in wireless sensor networks. *Mob. Netw. Appl.*, 11(3):327–339, 2006.

[20] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD '05*, pages 1–12.

[21] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS-X '02*, pages 96–107.

[22] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE '03*, pages 341–352.

[23] S. L. Lohr, editor. *Sampling: Design and Analysis*. Duxbury Press, Pacific Grove, 1999.

[24] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB '02*, pages 346–357.

[25] A. McLeod and D. Bellhouse. A convenient algorithm for drawing a simple random sample. *Applied Statistics*, 32:182184, 1983.

[26] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR '03*.

[27] F. Olken. *Random Sampling from Databases*. PhD thesis, Mailstop 50B-3238, 1 Cyclotron Road, Berkeley, California 94720, U.S.A., 1993.

[28] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB '86*, pages 160–169.

[29] R. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: Modeling a three-tier architecture for sparse sensor networks. In *IEEE SNPA Workshop '03*, 2003.

[30] A. A. Somasundara, A. Ramamoorthy, and M. B. Srivastava. Mobile element scheduling for efficient data collection in wireless sensor networks with dynamic deadlines. In *RTSS '04*, pages 296–305.

[31] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB '04*, pages 324–335.

[32] I. Vasilescu, K. Kotay, D. Rus, M. Dunbabin, and P. Corke. Data collection, storage, and retrieval with an underwater sensor network. In *SenSys '05*, pages 154–165.

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

[34] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. In *SIGMOD '05*, pages 359–370.

[35] Y. L. Y. Tirta, Z. Li and S. Bagchi. Efficient collection of sensor data in remote fields using mobile collectors. In *ICCCN 2004*.

[36] D. Yi-Leh Wu, Agrawal and A. El Abbadi. Query estimation by adaptive sampling. In *ICDE '02*, pages 639–648.