Reservoir Sampling over Joins

Binyang Dai
HKUST
bdaiab@connect.ust.hk

Xiao Hu
University of Waterloo
xiaohu@uwaterloo.ca

Ke Yi
HKUST
yike@ust.hk

ABSTRACT

Sampling over joins is a fundamental task in large-scale data analytics. Instead of computing the full join results, which could be massive, a uniform sample of the join results would suffice for many purposes, such as answering analytical queries or training machine learning models. In this paper, we study the problem of how to maintain a random sample over joins while the tuples are streaming in. Without the join, this problem can be solved by some simple and classical reservoir sampling algorithms. However, the join operator makes the problem significantly harder, as the join size can be polynomially larger than the input. We present a new algorithm for this problem that achieves a near-linear complexity. The key technical components are a generalized reservoir sampling algorithm that supports a predicate, and a dynamic index for sampling over joins. We also conduct extensive experiments on both graph and relational data over various join queries, and the experimental results demonstrate significant performance improvement over the state of the art.

1. INTRODUCTION

In large-scale data analytics, people often need to compute complicated functions on top of the query results over the underlying relational database. However, the join operator presents a major challenge, since the join size can be polynomially larger than the original database. Computing and storing the join results is very costly, especially as the data size keeps increasing. Sampling the join results is thus a common approach used in many complicated analytical tasks while providing provable statistical guarantees. One naive method is to first materialize the join results in a table and then randomly access the table, but this loses the performance benefit of sampling. As early as 1999, two prominent papers [10, 7] asked the intriguing question, of whether a sample can be obtained without computing the full join. As observed in by [10], the main barrier is that the sampling operator cannot be pushed down, i.e., $\mathsf{sample}(R \bowtie S) \neq \mathsf{sample}(R) \bowtie \mathsf{sample}(S)$. To overcome this barrier, the idea is to design some indexes to guide the sampling process. Notably, an index was proposed for *acyclic* joins [13] that can be built in $O(N)$ time, where N is the number of tuples in the database, which can then be used to draw a sample of the join results in $O(1)$ time [18, 9].

This problem becomes more challenging in the streaming setting, where input tuples arrive at a high velocity. How to efficiently and continuously maintain a uniform sample of the join results produced by tuples seen so far? One naive solution would be to rebuild the index and re-draw the samples after each tuple has arrived, but this results in a total runtime of $O(N^2)$ to process a stream with N tuples. Recently, [19] applied the reservoir sampling algorithms [17, 16] to this problem to update the sample incrementally. However, their index also suffers from a high maintenance cost that still leads to a total runtime of $O(N^2)$ in the worst case.

This paper presents a new reservoir sampling algorithm for maintaining a sample over joins with a near-linear runtime of $O(N \log N + k \log N \log \frac{N}{k})$, where k is the given sample size. Our algorithm does not need the knowledge of N; equivalently speaking, it works over an unbounded stream, and the total runtime over the first N tuples, for every $N \in \mathbb{Z}^+$, satisfies the aforementioned bound. This result is built upon the following two key technical ingredients, both of which are of independent interest.

Reservoir sampling with predicate. The classical reservoir sampling algorithm, attributed to Waterman by Knuth [15], maintains a sample of size k in $O(N)$ time over a stream of N items, which is already optimal. Assuming there is a $\mathsf{skip}(i)$ operation that can skip the next i items and jump directly to the next $(i+1)$-th item in $O(1)$ time, the complexity can be further reduced to $O(k \log \frac{N}{k})$, and there are several algorithms achieving this [17, 16]. In this paper, we design a more general reservoir sampling algorithm that, for a given predicate θ, maintains a sample of size k only over the items on which θ evaluates to true. The complexity of our algorithm is $O\left(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i+1}\right)\right)$, where r_i is the number of items in the first $i-1$ items that pass the predicate. Note that when $\theta(\cdot) \equiv \texttt{true}$, we have $r_i = i-1$ and the bound simplifies to $O(k \log \frac{N}{k})$, matching the classical result. Meanwhile, the complexity degrades gracefully as the stream becomes sparser, i.e., less items pass the predicate. Intuitively, sparse streams requires more care and we cannot skip too aggressively. In the extreme case where only one item passes the predicate, then the algorithm is required to return that item as the sample, and we have to check every

© ACM 2025. This is a minor revision of the paper entitled "Reservoir Sampling over Joins", published in Proc. ACM Manag. Data 2, 3 (SIGMOD), Article 118 (June 2024), 26 pages. DOI: https://doi.org/10.1145/3654921

item in order not to miss it.

However, the assumption that $\mathsf{skip}(i)$ takes $O(1)$ time is usually not true: One has to at least use a counter to count how many items have been skipped, which already takes $O(i)$ time. Interestingly, the reservoir sampling over joins problem provides a nice scenario where this assumption *is* true, except that it has an $O(\log N)$ cost. It is known [8] that the join results of N tuples can be as many as N^{ρ^*}, where ρ^* is the fractional edge cover number of the join. Thus, the stream of input tuples implicitly defines a polynomially longer (conceptual) stream of join results, which we want to sample from. As there is good structure in the latter, there is no need to materialize this simulated join result stream, and it is possible to skip its items without counting them one by one.

Dynamic sampling from joins. Let $\mathcal{Q}$ be an acyclic join query, $\mathcal{R}$ a database instance of size N, and $\mathcal{Q}(\mathcal{R})$ the join results of $\mathcal{Q}$ on $\mathcal{R}$. The second technical ingredient is an index structure that supports the following operations:

(1) After a tuple is added to $\mathcal{R}$, the index structure can be updated in $O(\log N)$ time amortized.

(2) The index implicitly defines an array J that contains $\mathcal{Q}(\mathcal{R})$ plus some dummy tuples, but it is guaranteed that $|J| = O(|\mathcal{Q}(\mathcal{R})|)$, i.e., the dummy tuples are no more than a constant fraction. For any given $j \in [|J|]$, the index can return $J[j]$ in $O(\log N)$ time. It can also return $|J|$ in $O(1)$ time.

(3) The above is also supported for the delta query $\Delta\mathcal{Q}(\mathcal{R}, t) := \mathcal{Q}(\mathcal{R} \cup \{t\}) - \mathcal{Q}(\mathcal{R})$ for any tuple $t \notin \mathcal{R}$.

Note that operation (2) above directly solves the join sampling problem: We simply generate a random $j \in [|J|]$ and find $J[j]$, and repeat if it is dummy. Since $|J| = O(|\mathcal{Q}(\mathcal{R})|)$, this process will terminate after $O(1)$ trials in expectation, so the time to draw a sample is $O(\log N)$ expected. This is only slightly slower than the previous index structures [18, 9], which are inherently static. Furthermore, operations (1) and (2) together also provide a solution for the reservoir sampling over join problem: For each tuple, we first update the index in $O(\log N)$ time and then re-draw k samples in $O(k \log N)$ time. This leads to a total time of $O(Nk \log N)$, already better than [19], but still not near-linear.

To achieve near-linear time, we use operation (3) in conjunction with our reservoir sampling algorithm. The observation is that each incoming tuple t adds a batch of join results, which are defined by the delta query $\Delta\mathcal{Q}(\mathcal{R}, t)$. If we can access any tuple in $\Delta\mathcal{Q}(\mathcal{R}, t)$ by position, then we can implement a skip easily. Our index can almost provide this functionality, except that it does so over ΔJ, which is a superset of $\Delta\mathcal{Q}(\mathcal{R}, t)$ that contains some dummy tuples. This is exactly the reason why we need a reservoir sampling algorithm that supports a predicate. We will run it over the stream of batches, where each batch is the ΔJ of the corresponding delta query. The predicate evaluates to true for the real tuples while false for the dummies. Finally, since each batch is dense (at least a constant fraction is real), our reservoir sampling algorithm will have good performance.

2. PROBLEM DEFINITION

A multi-way (natural) join query can be defined as a hypergraph $\mathcal{Q} = (\mathcal{V}, \mathcal{E})$ [6], where $\mathcal{V}$ is the set of attributes, and $\mathcal{E} \subseteq 2^{\mathcal{V}}$ is the set of relations. Let $\mathrm{dom}(x)$ be the domain of attribute $x \in \mathcal{V}$. A database instance $\mathcal{R}$ consists of a relation instance R_e for each $e \in \mathcal{E}$, which is a set of tuples and each tuple $t \in R_e$ specifies a value in $\mathrm{dom}(v)$ for each attribute $v \in e$. For a tuple t, we use $\mathsf{supp}(t)$ to denote the support of t, i.e., the set of attributes on which t is defined. For attribute(s) x and tuple t with $x \subseteq \mathsf{supp}(t)$, the projection $\pi_x t$ is the value of tuple t on attribute(s) x. The join results of $\mathcal{Q}$ over instance $\mathcal{R}$, denoted by $\mathcal{Q}(\mathcal{R})$, is the set of all combinations of tuples, one from each R_e, that share common values for their common attributes, i.e.,

$$\mathcal{Q}(\mathcal{R}) = \left\{ t \in \prod_{x \in \mathcal{V}} \mathrm{dom}(x) \mid \forall e \in \mathcal{E}, \exists t_e \in R_e, \pi_e t = t_e \right\}.$$

For relation R_e and tuple t, the semi-join $R_e \ltimes t$ returns the set of tuples from R_e which have the same value(s) on the attribute(s) $e \cap \mathsf{supp}(t)$ with t. For a pair of relations $R_e, R_{e'}$, the semi-join $R_e \ltimes R_{e'}$ is the set of tuples from R_e which has the same value(s) on the attribute(s) $e \cap e'$ with at least one tuple from $R_{e'}$. Note that for a join query $\mathcal{Q}$, the delta query $\Delta\mathcal{Q}(\mathcal{R}, t)$ is equal to $\mathcal{Q}(\mathcal{R} \cup \{t\}) \ltimes t$.

In the streaming setting, we model each tuple as a triple $u = (t, i, R_e)$ for $i \in \mathbb{Z}^+$, indicating that tuple t is inserted into relation R_e at time i. Let D be the stream of input tuples, ordered by their timestamp. Let $\mathcal{R}^i$ be the database defined by the first i tuples of the stream, and set $\mathcal{R}^0 = \emptyset$. We use N to denote the length of the stream, which is only used in the analysis. The algorithms will not need the knowledge of N, so they work over an unbounded stream.

There are two versions of the join sampling problem: As studied in [7, 10, 18, 9, 11, 12, 14], one is an indexing (data structure) problem where we wish to have an index that supports drawing a sample from $\mathcal{Q}(\mathcal{R}^i)$. For this problem, we care about the sampling time t_s and the update time t_u. For a static index, we care about the index construction time and the sampling time. The other is the reservoir sampling problem, as studied in [17, 16, 19], where we wish to maintain k random samples from $\mathcal{Q}(\mathcal{R}^i)$ without replacement for every $\mathcal{Q}(\mathcal{R}^i), i \in \mathbb{Z}^+$. For this problem, we care about the total runtime. Note that any solution for the former yields a solution for the latter with total time $O(t_u \cdot N + t_s \cdot Nk)$, but this may not be optimal. For both versions of the problem, all algorithms, including ours, use $O(N)$ space.

We follow the convention of data complexity [6] and analyze the runtime in terms of the input size N and sample size k, while taking the size of $\mathcal{Q}$ (i.e., $|\mathcal{V}|$ and $|\mathcal{E}|$) as a constant. We follow the set semantics, so inserting a tuple into a relation that already has it has no effect, thus duplicates have been removed from the input stream.

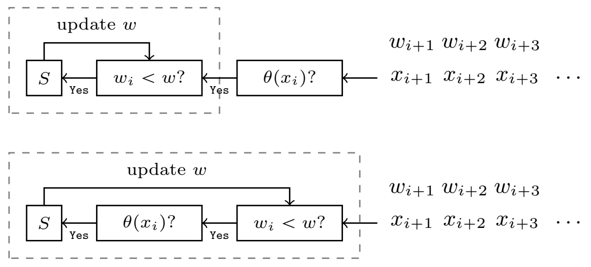
3. RESERVOIR SAMPLING WITH PREDICATE

3.1 Reservoir Sampling Revisited

Reservoir sampling [17, 16] is a family of algorithms for maintaining a random sample, without replacement, of k items from a possibly infinite stream. The classical version [15] works as follows. **(Step 1)** It initializes an array S (called the *reservoir*) of size k, which contains the first k items of the input. **(Step 2)** For each new input x_i, it generates a random number j uniformly in $[1, i]$. If $j \leqslant k$, then it replaces $S[j]$ with x_i. Otherwise, it simply discards x_i. At any time, S is

Algorithm 1: RESERVOIR(D, k, θ)

```
Input : An input stream $D$ of items, an integer $k > 0$, and a predicate $\theta$;
Output: A set $S$ maintaining $k$ random samples without replacement of items on which $\theta$ evaluates to true;
1  $S \leftarrow \varnothing$;
2  while $|S| < k$ do
3      $x \leftarrow D$.next();
4      if $x =$ null then break;
5      if $\theta(x)$ then $S \leftarrow S \cup \{x\}$;
6  $w \leftarrow \mathsf{rand}()^{1/k}$;
7  $q \leftarrow \lfloor \ln(\mathsf{rand}())/\ln(1-w) \rfloor$;
8  while true do
9      $x \leftarrow D.\mathsf{skip}(q)$;
10     if $x =$ null then break;
11     if $\theta(x)$ then
12         $y \leftarrow$ a randomly chosen item from $S$;
13         $S \leftarrow S - \{y\} + \{x\}$;
14         $w \leftarrow w \cdot \mathsf{rand}()^{1/k}$;
15     $q \leftarrow \lfloor \ln(\mathsf{rand}())/\ln(1-w) \rfloor$; ($q \sim \mathsf{Geo}(w)$)
```



update w
S
Yes
$w_i < w$?
Yes
$\theta(x_i)$?
w_{i+1} w_{i+2} w_{i+3}
x_{i+1} x_{i+2} x_{i+3} $\cdots$
update w
S
Yes
$\theta(x_i)$?
Yes
$w_i < w$?
w_{i+1} w_{i+2} w_{i+3}
x_{i+1} x_{i+2} x_{i+3} $\cdots$

Figure 1: Comparison between the naive (above) and our algorithm (below) on reservoir sampling with predicate.

a uniform sample without replacement of k items of all items processed so far. Clearly, this algorithm takes $O(N)$ time to process a stream of N items. Also, the algorithm does not need the knowledge of N, so it works over an unbounded stream.

Assuming a $\mathsf{skip}(i)$ operation that can skip the next i items in $O(1)$ time, more efficient versions are known. In particular, we will make use of the one from [16]. It is based on the fact that, in a set of N independent random numbers drawn the uniform distribution $\mathsf{Uni}(0,1)$, the indices of the smallest k random numbers are a sample without replacement from the index set $\{1, 2, \cdots, N\}$. The algorithm works as follows. **(Step 1)** It initializes S as before, and set $w = u^{1/k}$ for $u \sim \mathsf{Uni}(0,1)$. **(Step 2)** It draws a random number q from the geometric distribution $\mathsf{Geo}(w)$, and skip the next q items. It then replaces a random item from S with x_i, and updates w to $w \cdot u^{1/k}$ for $u \sim \mathsf{Uni}(0,1)$. It can be shown [16] that at any time, S is a sample without replacement of k items of all items processed so far, and this algorithm runs in $O(k \cdot \log \frac{N}{k})$ expected time, which is optimal.

3.2 Reservoir Sampling with Predicate

The problem of *reservoir sampling with predicate* is defined as follows. Given an input stream of items, a predicate θ and an integer $k > 0$, it asks to maintain a sample of size k of all items on which θ evaluates to true (these items are also called real items, while the others dummy). We assume that θ can be evaluated in $O(1)$ time. Note that the $O(N)$-time algorithm easily supports a predicate: We just evaluate θ on each item and feed the real items to the algorithm as illustrated in Figure 1. It is more nontrivial to adapt the $O(k \log \frac{N}{k})$ algorithm since the skip operation skips an unknown number of real items.

We adapt the reservoir sampling algorithm [16] to Algorithm 1. The insight is that the random variable w_i, which is drawn from $\mathsf{Uni}(0,1)$ when item x is visited, is independent of $\theta(x)$. This property allows us to swap the order of the two comparisons ($w_i < w$ and whether $\theta(x)$ is true), as illustrated in Figure 1. Note that we do not need to explicitly assign a random number to each item as the number of failures before the next success (i.e., $w_i < w$) follows a geometric distribution parameterized by w. As in [16], we draw a random variable $q \sim \mathsf{Geo}(w)$ to determine the number of items to skip. After skipping q items, we fetch the next item x and evaluate $\theta(x)$. If it satisfies θ, we insert it into the reservoir S and update w to $w \cdot u^{1/k}$, where $u \sim \mathsf{Uni}(0,1)$. Otherwise, we keep w unchanged and draw the next random variable $q \sim \mathsf{Geo}(w)$.

In the description, we use the following primitives:

- **next()** returns the next item if it exists, and **null** otherwise;
- **skip**(i) skips the next i items and returns the $(i+1)$-th item if it exists, and **null** otherwise.

Compared with [16], we have made two changes: (lines 2-5) when the reservoir is not full, we only add real items to it; (lines 11 - 14) we only update the reservoir and the parameter w when the algorithm stops at a real item. The correctness proof of Algorithm 1 is given in the full version [1]. The time complexity of Algorithm 1 depends on how the real and dummy items are distributed in the stream, as more precisely characterized by the following theorem:

THEOREM 1. *Algorithm 1 runs in $O(\alpha \cdot (p-1) + \gamma \cdot \sum_{i=p}^{N} \frac{k}{r_i+1})$ expected time over a stream of N items, where r_i is the number of real items in the first $i-1$ items, p is the smallest i such that $r_i = k$ (set $p = N+1$ if no such p exists), and α and γ are the runtime of* $\mathsf{next}(\cdot)$ *and* $\mathsf{skip}(\cdot)$*, respectively.*

In the degenerate case where all items are real, we have $r_i = i-1$ and the runtime of Algorithm 1 becomes $O(k \log \frac{N}{k})$ (when taking α, γ as $O(1)$), matching the optimal reservoir sampling runtime [17, 16]. In the other extreme case, all items are dummy, so $p = N+1$ and $r_i = 0$, and the runtime becomes $O(N)$, i.e., no item is skipped. Indeed in this case, it is not safe to skip anything; otherwise, the algorithm may miss the first real item if one shows up, which must be sampled. Below, we formalize this intuition and prove that Algorithm 1 is not just optimal in these two degenerate cases, but in all cases, namely, it is instance-optimal.

THEOREM 2. *For any input stream S of N elements, any algorithm maintaining a uniform sample of size k over all real elements runs in $\Omega\left(\sum_{i=1}^{N} \min\{1, \frac{k}{r_i+1}\}\right)$ expected time.*

Although the runtime of Algorithm 1 can vary significantly from $O(k \log \frac{N}{k})$ to $O(N)$, it is closer to the former as

```
Algorithm 2: BatchReservoir($D, k, \theta$)
  Input : An input stream $D$ of item-disjoint batches,
          an integer $k > 0$, and a predicate $\theta$;
  Output: A set $S$ maintaining $k$ random samples
          without replacement of items on which $\theta$
          evaluates to true;
1 $S \leftarrow \varnothing$, $w \leftarrow +\infty$, $q \leftarrow 0$;
2 foreach batch $B \in D$ do
3     $(S, w, q) \leftarrow$ BatchUpdate($S, k, B, q, w, \theta$);
```

```
Algorithm 3: BatchUpdate($S, k, B, q, w, \theta$)
  Input : A set $S$ of random samples, an integer $k > 0$,
          a new batch $B$ with the first $q$ items to be
          skipped, parameter $w$ and a predicate $\theta$;
  Output: Updated $S$, $w$ and $q$;
1  while $|S| < k$ and $B$.remain() $> 0$ do
2      $x \leftarrow B$.next();
3      if $\theta(x)$ then $S \leftarrow S \cup \{x\}$;
4  if $|S| < k$ then return $S, w, q$;
5  if $w > 1$ then
6      $w \leftarrow \mathsf{rand}()^{1/k}$;
7      $q \leftarrow \lfloor(\ln(\mathsf{rand}())/\ln(1-w))\rfloor$; ($q \sim \mathsf{Geo}(w)$)
8  while $B$.remain() $> q$ do
9      $x \leftarrow B$.skip($q$);
10     if $\theta(x)$ then
11         $y \leftarrow$ a randomly chosen item from $S$;
12         $S \leftarrow S - \{y\} + \{x\}$;
13         $w \leftarrow w \cdot \mathsf{rand}()^{1/k}$;
14     $q \leftarrow \lfloor(\ln(\mathsf{rand}()/\ln(1-w))\rfloor$; ($q \sim \mathsf{Geo}(w)$)
15 return $S, w, q - B$.remain();
```

long as the stream is dense enough. Combining Theorem 1 and Definition 1 we obtain:

Definition 1. *Given a stream $S = \langle x_1, x_2, \cdots, x_n \rangle$, S is ϕ-dense for $0 < \phi \leqslant 1$, if $r_i \geqslant \phi \cdot (i-1)$ for all i.*

Corollary 1. *For any ϕ-dense stream where ϕ is a constant, Algorithm 2 runs in $O(\alpha \cdot k + \gamma \cdot k \log \frac{N}{k})$ expected time.*

We also mention three important properties for dense streams, which will be used later for joins. Lemma 1 implies that straightforwardly concatenating two streams still preserves their minimum density of real items. Lemma 2 implies that mixing two streams as their Cartesian product preserves a density that is at least half of their density product. Lemma 3 implies that if one stream only consists of dummy items, it is possible to get a better bound on the density of real items in the entire stream. The more dummy items padded, the sparser the stream becomes.

Lemma 1. *Given two streams $S_1 = \langle x_1, x_2, \cdots x_m \rangle$ and $S_2 = \langle y_1, y_2, \cdots, y_n \rangle$, if S_1 is ϕ_1-dense and S_2 is ϕ_2-dense, their concatenation $S_1 \circ S_2 := \langle x_1, x_2, \cdots, x_m, y_1, y_2, \cdots, y_n \rangle$ is $\min\{\phi_1, \phi_2\}$-dense.*

Lemma 2. *Given two streams $S_1 = \langle x_1, x_2, \ldots, x_m \rangle$ and $S_2 = \langle y_1, y_2, \ldots, y_n \rangle$, if S_1 is ϕ_1-dense and S_2 is ϕ_2-dense, their Cartesian product $S_1 \times S_2 := \langle (x_1, y_1), \cdots, (x_1, y_n),$ $(x_2, y_1), \cdots, (x_2, y_n), (x_m, y_1), \cdots (x_m, y_n) \rangle$ is $\left(\frac{\phi_1 \phi_2}{2}\right)$-dense, where (x_i, x_j) is real if and only if both x_i and x_j are real.*

Lemma 3. *Given a ϕ-dense stream of size m, padding n dummy items at the end yields a $\left(\frac{m}{m+n} \cdot \phi\right)$-dense stream.*

```
Algorithm 4: ReservoirJoin($\mathcal{Q}, D, k$)
  Input : A join query $\mathcal{Q}$, an input stream $D$ of tuples,
          and an integer $k > 0$;
  Output: A set $S$ maintaining $k$ random samples
          without replacement for the join results of $\mathcal{Q}$
          over tuples seen as far;
1 Initialize $\mathcal{L}$, $S \leftarrow \varnothing$, $w \leftarrow +\infty$, $q \leftarrow 0$, $\theta \leftarrow \mathsf{isReal}(\cdot)$;
2 while true do
3     $t \leftarrow D$.next();
4     if $t$ = null then break;
5     $\mathcal{L} \leftarrow$ IndexUpdate($\mathcal{L}, t$);
6     $B \leftarrow$ BatchGenerate($\mathcal{Q}, \mathcal{L}, t$);
7     $(S, w, q) \leftarrow$ BatchUpdate($S, k, w, q, B, \theta$);
```

3.3 Batched Reservoir Sampling with Predicate

As shown in Section 1, each arriving tuple t derives a batch of new join results $\Delta\mathcal{Q}(\mathcal{R}, t)$. To apply our approach, we first adapt Algorithm 1 into a batched version. Formally, given an input stream of item-disjoint batches $\langle B_1, B_2, \cdots, B_m \rangle$, and a predicate θ, the goal is to maintain k uniform samples without replacement from $B_1^\theta \cup B_2^\theta \cup \cdots \cup B_i^\theta$ for every i, where $B_i^\theta \subseteq B_i$ is the set of real items in batch B_i. The framework of our batched version is described in Algorithm 2. It invokes BatchUpdate to every batch, which essentially runs Algorithm 1 on the given batch while guarding against the case where skip() may skip out of the batch. For this purpose, it needs another primitive:

- **remain()** returns the number of remaining items in a batch.

More precisely, when B.remain() $\leqslant q$, we skip all the remaining items in the current batch, and pass $q - B$.remain() as another parameter to the next batch so that the first $q - B$.remain() items in the next batch will be skipped. The details are given in Algorithm 3. Moreover, we note that parameters w, q are only initialized once (as line 6-7 in Algorithm 1), i.e., the first time when the reservoir S is filled with k items. To ensure this in the batched version, we set w with $+\infty$ at the beginning (line 1 of Algorithm 2), so that w, q will be initialized the first time when the reservoir S is filled with k items, and will never be initialized again no matter how many times Algorithm 3 is invoked. The samples maintained by Algorithm 2 are the same as that maintained by Algorithm 1 over items in batches, so correctness follows.

Theorem 3. *Given a stream of m item-disjoint batches each of which is ϕ-dense for some constant ϕ, Algorithm 2 runs in $O((\alpha + \beta) \cdot k + (\beta + \gamma) \cdot k \log \frac{N}{k} + m)$ expected time, where N is the total number of items, and α, β, γ are the runtime of* next($\cdot$)*,* remain($\cdot$)*,* skip($\cdot$) *respectively.*

3.4 Reservoir Sampling Over Joins

The framework for reservoir sampling over joins is described in Algorithm 4. For each tuple t in the input stream,

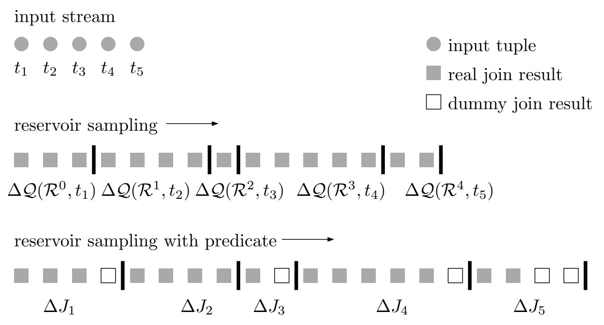

input stream
t_1 t_2 t_3 t_4 t_5
input tuple
real join result
dummy join result
reservoir sampling
$\Delta\mathcal{Q}(\mathcal{R}^0, t_1)$ $\Delta\mathcal{Q}(\mathcal{R}^1, t_2)$ $\Delta\mathcal{Q}(\mathcal{R}^2, t_3)$ $\Delta\mathcal{Q}(\mathcal{R}^3, t_4)$ $\Delta\mathcal{Q}(\mathcal{R}^4, t_5)$
reservoir sampling with predicate
ΔJ_1 ΔJ_2 ΔJ_3 ΔJ_4 ΔJ_5

Figure 2: Comparison between the prior algorithm (above) and our algorithm (below) on reservoir sampling over joins.

we invoke procedure BatchGenerate to conceptually generate a batch $\Delta\mathcal{Q}(\mathcal{R}, t)$ and feed it into the batched reservoir sampling algorithm. However, we cannot afford to materialize each batch, whose total size could be as large as $O(N^{\rho^*})$, where ρ^* is the fractional edge cover number of join. Instead, we will maintain a linear-size index $\mathcal{L}$, which supports a *retrieve* operation that returns the item at position z in $\Delta\mathcal{Q}(\mathcal{R}, t)$ for any given z. The index should also be able to return N_B, the batch size. We further maintain a variable pos to indicate the position of the current item retrieved. Then the three primitives required by the batched reservoir sampling algorithm can be implemented as follows:

- **remain()** returns $N_B - \mathsf{pos}$, where N_B is the size of batch;
- **skip(i)** increases pos by $i+1$ and returns the item at pos (i.e., skips the next i items and jumps directly to the $(i+1)$-th item);
- **next()** simply returns **skip(0)**;

Thus, to apply batched reservoir sampling on any join query $\mathcal{Q}$, it suffices to show how to maintain a linear-size index $\mathcal{L}$ that can efficiently support the retrieve operation for each $\Delta\mathcal{Q}(\mathcal{R}, t)$, as well as $|\Delta\mathcal{Q}(\mathcal{R}, t)|$. This is still hard. To get around this difficulty, we devise an approximate solution. Our index $\mathcal{L}$ will implicitly define a ΔJ that contains all the join results in $\Delta\mathcal{Q}(\mathcal{R}, t)$, plus some dummy results. However, we should not sample from these dummy join results, and this is exactly the reason why we must use a reservoir sampling algorithm that supports the predicate. We set the predicate θ to $\mathsf{isReal}(\cdot)$, which filters out the dummy results. We conceptually add some dummy tuples to base relations as well as some dummy partial join results. In this way, a join result is real if and only if all participated tuples are real, and dummy otherwise (i.e., at least one participated tuple or partial join result is dummy). Finally, we will also guarantee that each ΔJ is dense so as to apply Theorem 3. More specifically, we have $\alpha = \gamma = O(\log n)$ and $\beta = O(1)$. The number of batches is N, and the total number of items is N^{ρ^*}, where ρ^* is a constant since it only depends on the join query, not the data size. Hence, we obtain:

Theorem 4. *Given an acyclic join $\mathcal{Q}$, an initially empty database $\mathcal{R}$, a sample size k, and a stream of N tuples, Algorithm 4 maintains k uniform samples without replacement for each $\mathcal{Q}(\mathcal{R}^i)$, and runs in $O(N \log N + k \log N \log \frac{N}{k})$ expected time.*

Comparison with [19]. [19] follows the same framework as ours, but they simply used the classical reservoir sampling algorithm without a predicate. As such, they must use an index that supports the retrieve operation and the size information directly on $\Delta\mathcal{Q}(\mathcal{R}, t)$; please see Figure 2. Such an index takes $O(N)$ time to update, although they used some heuristics to improve its practical performance. On the other hand, our predicate-enabled reservoir sampling algorithm allows us to use an index filled with dummy results, which can be updated in $O(\log N)$ time as shown next.

4. SAMPLING OVER LINE-3 JOIN

We use the simplest non-trivial join, the line-3 join $R_1(X,Y) \bowtie R_2(Y,Z) \bowtie R_3(Z,W)$ to show how our algorithm works; please see [1] for the full algorithm that works for any acyclic join. For the line-3 join, even maintaining an index for just finding the delta query sizes is difficult: It is still an open problem if there is a better algorithm than computing each delta query size from scratch, which takes $O(N)$ time. This is where we need to introduce dummy join results.

Index. For each $b \in \pi_Y R_1$, we maintain the degree of b in R_1, i.e., $\mathsf{cnt}(b) = |R_1 \ltimes b|$ and its approximation $\tilde{\mathsf{cnt}}(b) = 2^{\lceil \log_2 \mathsf{cnt}(b) \rceil}$ by rounding $\mathsf{cnt}(b)$ up to the nearest power of 2. Similarly, we maintain $\mathsf{cnt}(c) = |R_3 \ltimes c|$ and $\tilde{\mathsf{cnt}}(c) = 2^{\lceil \log_2 \mathsf{cnt}(c) \rceil}$ for each $c \in \pi_Z R_3$. Note that $\tilde{\mathsf{cnt}}(\cdot)$ changes at most $O(\log N)$ times. For each value $b \in \pi_Y R_2$, we organize the tuples $R_2 \ltimes b$ into at most $\log N$ buckets according to the approximate degree of c, where the i-th bucket is

$$\Phi_i(b) = \left\{(b,c) \in R_2 : \tilde{\mathsf{cnt}}(c) = 2^i\right\}.$$

Let $\mathcal{L}_b$ be the list of non-empty buckets. Define $\varphi_i(b) = 2^i \cdot |\Phi_i(b)|$. We also maintain $N_b = \sum_{i \in [\log N]} \varphi_i(b)$ for each value $b \in \pi_Y R_2$, which is an upper bound on the number of new join results if some tuple (a,b) is added to R_1. Symmetrically, for each $c \in \pi_Z R_2$, we maintain such a list $\mathcal{L}_c$, and $N_c = \sum_{i \in [\log N]} \varphi_i(c)$.

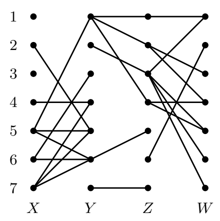
Space Usage. As there are $O(N)$ values in $\pi_Y R_1$, we need to maintain $O(N)$ degrees and their approximations in total. For each $b \in \pi_Y R_2$, it needs to organize the tuples $R_2 \ltimes b$ into buckets and maintain a value N_b. The size of non-empty buckets maintained for b is essentially $|R_2 \ltimes b|$. Summing over all values $b \in \pi_Y R_2$, the total size is $O(N)$. A similar argument applies to $\pi_Z R_2$.

Index Update. After a tuple t has arrived, we update our data structure as follows. If $t \in R_2$, say $t = (b,c)$, we add (b,c) to $\Phi_i(b)$ for $i = \log_2 \tilde{\mathsf{cnt}}(c)$. This just takes $O(1)$ time.

If $t = (a,b) \in R_1$ (the $t \in R_3$ case is similar), we increase $\mathsf{cnt}(b)$ by 1, and update $\tilde{\mathsf{cnt}}(b)$ if needed. If $\tilde{\mathsf{cnt}}(b)$ has changed, for each $c \in \pi_Z(R_2 \ltimes b)$, we remove (b,c) from $\Phi_{i-1}(c)$ and add (b,c) to $\Phi_i(c)$, where $i = \log_2 \tilde{\mathsf{cnt}}(b)$. This may take $O(N)$ time, but this update is only triggered when $\tilde{\mathsf{cnt}}(b)$ doubles, which happens at most $O(\log N)$ times. Thus, the total update cost is bounded by $O(N \log N)$ since

$$\sum_b \lceil \log \mathsf{cnt}(b) \rceil \cdot |\pi_Z(R_2 \ltimes b)| \leqslant \log N \cdot \sum_b |\pi_Z(R_2 \ltimes b)|,$$

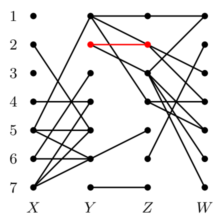
namely, the amortized update cost is $O(\log N)$. Finally, whenever some $\Phi_i(b)$ or $\Phi_i(c)$ changes, we update N_b and N_c accordingly. The time for this update is the same as that for $\Phi_i(b)$ and $\Phi_i(c)$.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	(1, 1)	1	5
	1	(1, 2)(1, 4)	4	
2	2	(2, 3)	4	4

c	i	$\Phi_i(c)$	$\varphi_i(c)$	N_c
1	0	(1, 1)	1	1
2	0	(1, 2)	1	1
4	0	(1, 4)	1	1
5	2	(6, 5)	4	4

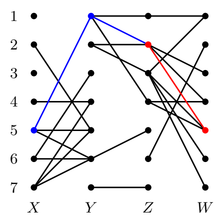
Figure 3: Initial state.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	(1, 1)	1	5
	1	(1, 2)(1, 4)	4	
2	1	(2, 2)	2	6
	2	(2, 3)	4	

c	i	$\Phi_i(c)$	$\varphi_i(c)$	N_c
1	0	(1, 1)	1	1
2	0	(1, 2)	1	1
4	0	(1, 4)	1	1
5	2	(6, 5)	4	4

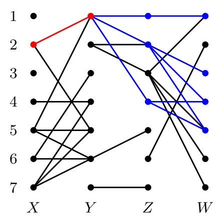
Figure 4: Insert $(2, 2)$ into R_2.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	(1, 1)	1	7
	1	(1, 4)	2	
	2	(1, 2)	4	
2	2	(2, 2)(2, 3)	8	8

c	i	$\Phi_i(c)$	$\varphi_i(c)$	N_c
1	0	(1, 1)	1	1
2	0	(1, 2)	1	1
4	0	(1, 4)	1	1
5	2	(6, 5)	4	4

Figure 5: Insert $(2, 5)$ into R_3.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	(1, 1)	1	7
	1	(1, 4)	2	
	2	(1, 2)	4	
2	2	(2, 2)(2, 3)	8	8

c	i	$\Phi_i(c)$	$\varphi_i(c)$	N_c
1	1	(1, 1)	2	2
2	1	(1, 2)	2	2
4	1	(1, 4)	2	2
5	2	(6, 5)	4	4

Figure 6: Insert $(2, 1)$ into R_1.

Batch Generate. The delta query $\Delta\mathcal{Q}(\mathcal{R}, t)$ on the line-3 join falls into the following 3 cases:

$$\Delta\mathcal{Q}(\mathcal{R}, t) = \begin{cases} \{t\} \times (R_3 \ltimes (R_2 \ltimes b)) & \text{if } t = (a, b) \in R_1 \\ (R_1 \ltimes b) \times \{t\} \ltimes (R_3 \ltimes c) & \text{if } t = (b, c) \in R_2 \\ (R_1 \ltimes (R_2 \ltimes c)) \times \{t\} & \text{if } t = (c, d) \in R_3 \end{cases}$$

The batch $\Delta J \supseteq \Delta\mathcal{Q}(\mathcal{R}, t)$ for any t is defined as follows. If $t \in R_2$, say $t = (b, c)$, then $\Delta J := (R_1 \ltimes b) \times (R_3 \ltimes c)$. This batch is 1-dense and $|\Delta J| = \mathsf{cnt}(b) \cdot \mathsf{cnt}(c)$.

Next, consider the case $t = (a, b) \in R_1$. Consider a bucket $\langle i, \Phi_i(b)\rangle \in \mathcal{L}_b$. For each $(b, c) \in \Phi_i(b)$, define a mini-batch that consists of all tuples in $R_3 \ltimes c$, followed by $\tilde{\mathsf{cnt}}(c) - \mathsf{cnt}(c)$ dummy tuples. We concatenate these mini-batches to form the batch for the bucket. Then we join these mini-batches with t and concatenate all the results to form ΔJ. This ΔJ is $\frac{1}{2}$-dense, since each mini-batch is $\frac{1}{2}$-dense and then we invoke Lemma 1. Moreover, $|\Delta J| = N_b$ and can be returned in $O(1)$ time. The case $t \in R_3$ is similar.

Retrieve. We next show how to retrieve a specific element from the ΔJ defined above. We consider the two cases $t \in R_2$ and $t \in R_1$ ($t \in R_3$ is similar), respectively. If $t = (b, c) \in R_2$, ΔJ is the Cartesian product of $R_2 \ltimes b$ and $R_3 \ltimes c$. Given a position $z \in [|\Delta J|]$, we first find the unique pair $(z_1, z_2) \in [|R_2 \ltimes b|] \times [|R_3 \ltimes c|]$ such that $z = z_1 \cdot |R_3 \ltimes c| + z_2$. Then, we just return the combination of the tuple at position z_1 in $R_2 \ltimes b$ and the tuple at position z_2 in $R_3 \ltimes c$. The retrieve operation in this case takes $O(1)$ time. If $t = (a, b) \in R_1$, we retrieve the tuple at position z as follows:

- Let $i \in [0, \log N]$ be the unique integer such that
$$\sum_{i' \leq i-1 : \Phi_{i'}(b) \neq \varnothing} \varphi_{i'}(b) < z + 1 \leq \sum_{i' \leq i : \Phi_{i'}(b) \neq \varnothing} \varphi_{i'}(b).$$
- Set $j = \left\lfloor \left(z - \sum_{i' \leq i-1 : \Phi_{i'}(b) \neq \varnothing} \varphi_{i'}(b)\right) / 2^i \right\rfloor$.
- Set $\ell = z - \sum_{i' \leq i-1 : \Phi_{i'}(b) \neq \varnothing} \varphi_{i'}(b) - 2^i \cdot j$.

Let t' be the tuple at position j in $\Phi_i(b)$. We return the join result of t and the tuple at position ℓ in $R_3 \ltimes t'$ if $\ell < |R_3 \ltimes t'|$, or a dummy tuple otherwise. As there are at most $O(\log N)$ distinct i's with $\Phi_i(b) \neq \varnothing$, the value of i, j, ℓ can be computed in $O(\log N)$ time. So the retrieve operation takes $O(\log N)$ time in this case.

Example 4.1. *Consider an example database in Figure 3 with an initialized index. Figure 4 shows the updated index after inserting $(2, 2)$ to R_2, where $(2, 2)$ is added to bucket $\Phi_1(2)$, $\varphi_1(2)$ is updated to 2, and N_b is updated to 6 for $b = 2$. Note that $\mathcal{L}_c$ stays unchanged as $2 \notin \pi_Y R_1$. Next, Figure 5 shows the updated index after inserting $(2, 5)$ into R_3, where $(1, 2)$ is moved from $\Phi_1(1)$ to $\Phi_2(1)$ and $(2, 2)$ is moved from $\Phi_1(2)$ to $\Phi_2(2)$. The value of $\varphi_1(1)$, $\varphi_2(1)$, $\varphi_2(2)$, N_1 and N_2 are updated accordingly. Figure 6 shows the updated index after inserting $(2, 1)$ into R_1. Inserting $(2, 2)$ into R_2 generates an empty batch as* $\mathsf{cnt}(b) = 0$ *for $b = 2$. Inserting $(2, 5)$ into R_3 generates a batch with single tuple $(5, 1, 2, 5)$. All tuples involved in this batch are highlighted in blue in Figure 5. Inserting $(2, 1)$ into R_1 generates a batch $\langle(2, 1, 1, 1), (2, 1, 4, 4), (2, 1, 4, 5), (2, 1, 2, 3), (2, 1, 2, 4), (2, 1, 2, 5), \perp\rangle$, which is the concatenation of three mini-batches joining with $(2, 1)$.*

- *$\Phi_0(1)$ produces a mini-batch $\langle(1, 1)\rangle$;*
- *$\Phi_1(1)$ produces a mini-batch $\langle(4, 4), (4, 5)\rangle$;*
- *$\Phi_2(1)$ produces a mini-batch $\langle(2, 3), (2, 4), (2, 5), \perp\rangle$.*

Note that we do not need to materialize the entire batch. For example, to retrieve the item at position 4 in the batch, we get $i = 2$, $j = 0$, and $l = 1$ by the formulas above. Then, we can retrieve $(1, 2)$ from $\Phi_2(1)$ and $(2, 4)$ from $R_3 \ltimes (1, 2) = \{(2, 3), (2, 4), (2, 5)\}$. Finally, we join $(2, 1)$ with $(1, 2), (2, 4)$ to produce the result $(2, 1, 2, 4)$.

5. EXPERIMENTS

5.1 Setup

Implementation. We compare our algorithm (denoted as RSJoin) and the optimized version when foreign-key join exists (denoted as RSJoin_opt), with the algorithm in [19]

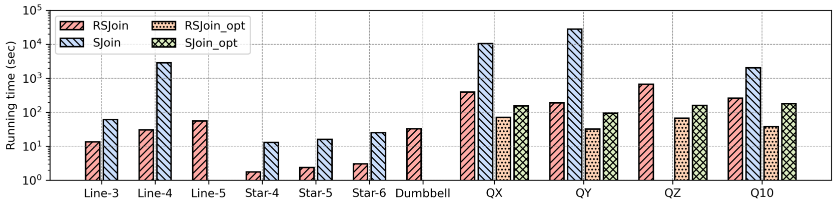


Figure 7: Runtime over different join queries

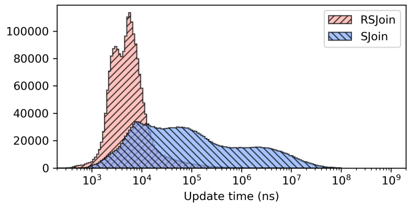


Figure 8: Update time distribution

(denoted as SJoin) and its optimized version when foreign-key join exists (denoted as SJoin_opt), the state-of-the-art method for random sampling over joins under updates. The symmetric hash join algorithm [2] was proposed for computing the (delta) join results for the basic two-table join over data streams, but its performance is overall dominated by [19], hence we do not include it in our experiments. We implement our algorithms in C++, and conduct experiments on a machine with two Intel Xeon 2.1GHz processors with 24 cores and 251 GB of memory, running CentOS 7.

Datasets and Queries. We test both graph and relational datasets/queries. All queries can be found in the version [1]. We use the Epinions dataset that contains 508,837 edges from SNAP [3] as the graph dataset. Each relation contains all edges. We randomly shuffle all edges for each relation to simulate the input stream. On Epinions, we evaluate line-k joins (which find paths in the graph of length k), star-k joins (which find all combinations of k edges sharing a common vertex), and dumbbell join (which find all pairs of triangles that are connected by an edge). No foreign-key join is in graph queries. We use two relational datasets. One is the TPC-DS [4], which focuses on a decision support system. We evaluate the same QX, QY, and QZ queries as [19] on TPC-DS, which include the foreign-key joins, and follow the same setup as [19], such that small dimension tables (such as date_dim and household_demographics are pre-loaded while remaining tables are loaded in a streaming fashion. The other is LDBC-SNB [5], which focuses on join-heavy complex queries with updates. We tested Q10 query from the BI workload 10. Similar to before, the static tables (such as tag and city) are pre-loaded, and the dynamic tables are loaded in a streaming fashion.

5.2 Experiment Results

Runtime. Figure 7 shows the runtime of all algorithms on tested queries. For graph queries (i.e., line-k, star-k, and dumbbell),the sample size is 100,000. For relational queries (i.e., QX, QY, QZ, and Q10), the sample size is 1,000,000. For the TPC-DS dataset, we use a scale factor of 10, while for the LDBC-SNB dataset, we use a scale factor of 1. Firstly, RSJoin and RSJoin_opt can finish all queries within 12-hour time limit while SJoin cannot finish on the line-5 join and the QZ join. For the dumbbell join, the result is missing for SJoin since it does not support cyclic queries. Secondly, RSJoin is always the fastest over all join queries. Based on existing results, RSJoin achieves a speedup ranging from 4.6x to 147.6x over SJoin, not mention the case when SJoin cannot finish in time. When foreign-key join exists (i.e., QX, QY, QZ, and Q10), RSJoin_opt achieves an improvement of 2.2x to 4.7x over SJoin_opt. Furthermore, for QX, QY, QZ, and Q10, RSJoin does not heavily rely on foreign-key optimizations as SJoin. As long as data satisfies foreign-key constraints, RSJoin finishes the execution within a reasonable amount of time, but this is not the case for SJoin.

Update time. To compare the update time, we disable the sampling part of both algorithms and measure the update time required for each input tuple. Figure 8 shows the result on line-4 join. Most of the update time required is roughly 10 µs, with an average of 13 µs. Some tuples may incur a much larger update time (51 ms in this case), but the overall update time remains small, which aligns with our theoretical analysis of $O(\log N)$ amortized cost. In contrast, there is no guarantee on the update time for SJoin, and its update time ranges from 0.5 µs to 165 ms, with an average of 1.4 ms.

Input size and Join size. We next investigate how the input size N as well as the join size (i.e., the number of join results) affect the total execution time of all methods. We fix the sample size k to be $10,000$ and record the total execution after every 10% of input data is processed for line-3 join. Figure 9 shows the progress of total number of join results generated and the total execution time. We can see that the total number of join results grows exponentially with the input size, while the total execution time of RSJoin scales almost linearly proportional to the input size, instead of the join size. This is expected as the time complexity of RSJoin is $O(N \cdot \log N + k \cdot \log N \cdot \log \frac{N}{k})$, where the term $N \log N$ almost dominates the total execution time in this case. In contrast, the total execution time of SJoin shows a clear increase trend together with the increase in the join size, which is much larger than the input size.

Sample size. We next study how the sample size k affects the total execution time of both algorithms. Figure 10 shows the runtime on line-3 join, when k varies from 10,000 to 5,000,000. The dashed line indicates the input size $N = 508,837$ and the number of join results is 3,721,042,797. When the sample size is smaller than the input size, i.e., $k \leqslant N$, the total execution time of RSJoin grows very slowly. More specially, when k increases from 1 to 50, the total execution time of RSJoin only increases by a factor of 2. However, when the sample time overrides the input size, i.e., $k > N$, the total execution time of RSJoin starts to increase rapidly. This is also expected again as the theoretical complexity of RSJoin is $O(N \cdot \log N + k \cdot \log N \cdot \log \frac{N}{k})$. When $k \leqslant N$, the term $O(N \cdot \log N)$ dominates the overall execution time, hence increasing the sample size within this regime does not change the total execution time significantly. When $k > N$, the term $O(k \cdot \log N \cdot \log \frac{N}{k})$ dominates the overall execution time instead, hence increasing the sample size results in a rapid increase in the total execution time. SJoin follows a similar trend. Moreover, when the sample size reaches $k = 10,000$, the runtime required by SJoin is even more than that required by RSJoin for the case when the sample size is as large as $k = 5,000,000$.

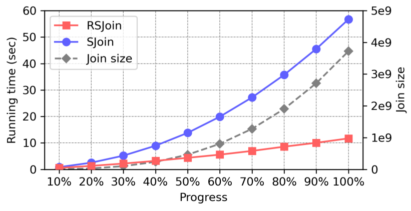


Figure 9: Runtime v.s. input/join size

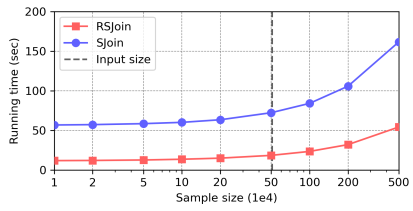


Figure 10: Runtime v.s. sample size

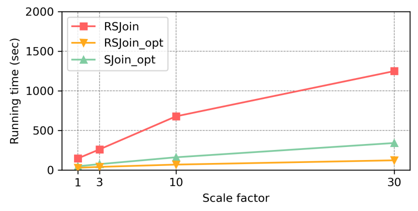


Figure 11: Runtime v.s. scale factor

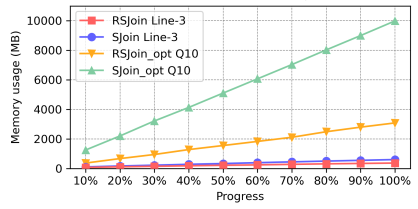


Figure 12: Memory usage v.s. input size

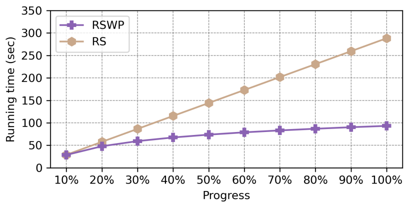


Figure 13: Runtime v.s. input size

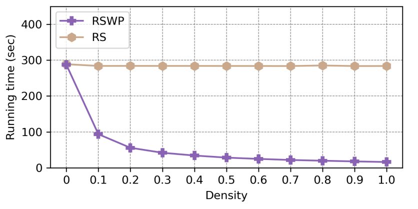


Figure 14: Runtime v.s. density

Scalability. To examine the scalability of both methods, we evaluate the QZ query on the TPC-DS dataset with scale factors of 1,3,10, and 30. The results are shown in Figure 11. The input size of QZ is approximately 226MB when the scale factor is 1, while the input size reaches around 6.6GB when the scale factor reaches 30. We do not include the results of SJoin here since it takes more than 4 hours to finish the execution even with a scale factor of 1. We observe that even without applying foreign-key optimization, RSJoin achieves linear growth in the runtime as the scale factor increases, which indicates that RSJoin is scalable and practical even when dealing with a significantly huge input size.

Memory usage. In addition, we explore the memory usage of all methods. Figure 12 shows the memory usage by RSJoin and SJoin on line-3 join and RSJoin_opt and SJoin_opt on Q10 query. The input size is roughly 21MB for line-3 join and 505MB for Q10 query. After processing every 10% of the input data, we record the memory usage as shown in Figure 12. The memory usage of Q10 grows much faster than line-3 join as it is much more complex with more dedicated index built. The memory usage required by all algorithms is linear to the input size. On line-3 join, RSJoin requires only 60% of the memory by SJoin, and on Q10, RSJoin_opt needs only 31% of the memory by SJoin_opt. This demonstrates a nice property of our algorithm: the amount of memory used by RSJoin and RSJoin_opt during execution scales linearly with the input size even when the join size grows exponentially, which also enables our algorithm to handle much more complex queries over large input datasets with limited memory resources.

Reservoir Sampling with Predicate. We compare reservoir sampling with predicate by our new algorithm (denoted as RSWP) with the naive algorithm (denoted as RS) on data streams. We generate a data stream as follows. We fix a random string of 1024 characters, referred to as the query string. Each item in the input stream is a random string, within edit distance ranging from 0 to 64 from the base string. The predicate selects all strings in the stream whose edit distance from the query string is no more than 16.

In Figure 13, we take a $\frac{1}{10}$-dense stream of $100,000$ strings with sample size $k = 1,000$. We record the execution time after processing every 10% of the input stream. As RS needs to process every item (i.e., compute the edit distance from the query string), the runtime of RS is linear to the number of items in the stream processed so far. The time required by RSWP for processing the first 10% of the input stream is the same as RS, since both of them need to process every one in the first 10,000 items (approximately) until it fills the reservoir. After that, the runtime of RSWP grows slower and slower, which is consistent with our theoretical result that it takes $O(\frac{k}{r_i+1})$ expected time to process the i-th item.

In figure 14, we measure the runtime of both RSWP and RS over 11 streams of same input size but different densities. As RS needs to process every item in the stream, its runtime only depends on the input size, instead of the density of input stream. In contrast, the runtime of RSWP depends on the density of stream. In an extreme case, when no item passes the predicate (i.e., the density is 0), RSWP cannot skip any item and hence requires the same time as RS. However, as density increases, the runtime of RSWP decreases significantly. In another extreme case, when every item passes the predicate (i.e., the density is 1.0), RSWP exhibits a speed advantage of 17.7x over RS.

6. CONCLUSION

In this paper, we propose a general reservoir sampling algorithm that supports a predicate. We design a dynamic index that supports efficient updates and direct access to the join results. By combining these two key techniques, we present our reservoir sampling over joins algorithm which runs in near-linear time. There are several interesting questions left open, such as uniform sampling over join-project queries over data streams.

7. REFERENCES

[1] `https://arxiv.org/pdf/2404.03194.pdf`.

[2] `https://en.wikipedia.org/wiki/Symmetric_hash_join`.

[3] `http://snap.stanford.edu/data`.

[4] `https://www.tpc.org/tpcds/`.

[5] `https://ldbcouncil.org/benchmarks/snb/`.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[7] S. Acharya, P. B. Gibbons, V. Poosala, and

S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

[8] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE, 2008.

[9] N. Carmeli, S. Zeevi, C. Berkholz, B. Kimelfeld, and N. Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *PODS*, pages 393–409, 2020.

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

[11] Y. Chen and K. Yi. Random sampling and size estimation over cyclic joins. In *ICDT*, 2020.

[12] S. Deng, S. Lu, and Y. Tao. On join sampling and hardness of combinatorial output-sensitive join algorithms. In *PODS*, 2023.

[13] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM*, 30(3):514–550, 1983.

[14] K. Kim, J. Ha, G. Fletcher, and W.-S. Han. Guaranteeing the $\tilde{\mathrm{O}}$(agm/out) runtime for uniform sampling and size estimation over joins. In *PODS*, page 113–125, 2023.

[15] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[16] K.-H. Li. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *TOMS*, 20(4):481–493, 1994.

[17] J. S. Vitter. Random sampling with a reservoir. *TOMS*, 11(1):37–57, 1985.

[18] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539, 2018.

[19] Z. Zhao, F. Li, and Y. Liu. Efficient join synopsis maintenance for data warehouse. In *SIGMOD*, pages 2027–2042, 2020.