Reservoir Sampling over Joins

BINYANG DAI[*], Hong Kong University of Science and Technology, China
XIAO HU[*], University of Waterloo, Canada
KE YI, Hong Kong University of Science and Technology, China

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.

CCS Concepts: • **Theory of computation → Sketching and sampling**; • **Information systems → Join algorithms**.

Additional Key Words and Phrases: Data Stream, Acyclic Join, Uniform Sample

ACM Reference Format:
Binyang Dai, Xiao Hu, and Ke Yi. 2024. Reservoir Sampling over Joins. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 118 (June 2024), 26 pages. https://doi.org/10.1145/3654921

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. In as early as 1999, two prominent papers [6, 13] asked the intriguing question, whether a sample can be obtained without computing the full join. As observed in by [13], 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 index structures to guide the sampling process. Notably, an index was proposed for *acyclic* joins (formally defined in Section 4) that can be built in

[*]Both authors contributed equally to this research.

Authors' addresses: Binyang Dai, bdaiab@connect.ust.hk, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong, China; Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1; Ke Yi, yike@ust.hk, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong, China.

ACM 2836-6573/2024/6-ART118
https://doi.org/10.1145/3654921

$O(N)$ 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 [12, 30]. Please see Section 2.2 for a more comprehensive review on the sampling complexity for different join queries.

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 re-build the index and re-draw the samples after each tuple has arrived, but this results in a total running time of $O(N^2)$ to process a stream with N tuples. Recently, Zhao et al. [31] applied the reservoir sampling algorithms [24, 27] to this problem to update the sample incrementally. However, their index also suffers from a high maintenance cost that still leads to a total running time 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 running time 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 running time 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 a predicate. The classical reservoir sampling algorithm, attributed to Waterman by Knuth [22], 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 [24, 27]. 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 the number of items in the first $i-1$ items that pass the predicate. Note that when $\theta(\cdot) \equiv \mathtt{true}$, 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 are more difficult, as it is not safe to skip items. 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 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 [7] 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 (details provided in Section 5). 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 moreover it is possible to skip its items without counting them one by one.

Dynamic sampling from joins. Let Q be an acyclic join query, $\mathcal{R}$ a database instance of size N, and $Q(\mathcal{R})$ the join results of 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 $Q(\mathcal{R})$ plus some dummy tuples, but it is guaranteed that $|J| = O(|Q(\mathcal{R})|)$, 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 $\Delta Q(\mathcal{R}, t) := Q(\mathcal{R} \cup \{t\}) - Q(\mathcal{R})$ 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 $|J| = O(|Q(\mathcal{R})|)$, 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 [12, 30], 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 [31], 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 $\Delta Q(\mathcal{R}, t)$. If we can access any tuple in $\Delta Q(\mathcal{R}, t)$ 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 $\Delta Q(\mathcal{R}, t)$ 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.

The contributions of this paper are thus summarized as follows:

- **(Section 3)** We formulate the problem of reservoir sampling with a predicate. Assuming skip takes $O(1)$ time, we design an algorithm that can maintain a sample (without replacement) of size k in time $O\left(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i+1}\right)\right)$, which we also show is instance-optimal.
- **(Section 4)** We present a dynamic index for acyclic joins that can be updated in $O(\log N)$ amortized time, and return a sample from either the full query or the delta query in $O(\log N)$ time. Combined with our reservoir sampling with predicate algorithm, we show how the reservoir sampling over join problem can be solved in time $O(N \log N + k \log N \log \frac{N}{k})$. We show how our algorithm can be optimized when key constraints are present.
- **(Section 5)** We extend the algorithm to cyclic joins using the generalized hypertree decomposition technique. In this case, the running time becomes $O(N^{\mathrm{w}} \log N + k \log N \log \frac{N}{k})$, where w is the fractional hypertree width of the query.
- **(Section 6)** We implement our algorithm and evaluate it over both graph and relational data. The experimental results show that our algorithm significantly outperforms the state-of-the-art solution [31].

2 PRELIMINARIES

2.1 Problem Definition

We first recap some standard notation in relational algebra [5]. A multi-way (natural) join query can be defined as a hypergraph $Q = (\mathcal{V}, \mathcal{E})$, 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 $\mathrm{supp}(t)$ 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 $x \subseteq \mathrm{supp}(t)$, the projection $\pi_x t$ is the value of tuple t on attribute(s) x. The join results of Q over instance $\mathcal{R}$, denoted by $Q(\mathcal{R})$, is the set of all combinations of tuples, one from each R_e, that share common values for their common attributes, i.e.,

$$Q(\mathcal{R}) = \left\{ t \in \prod_{x \in \mathcal{V}} \mathrm{dom}(x) \mid \forall e \in \mathcal{E}, \exists t_e \in R_e, \pi_e t = t_e \right\}. \tag{1}$$

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 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 attribute(s) $e \cap e'$ with at least one tuple from $R_{e'}$. Note that for a join query Q, the delta query $\Delta Q(\mathcal{R}, t)$ is equal to $Q(\mathcal{R} \cup \{t\}) \ltimes t$.

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: The first is the sampling over join problem as studied in [6, 12–15, 21, 30]. This is an indexing (data structure) problem where we wish to have an index that supports drawing a sample from $Q(\mathcal{R}^i)$. 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 [24, 27, 31], where we wish to maintain k random samples from $Q(\mathcal{R}^i)$ without replacement for every $Q(\mathcal{R}^i), i \in \mathbb{Z}^+$. For this problem, we just care about the total running time. 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. Note that the classical reservoir sampling only uses $O(k)$ space, but sub-linear space is not possible when Q has joins. Just consider a two-table join $Q := R_1(X, Y) \bowtie R_2(Y, Z)$. Suppose the first N tuples in the stream are all in R_1. The algorithm must keep all of them in memory, otherwise it will miss the first join result, which must be sampled, when some tuple in R_2 arrives.

We follow the convention of data complexity [5] and analyze the running time in terms of the input size N and sample size k, while taking the size of 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 we assume that duplicates have been removed from the input stream.

2.2 Previous Results

Sampling over joins. Chaudhuri et al. [13] showed, for the basic two-table join $R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$, how to construct an index structure in $O(N)$ time, such that a sample can be drawn in $O(1)$ time. Acharya et al. [6] achieved the same complexity result for multi-way joins but all joins are restricted to foreign-key joins. These results have been later extended to all acyclic joins [12, 30]. All these sampling indexes on acyclic joins are static. For cyclic joins, there is an index that can be built in $O(N)$ time, while a sample can be drawn in $O\left(\frac{N^{\rho^*+1}}{|Q(\mathcal{R})|}\right)$ time [14], which has recently been improved to $O\left(\frac{N^{\rho^*}}{|Q(\mathcal{R})|}\right)$ [15, 21], who also make the index dynamic, but note that the sampling time for cyclic joins is significantly higher than that for acyclic joins.

Reservoir sampling over joins. When Q has no joins, the classic reservoir sampling algorithm [22] solves the problem in $O(N)$ time. Assuming that skip takes $O(1)$ time, this can be reduced to $O(k \log \frac{N}{k})$ [24, 27]. Zhao et al. [31] investigated the problem over acyclic joins, and proposed some efficient heuristics. But their solution takes $O(N^2)$ time in the worst case, which is the same as the naive solution that rebuilds the static join sampling index [12, 30] at each time step.

Hardness results. In this paper we will focus on acyclic joins over an insertion-only stream. Both restrictions turn out to be necessary for achieving a near-linear running time, following some existing hardness results. The observation is that sampling a query is at least as hard as the corresponding Boolean query (i.e., determining whether $Q(\mathcal{R}) = \emptyset$): We can just return true for the Boolean query if there is any sample returned from the sampling algorithm, and false otherwise. It is known that it requires $\Omega(N^{\mathrm{w}})$ time to compute a Boolean cyclic query, for some width parameter $\mathrm{w} > 1$ of the query [8]. So for a cyclic query, there is no hope for taking a sample of the join in near-linear time, even over a static database. Meanwhile, for a general (more precisely, a non-hierarchical) acyclic query, it is known that the update time must be $\Omega(\sqrt{N})$ just to maintain the Boolean answer, when both insertions and deletions are allowed [10]. This means that the reservoir sampling problem requires at least $\Omega(N^{1.5})$ time over a fully-dynamic stream.

3 RESERVOIR SAMPLING WITH PREDICATE

3.1 Reservoir Sampling Revisited

Reservoir sampling [24, 27] is a family of algorithms for maintaining a random sample, without replacement, of k items from a possibly infinite stream. The classical version, as described in [22], 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 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 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 [24]. 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 $\mathrm{Geo}(w)$, 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 [24] 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. 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 [24] to Algorithm 1. In the description, we use the following two 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 [24], we have made two changes: (line 2-5) when the reservoir is not full, we only add real items to it; (line 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, given in the full version [2], is quite technical. We only provide some intuition here. For each item x in the stream, Algorithm 1

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     $q \leftarrow \lfloor(\ln(\mathsf{rand}()/\ln(1-w))\rfloor$; (note that $q \sim \mathsf{Geo}(w)$)
```

generates a random variable from $\mathsf{Uni}(0,1)$, say u_x, and executes line 11-14 for x if $u_x < w$, since the next random variable generated from $\mathsf{Uni}(0,1)$ being smaller than w follows the geometric distribution parameterized by w. We can further exchange the two if-conditions ($w < u$ and x is real) since these two events are independent. Then, one can show that Algorithm 1 is equivalent to feeding only the real items to the non-predicate version of the algorithm in [24].

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 3.1. *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 time complexities of* $\mathsf{next}(\cdot)$ *and* $\mathsf{skip}(\cdot)$*, respectively.*

Proof. For the ease of analysis, we add one additional real item as a sentinel object to the end of the stream. We will ignore the cost for this real item later. Let S' denote the set of items with index larger than or equal to p. We analyze the number of invocations of $\mathsf{skip}(\cdot)$ by Algorithm 1. Each time Algorithm 1 calls $\mathsf{skip}(\cdot)$ and returns some item x, we say it stops at x. Note that Algorithm 1 only stops at items that are from S'. We consider an equivalent version for the ease of analysis: for each item, we generate a random variable from $\mathsf{Uni}(0,1)$, say u, and execute line 11 - 14 only for successful trials (i.e. $u < w$). The number of stops is the same as the number of successful trials. If w_i is the value of w when processing item x_i, then the probability that Algorithm 1 stops at x_i is exactly w_i for $i \geqslant p$. Let $\mathbf{w} = \langle w_1, w_2, \cdots, w_{N+1}\rangle$ be the state of Algorithm 1. For any w, we observe that (1) $w_i \in [0,1]$ if $i \in [1 \ldots N+1]$; (2) $w_i \geqslant w_j$ if $i < j$; (3) $w_i = 1$ if $i < p$; (4) $w_i = w_j$ if the j-th item is real and the i-th item to the $(j-1)$-th item are all dummy. For (4), we use function $\pi(i)$ to denote the smallest index j where $j \geqslant i$ such that x_j is real. Note that $w_i = w_{\pi(i)}$ for $i \in [p \ldots N+1]$. Let W be the set of all possible states of Algorithm 1. The expected number of stops in S' is

$$\begin{aligned}\mathbb{E}[\#\text{stops in } S'] &= \sum_{\mathbf{w}\in W} \Pr(\mathbf{w}) \cdot \mathbb{E}[\#\text{stops in } S' | \mathbf{w}] \\ &= \sum_{\mathbf{w}\in W} \Pr(\mathbf{w}) \cdot \sum_{i=p}^{N} w_{\pi(i)} = \sum_{i=p}^{N} \sum_{\mathbf{w}\in W} \Pr(\mathbf{w}) \cdot w_{\pi(i)} = \sum_{i=p}^{N} \frac{k}{r_i + 1}\end{aligned}$$

where the rationale behind the last equality is that $\sum_{\mathbf{w}\in W} \Pr(\mathbf{w}) \cdot w_{\pi(i)}$ is exactly the probability that $x_{\pi(i)}$ enters the reservoir ever. Next, we move to the next() primitive. For while loop in line 2, we keep adding real items into the reservoir until it becomes full. So, the number of invocations of the next() is exactly $p - 1$. Putting everything together, we complete the proof. □

Note that in the degenerate case where all items are real, we have $r_i = i - 1$ and the running time of Algorithm 1 becomes $O(k \log \frac{N}{k})$ (when taking α, γ as $O(1)$), matching the optimal reservoir sampling running time [24, 27]. In the other extreme case, all items are dummy, so $p = N + 1$ and $r_i = 0$, and the running time 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 3.2. *For any input stream S of N elements, any algorithm that can maintain a uniform sample of size k over all real elements must run in $\Omega\left(\sum_{i=1}^{N} \min\{1, \frac{k}{r_i+1}\}\right)$ expected time.*

Proof. Consider an arbitrary input stream S and an arbitrary $i \in [N]$. Any correct algorithm for maintaining a uniform sample over real elements at timestamp i must stop at x_i with probability at least $\min\{1, \frac{k}{r_i+1}\}$. Recall that any algorithm cannot distinguish whether an element is real or dummy until it stops (and checks). Suppose x_i is real. If the probability is smaller than $\min\{1, \frac{k}{r_i+1}\}$, then the probability that x_i enters into the reservoir must be smaller than $\min\{1, \frac{k}{r_i+1}\}$, which contradicts the fact that this algorithm can return a uniform sample at timestamp i. A correct algorithm for maintaining a uniform sample of size k over all real elements in the stream must be correct at timestamp i for $1 \leqslant i \leqslant N$. Hence, any correct algorithm must stops at $\sum_{i=1}^{N} \min\{1, \frac{k}{r_i+1}\}$ expected numbers. □

Although the running time of Algorithm 1 can vary significantly from $O(k \log \frac{N}{k})$ to $O(N)$, but it is closer to the former as long as the stream is dense enough.

Definition 3.3 (Dense stream). 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.

Combining Theorem 3.1 and Definition 3.3 we obtain:

Corollary 3.4. *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 3.5 implies that straightforwardly concatenating two streams still preserves their minimum density of real items. If one stream only consists of dummy items, it is possible to get a better bound on the density of real items in the whole stream, which is essentially captured by Lemma 3.7. The more dummy items padded, the sparser the stream becomes. Lemma 3.6 implies that mixing two streams as their Cartesian product preserves a density that is at least half of their density product. For the ease of notation, we denote $q_i = r_{i+1}$ (i.e., the number of real items in the first i items) in the following proofs of Lemma 3.5, Lemma 3.6, and Lemma 3.7. It is easy to see that a stream S is ϕ-dense if $q_i \geqslant \phi \cdot i$ for all i.

Lemma 3.5. *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.*

Proof. Consider the stream $S_1 \circ S_2$. As S_1 is ϕ_1-dense, we have $q_i \geqslant \phi_1 \cdot i$ for every $i \in [1 \ldots m]$. Moreover, $q_m \geqslant \phi_1 \cdot m$. As S_2 is ϕ_2-dense, we have $q_j - q_m \geqslant \phi_2 \cdot (j - m)$ for every $m \leqslant j \leqslant m + n$. Hence, we obtain

$$q_j \geqslant \phi_2 \cdot (j - m) + q_m \geqslant \phi_2 \cdot (j - m) + \phi_1 \cdot m \geqslant \min\{\phi_1, \phi_2\} \cdot j$$

for every $m \leqslant j \leqslant m + n$. Together with the fact $q_j \geqslant \phi_1 \cdot j \geqslant \min\{\phi_1, \phi_2\} \cdot j$ for every $j \in [1 \ldots m]$, we have proved that $q_j \geqslant \min\{\phi_1, \phi_2\} \cdot j$ for every $j \in [1 \ldots m + n]$. □

Lemma 3.6. *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.*

Proof. Consider the stream $S_1 \times S_2 = \langle z_1, z_2, \ldots z_{mn} \rangle$ and an arbitrary $1 \leqslant i \leqslant mn$. Assume that $m > 0$ and $n > 0$. Let $i_1 = \lfloor \frac{i}{n} \rfloor$ and $i_2 = i - i_1 \cdot n$. Let $\langle z_{jn+1}, z_{jn+2}, \ldots z_{jn+n} \rangle$ be the row_j, where $j = 0, 1, \ldots i_1 - 1$. Then row_j contains at least $\phi_2 \cdot n$ real items if x_{j+1} in S_1 is real. Otherwise, all the n items in row_j are dummy. Let q_i be the number of items that are real in the first i items of the resulted stream. We distinguish the following 2 cases:

- $i_1 = 0$: Since S_1 is ϕ_1-dense, the first item of S_1 must pass predicate θ. Then in this case, $q_i \geqslant \phi_2 \cdot i$ as S_2 is ϕ_2-dense.
- $i_1 > 0$: The first i items of the resulted stream can be represented as i_1 rows followed by i_2 items. In the i_1 rows, there are at least $\phi_1 \cdot i_1$ rows each contains at least $\phi_2 \cdot n$ items that are real as S_1 is ϕ_1-dense. In total, there are at least $\phi_1 \cdot \phi_2 \cdot n \cdot i_1$ item that are real. As $i \leqslant (i_1 + 1) \cdot n$, we have

$$q_i \geqslant \frac{\phi_1 \cdot \phi_2 \cdot n \cdot i_1}{i} \cdot i \geqslant \frac{\phi_1 \cdot \phi_2 \cdot n \cdot i_1}{(i_1 + 1) \cdot n} \cdot i \geqslant \phi_1 \cdot \phi_2 \cdot \frac{i_1}{i_1 + 1} \cdot i \geqslant \frac{\phi_1 \cdot \phi_2}{2} \cdot i$$

As $0 < \phi_1 \leqslant 1$, $q_i \geqslant \phi_2 \cdot i \geqslant \frac{\phi_1 \cdot \phi_2}{2} \cdot i$ for the case $i_1 = 0$. Putting all together, we have $q_i \geqslant \frac{\phi_1 \cdot \phi_2}{2} \cdot i$. □

Lemma 3.7. *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.*

Proof. Let S, S' be the original and resulted stream respectively. Let $|S| = m$ and $|S'| = m + n$, i.e., padding n dummy items at the end of S. It suffices to prove $q_i \geqslant \phi \cdot \frac{m}{m+n} \cdot i$ for any $m + 1 \leqslant i \leqslant m + n$. As S is ϕ-dense, we note that $q_m \geqslant \phi \cdot m$. We have

$$q_i = q_m \geqslant \phi \cdot m \geqslant \phi \cdot \frac{m}{m+n} \cdot (m + n) \geqslant \phi \cdot \frac{m}{m+n} \cdot i$$ □

3.3 Batched Reservoir Sampling with Predicate

As described in Section 1, each arriving tuple t generates a batch of new join results $\Delta Q(\mathcal{R}, t)$. To apply our reservoir sampling algorithm over joins, 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 reservoir sampling is described in Algorithm 2, which calls BatchUpdate (Algorithm 3) for every batch. BatchUpdate essentially runs Algorithm 1 on the

```
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  $S \leftarrow \emptyset$, $w \leftarrow +\infty$, $q \leftarrow 0$;
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      $w \leftarrow \text{rand}()^{1/k}$;
7      $q \leftarrow \lfloor(\ln(\text{rand}())/\ln(1-w))\rfloor$; (note that $q \sim \text{Geo}(w)$)
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         $w \leftarrow w \cdot \text{rand}()^{1/k}$;
14     $q \leftarrow \lfloor(\ln(\text{rand}()/\ln(1-w))\rfloor$; (note that $q \sim \text{Geo}(w)$)
15 return $S, w, q - B$.remain();
```

given batch B, but it must guard against the case where a skip(q) 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, since the value of w is always no larger than 1 after initialization.

The samples maintained by Algorithm 2 are exactly the same as that maintained by Algorithm 1 over items in batches, so correctness follows immediately. Below we analyze its running time.

Algorithm 4: ReservoirJoin(Q, D, k)

```
Input  : A join query $Q$, an input stream $D$ of tuples, and the target number of samples $k$;
Output : A set $S$ maintaining $k$ random samples without replacement for the join results of
         $Q$ over tuples seen as far;
1 Initialize index $\mathcal{L}$, $S \leftarrow \emptyset$, $w \leftarrow +\infty$, $q \leftarrow 0$, $\theta \leftarrow$ 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$(Q, \mathcal{L}, t)$;
7     $(S, w, q) \leftarrow$ BatchUpdate$(S, k, w, q, B, \theta)$;
```

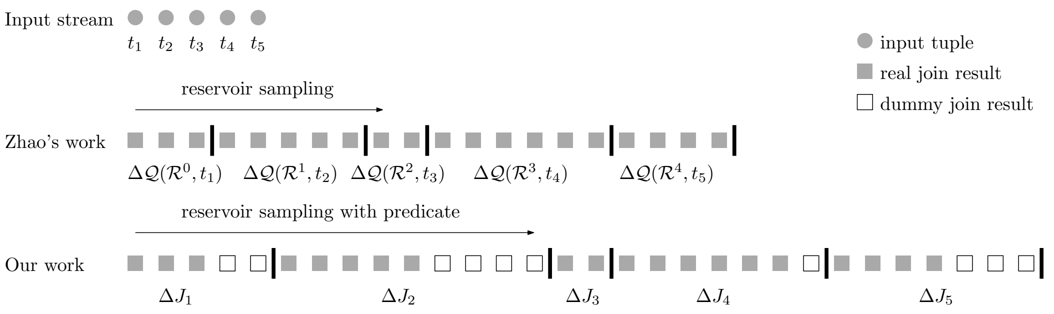

Input stream
t_1 t_2 t_3 t_4 t_5
● input tuple
▨ real join result
□ dummy join result
reservoir sampling
Zhao's work
$\Delta Q(\mathcal{R}^0, t_1)$ $\Delta Q(\mathcal{R}^1, t_2)$ $\Delta Q(\mathcal{R}^2, t_3)$ $\Delta Q(\mathcal{R}^3, t_4)$ $\Delta Q(\mathcal{R}^4, t_5)$
reservoir sampling with predicate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

Fig. 1. An illustration of reservoir sampling over join.

Theorem 3.8. *Given an input stream of 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 over a stream of m item-disjoint batches, where N is the total size of items in all batches, and α, β, γ are the time complexities of* next$(\cdot)$, remain$(\cdot)$, skip$(\cdot)$ *respectively.*

Proof. Running Algorithm 2 on a stream of batches containing ϕ-dense streams is essentially the same as running Algorithm 1 on the concatenation of the ϕ-dense streams. The numbers of invocation to next$(\cdot)$ and skip$(\cdot)$ in Algorithm 2 are the same as in Algorithm 1. Note that each call to remain$(\cdot)$ is immediately followed by a call to next$(\cdot)$ or skip$(\cdot)$ except for line 15. In line 15, the return value of remain$(\cdot)$ must be the same as the last call to remain$(\cdot)$ in line 8. Hence, we can store it in a variable and eliminate the invocation in line 15. The last $O(m)$ term comes from the fact that Algorithm 2 makes m calls to BatchUpdate in total. Then, it follows Corollary 3.4. □

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, we invoke procedure BatchGenerate to conceptually generate a batch $\Delta Q(\mathcal{R}, t)$ 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 (details provided in Section 5). Instead, we will maintain a linear-size index $\mathcal{L}$, which support a *retrieve* operation that returns the item at position z in $\Delta Q(\mathcal{R}, t)$ for any given z. The index should also be able to return N_B, the size of the batch. 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 $N_B - \text{pos}$, 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 Q, it suffices to show how to maintain a linear-size index $\mathcal{L}$ that can efficiently support the retrieve operation for each $\Delta Q(\mathcal{R}, t)$, as well as $|\Delta Q(\mathcal{R}, t)|$. This is still hard. To get around this difficulty, in Section 4 we devise an approximate solution. Our index $\mathcal{L}$ will implicitly define a ΔJ that contains all the join results in $\Delta Q(\mathcal{R}, t)$, 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 predicate. We set the predicate θ to 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). The details of these dummy join results will be clear in Section 4.2 and Section 4.3. Finally, we will also guarantee that each ΔJ is dense so as to apply Theorem 3.8.

Comparison with [31]. The algorithm in Zhao et al. [31] in fact follows the same framework, but they simply used the classical reservoir sampling algorithm without predicate. As such, they must use an index $\mathcal{L}$ that supports the retrieve operation and the size information directly on $\Delta Q(\mathcal{R}, t)$; please see Figure 1 for an illustration. 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 $\mathcal{L}$ filled with dummy join results, which can be updated in $O(\log N)$ time as shown in the next section.

4 SAMPLING OVER ACYCLIC JOINS

We first recall the definition of an acyclic join:

Definition 4.1 (Acyclic Join [9]). A (natural) join $Q = (\mathcal{V}, \mathcal{E})$ is α-acyclic if there exists a tree $\mathcal{T}$ (called the join tree) such that (1) there is a one-to-one correspondence between the relations in $\mathcal{E}$ and nodes in $\mathcal{T}$; and (2) for every attribute $X \in \mathcal{V}$, the set of nodes in $\mathcal{T}$ containing X are connected in $\mathcal{T}$.

In this section, we present a dynamic index that can achieve the following guarantees:

Theorem 4.2. *Given any acyclic join Q, an initially empty database $\mathcal{R}$, and a stream of N tuples, we can maintain an index $\mathcal{L}$ on $\mathcal{R}$ using $O(N)$ space while supporting the following operations:*

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

(2) The index implicitly defines an array $J \supseteq Q(\mathcal{R})$ where the tuples in $Q(\mathcal{R})$ are the real tuples and the others are dummy. The index can return $|J|$ in $O(1)$ time. For any given $j \in [|J|]$, it can return $J[j]$ in $O(\log N)$ time. Furthermore, J is guaranteed to be ϕ-dense for some constant $0 < \phi \leqslant 1$.

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

This index (using operation (1) and (2) above) immediately solves the dynamic sampling over join problem with an update time of $O(\log N)$ and sampling time $O(\log N)$. Thanks to operation (3) and the density guarantee, it also solves the reservoir sampling over join problem by plugging into Theorem 3.8 with $\alpha = \gamma = O(\log n)$ and $\beta = O(1)$. The number of batches is $m = N$, while the N in Theorem 3.8, which corresponds to the stream of join results now, becomes N^{ρ^*}. Note that ρ^* only depends on the query and not the input size, so it is taken as a constant.

Corollary 4.3. *Given any acyclic join 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 $Q(\mathcal{R}^i)$, and runs in $O(N \log N + k \log N \log \frac{N}{k})$ expected time.*

In fact, operation (2) can be reduced to operation (3): The J for $Q(\mathcal{R})$ is just the concatenation of all the ΔJ's of the delta queries. The size of J is the sum of all the $|\Delta J|$'s, which can be easily maintained in $O(1)$ time. The concatenated J is still dense as long as each ΔJ is dense, due to Lemma 3.5. Henceforth we will only focus on operation (1) and (3).

4.1 Two-table Join

We start by considering the simple two-table join $R_1(X,Y) \bowtie R_2(Y,Z)$. For this query, the index simply consists of two arrays $R_1 \ltimes b$ and $R_2 \ltimes b$, as well as their sizes, for every $b \in \text{dom}(Y)$. The size of two arrays $R_1 \ltimes b$ and $R_2 \ltimes b$ is $|R_1 \ltimes b|$ and $|R_2 \ltimes b|$ respectively. Summing over all $b \in \text{dom}(Y)$, the whole index uses $O(N)$ space. Then operation (1) can be easily supported in $O(1)$ time: We just add the tuple t to $R_1 \ltimes t.Y$ if $t \in R_1$, or $R_2 \ltimes t.Y$ if $t \in R_2$. For operation (3), suppose $t \in R_1$. We set $\Delta J = R_2 \ltimes t$. Clearly, J is 1-dense as there are no dummy tuples, and any $J[j]$ can be retrieved in $O(1)$ time.

4.2 Line-3 Join

When we move to the line-3 join $R_1(X,Y) \bowtie R_2(Y,Z) \bowtie R_3(Z,W)$, the situation becomes more complicated. However, 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 Structure. For each $b \in \pi_Y R_1$, we maintain the degree of b in R_1, i.e., $\text{cnt}(b) = |R_1 \ltimes b|$ and its approximation $\tilde{\text{cnt}}(b) = 2^{\lceil \log_2 \text{cnt}(b) \rceil}$ by rounding $\text{cnt}(b)$ up to the nearest power of 2. Similarly, we maintain $\text{cnt}(c) = |R_3 \ltimes c|$ and $\tilde{\text{cnt}}(c) = 2^{\lceil \log_2 \text{cnt}(c) \rceil}$ for each $c \in \pi_Z R_3$. Note that $\tilde{\text{cnt}}(\cdot)$ 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

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

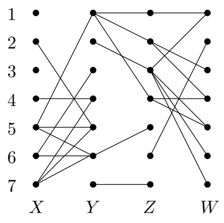
Let $\mathcal{L}_b$ be the list of non-empty buckets. Define $\varphi_i(b) = 2^i \cdot |\Phi_i(b)|$. We also maintain $N_b = \sum_{i \in [\log N]} \varphi_i(b)$ for each value $b \in \pi_Y R_2$, which is an upper bound on the number of new join results if some tuple (a,b) is added to R_1. Symmetrically,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

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 $O(N)$. 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 $i = \log_2 \tilde{\text{cnt}}(c)$. This just takes $O(1)$ time.

If $t = (a,b) \in R_1$ (the $t \in R_3$ case is similar), we increase $\text{cnt}(b)$ by 1, and update $\tilde{\text{cnt}}(b)$ if needed. If $\tilde{\text{cnt}}(b)$ 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 $i = \log_2 \tilde{\text{cnt}}(b)$. This may take $O(N)$ time, but this update is only triggered when $\tilde{\text{cnt}}(b)$ doubles, which happens at most $O(\log N)$ times. Thus, the total update cost is

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



Database of line-3 join

b	$\mathcal{L}_b$	$\psi_i(b)$	N_b
1	$i=0$ $(1,1)$	1	5
	$i=1$ $(1,2)\,(1,4)$	4	
2	$i=2$ $(2,3)$	4	4

c	$\mathcal{L}_c$	$\psi_i(c)$	N_c
1	$i=0$ $(1,1)$	1	1
2	$i=0$ $(1,2)$	1	1
4	$i=0$ $(1,4)$	1	1
5	$i=2$ $(6,5)$	4	4

(i) Initialization of data structures built for relation R_2

b	$\mathcal{L}_b$	$\psi_i(b)$	N_b
1	$i=0$ $(1,1)$	1	5
	$i=1$ $(1,2)\,(1,4)$	4	
2	$i=1$ $(2,2)$	2	6
	$i=2$ $(2,3)$	4	

(ii) After inserting (2,2) into R_2

b	$\mathcal{L}_b$	$\psi_i(b)$	N_b
1	$i=0$ $(1,1)$	1	7
	$i=1$ $(1,4)$	2	
	$i=2$ $(1,2)$	4	
2	$i=2$ $(2,2)\,(2,3)$	8	8

(iii) After inserting (2,5) into R_3

b	$\mathcal{L}_b$	$\psi_i(b)$
1	$i=0$ $(1,1)$	1
	$i=1$ $(1,4)$	2
	$i=2$ $(1,2)$	4

$j=0, \ell=1$

(iv) Retrieve at position 4 from the batch of $(2,1) \in R_1$

pos	item
0	$(2,1,1,1)$
1	$(2,1,4,4)$
2	$(2,1,4,5)$
3	$(2,1,2,3)$
4	$(2,1,2,4)$
5	$(2,1,2,5)$
6	$\perp$

(v) (Conceptual) batch generated for $(2,1) \in R_1$

Fig. 2. An illustration of index structure built for line-3 join $R_1(X,Y) \bowtie R_2(Y,Z) \bowtie R_3(Z,W)$.

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 $\Phi_i(c)$.

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

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

The batch $\Delta J \supseteq \Delta Q(\mathcal{R}, t)$ 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 $|\Delta J| = \mathrm{cnt}(b) \cdot \mathrm{cnt}(c)$.

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 $\tilde{\mathrm{cnt}}(c) - \mathrm{cnt}(c)$ dummy tuples. We concatenate these mini-batches to form the batch for the bucket, and concatenate all the buckets to form ΔJ. This ΔJ is $\frac{1}{2}$-dense, since each mini-batch is $\frac{1}{2}$-dense and then we invoke Lemma 3.5. 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

$$\sum_{i' \leqslant i-1 : \Phi_{i'}(b) \neq \emptyset} \varphi_{i'}(b) < z + 1 \leqslant \sum_{i' \leqslant i : \Phi_{i'}(b) \neq \emptyset} \varphi_{i'}(b).$$

- Set $j = \left\lfloor \left(z - \sum_{i' \leqslant i-1 : \Phi_{i'}(b) \neq \emptyset} \varphi_{i'}(b)\right) / 2^i \right\rfloor$.

- Set $\ell = z - \sum_{i' \leqslant i-1: \Phi_{i'}(b) \neq \emptyset} \varphi_{i'}(b) - 2^i \cdot j$.

Let t' be the tuple at position j in $\Phi_i(b)$. Then we return the tuple at position ℓ in $R_3 \ltimes t'$ if $\ell < |R_3 \ltimes t'|$, and a dummy tuple otherwise. As there are at most $O(\log N)$ distinct i's with $\Phi_i(b) \neq \emptyset$, 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.

4.3 Acyclic Joins

Finally, we generalize the line-3 algorithm to an arbitrary acyclic join $Q = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{T}$ be any join tree of Q. Recall that there is a one-to-one correspondence between nodes in $\mathcal{T}$ and relations in $\mathcal{E}$. Moreover, for every attribute $X \in \mathcal{V}$, all nodes containing X are connected in $\mathcal{T}$. By definition, $\mathcal{T}$ is an unrooted tree, but we can root it by specifying any node as the root r. We will consider all the rooted trees where r ranges over all nodes, and the one with root r will be responsible for generating the batch $\Delta J \supseteq \Delta Q(\mathcal{R}, t)$ for any $t \in R_r$. For example, the line-3 join has one unrooted join tree $R_1 - R_2 - R_3$ but 3 rooted trees. The line-3 algorithm can be conceptually considered as 3 algorithms, each using one rooted tree. Some data structures among them can be shared, but below we will just focus on one rooted tree for conceptual simplicity.

Consider a $\mathcal{T}$ rooted at r. We use p_e to denote the parent of e. For the root r, set $p_r = \emptyset$. Let $\mathsf{key}(e) = e \cap p_e$ be the common attributes shared between e and its parent p_e. Let C_e be the child nodes of node e. For a leaf node e, $C_e = \emptyset$. Let $\mathcal{T}_e$ be the sub-tree below e. With a slight abuse of notation, we also use $\mathcal{T}_e$ to denote the set of relations whose corresponding nodes are in $\mathcal{T}_e$.

Index structure. We store input tuples in a hash table, so that for any $e \in \mathcal{E}$, a subset of attributes $x \subseteq e$ and a tuple $t \in \mathrm{dom}(x)$, we can get the list of tuples $R_e \ltimes t$ in $O(1)$ time. For each node $e \in \mathcal{T}$ and tuple $t \in \pi_{\mathsf{key}(e)} R_e$, we maintain a upper bound $\mathsf{cnt}[\mathcal{T}, e, t]$ on the *degree* of t in $\mathcal{T}_e$, i.e., the number of join results over relations in $\mathcal{T}_e$ whose projection onto attributes $\mathsf{key}(e)$ matches t:

$$\mathsf{cnt}[\mathcal{T}, e, t] = \begin{cases} |R_e \ltimes t| & \text{if } e \text{ is a leaf} \\ \sum_{t' \in R_e \ltimes t} \prod_{e' \in C_e} \tilde{\mathsf{cnt}}\left[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t'\right] & \text{otherwise} \end{cases}$$

Note that this definition depends on $\tilde{\mathsf{cnt}}(\cdot)$, which is recursively defined as $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] = 2^{\lceil \log_2 \mathsf{cnt}[\mathcal{T}, e, t] \rceil}$ by rounding $\mathsf{cnt}[\mathcal{T}, e, t]$ up to the nearest power of 2. We point out an important property of $\tilde{\mathsf{cnt}}(\cdot)$ in Lemma 4.4, which indicates that $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t]$ is a constant-approximation of the degree of t in $\mathcal{T}_e$.

Lemma 4.4. *For a join tree $\mathcal{T}$, node e and tuple $t \in \pi_{key(e)} R_e$, $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant 2^{|\mathcal{T}_e|} \cdot |(\Join_{e' \in \mathcal{T}_e} R_{e'}) \ltimes t|$.*

Proof. We prove it by induction. If e is a leaf node, $\mathsf{cnt}[\mathcal{T}, e, t] = |R_e \ltimes t|$. As $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant 2\mathsf{cnt}[\mathcal{T}, e, t]$, we have $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant 2 \cdot |R_e \ltimes t|$. If e is an internal node, we assume the lemma holds for every child node $e' \in C_e$ and tuple $t' \in \pi_{\mathsf{key}(e')} R_{e'}$. For an arbitrary tuple $t \in \pi_{\mathsf{key}(e)} R_e$, we can bound $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t]$ as

$$\begin{aligned} \tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant 2 \cdot \mathsf{cnt}[\mathcal{T}, e, t] &= 2 \cdot \sum_{t' \in R_e \ltimes t} \prod_{e' \in C_e} \tilde{\mathsf{cnt}}\left[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t'\right] \\ &\leqslant 2 \cdot \sum_{t' \in R_e \ltimes t} \prod_{e' \in C_e} 2^{|\mathcal{T}_{e'}|} \cdot \left|\left(\Join_{e'' \in \mathcal{T}_{e'}} R_{e''}\right) \ltimes \left(\pi_{\mathsf{key}(e')} t'\right)\right| \\ &= 2 \cdot 2^{|\mathcal{T}_e| - 1} \cdot \left|\left(\Join_{e' \in \mathcal{T}_e} R_{e'}\right) \ltimes t\right| \end{aligned}$$

where the last equality follows the intersections property of $\mathcal{T}$. □

```
Algorithm 5: IndexUpdate($\mathcal{T}, e, t, \mathsf{old}$)
  Input  : A join tree $\mathcal{T}$ for $Q$, a node $e$ and tuple $t \in R_e$, an approximate degree old of $t$ in $\mathcal{T}_e$
           before update;
  Output : Updated $\mathsf{cnt}(\cdot)$ and $\tilde{\mathsf{cnt}}(\cdot)$;
1 $t_e \leftarrow \pi_{\mathsf{key}(e)} t$;
2 $\mathsf{new} \leftarrow \prod_{e' \in C_e} \tilde{\mathsf{cnt}}[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t]$;
3 $i' \leftarrow \log_2 \mathsf{old}$ and $i \leftarrow \log_2 \mathsf{new}$;
4 if $i' > 0$ then $\Phi_{i',e}(t_e) \leftarrow \Phi_{i',e}(t_e) - \{t\}$;
5 $\Phi_{i,e}(t_e) \leftarrow \Phi_{i,e}(t_e) \cup \{t\}$;
6 $j \leftarrow 2^{\lceil \log \mathsf{cnt}[\mathcal{T}, e, t_e] \rceil}$;
7 $\mathsf{cnt}[\mathcal{T}, e, t_e] \leftarrow \mathsf{cnt}[\mathcal{T}, e, t_e] + \mathsf{new} - \mathsf{old}$;
8 if $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$ changes and $p_e$ is not the root then
9     foreach $t' \in R_{p_e} \ltimes t_e$ do
10        $\mathsf{old}' \leftarrow j \cdot \prod_{e' \in C_{p_e} - \{e\}} \tilde{\mathsf{cnt}}[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t']$;
11        IndexUpdate($\mathcal{T}, p_e, t', \mathsf{old}'$);
```

Together with the fact that $|(\Join_{e' \in \mathcal{T}_e} R_{e'}) \ltimes t| \leqslant |\Join_{e' \in \mathcal{T}_e} R_{e'}| \leqslant N^{|\mathcal{T}_e|}$, we obtain $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant (2N)^{|\mathcal{T}_e|}$, which implies that $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t]$ can only be doubled by at most $O(\log N)$ times.

Consider any non-root node e. For each tuple $t \in \pi_{\mathsf{key}(e)} R_e$, we organize the tuples $t' \in R_e \ltimes t$ into at most $|\mathcal{T}_e| \cdot \log 2N$ buckets according to the approximate degree of t' in $\mathcal{T}_e$, where the i-th bucket is

$$\Phi_{i,e}(t) = \left\{ t' \in R_e \ltimes t : \prod_{e' \in C_e} \tilde{\mathsf{cnt}}\left[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t'\right] = 2^i \right\}.$$

Let $\mathcal{L}_{e,t}$ be the list of non-empty buckets. For simplicity, we denote $\varphi_{i,e}(t) = 2^i \cdot |\Phi_{i,e}(t)|$ for each $i \in [|\mathcal{T}_e| \cdot \log 2N]$. We also maintain $N_t = \sum_{i \in [|\mathcal{T}_e| \cdot \log 2N]} \varphi_{i,e}(t)$ for each $t \in \pi_{\mathsf{key}(e)} R_e$.

Space Usage. We consider an arbitrary $e \in \mathcal{E}$ in an arbitrary join tree maintained. Let C_e be the children of R_e. We build an index on R_e with $\mathsf{key}(e_i)$ as the key for each $e_i \in C_e$ in order to perform the look up in line 9 of Algorithm 5. There are $|C_e|$ such indices of size $O(N)$ in total. Moreover, for any non-root node e and each tuple $t \in \pi_{\mathsf{key}(e)} R_e$, we organize tuples $R_e \ltimes t$ into buckets and maintain N_t. All these buckets are disjoint and the total size is $O(N)$. As there are $O(1)$ join trees and each join tree contains $O(1)$ nodes, the whole index uses $O(N)$ space.

Index Update. We define a generalized procedure for updating our index. As described in Algorithm 5, the procedure IndexUpdate takes as input the join tree $\mathcal{T}$, a node e, tuple $t \in R_e$ and an integer $\mathsf{old} \geqslant 0$ (indicating the approximated degree of t in $\mathcal{T}_e$ before update). The update proceeds in a recursive way. We first compute the approximated degree of t in $\mathcal{T}_e$ after update, denoted as new. For simplicity, denote $t_e = \pi_{\mathsf{key}(e)}$. We then remove t from the old bucket if it exists (line 4) and insert t into the new bucket (line 5). We increase $\mathsf{cnt}[\mathcal{T}, e, t_e]$ by $\mathsf{new} - \mathsf{old}$ (line 7), and update $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$ if needed. If $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$ has changed, we might need to propagate the updates upward (line 8-11). Furthermore, if p_e is not the root, for each tuple $t' \in R_{p_e} \ltimes t_e$, we compute the approximated degree of t' in $\mathcal{T}_{p_e}$ before update (line 10), denoted as old' and invoke this whole procedure recursively (line 11).

Algorithm 6: BatchGenerate$(\mathcal{T}, e, t)$

```
Input  : A join tree $\mathcal{T}$ for $Q$, a node $e$ and a tuple $t \in R_e$ or $t \in \pi_{\mathsf{key}(e)} R_e$;
Output : A $O(1)$-dense batch $\Delta J \supseteq \Delta Q(\mathcal{R}, t)$;
1 $x \leftarrow \mathsf{supp}(t)$;
2 if $e$ is a leaf node and $x = e$ then return $t$;
3 if $e$ is an internal node and $x = e$ then
4     foreach $e_i \in C_e$ do
5         $B_i \leftarrow$ BatchGenerate$(\mathcal{T}, e_i, \pi_{\mathsf{key}(e_i)} t)$;
6     return $\{t\} \times (\times_{e_i \in C_e} B_i)$;
7 for $t' \in R_e \ltimes t$ do $B_{t'} \leftarrow$ BatchGenerate$(\mathcal{T}, e, t')$;
8 $L \leftarrow \tilde{\mathsf{cnt}}[\mathcal{T}, e, t] - \mathsf{cnt}[\mathcal{T}, e, t]$ dummy elements;
9 return concatenation of $B_{t'}$ for $t' \in R_e \ltimes t$, followed by $L$;
```

When a tuple t is inserted into R_e, we just invoke IndexUpdate $(\mathcal{T}, e, t, 0)$ for every join tree $\mathcal{T}$ used in our index. This may take $O(N)$ time, but the observation is that this update is only triggered when $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$ changes, which happens at most $O(\log N)$ times. Thus, the total update cost is:

$$\sum_{e \in \mathcal{T}} \sum_{t \in \pi_{\mathsf{key}(e)} R_e} \log N \cdot |R_{p_e} \ltimes t| \leqslant \log N \cdot \sum_{e \in \mathcal{T}} \sum_{t \in \pi_{\mathsf{key}(e)} R_e} |R_{p_e} \ltimes t|$$
$$\leqslant \log N \cdot \sum_{e' \in \mathcal{T} : e' \text{ is an internal node}} |R_{e'}| \cdot |C_{e'}| = O(N \log N).$$

Whenever some $\Phi_{i,e}(t)$ changes, we update N_t accordingly. The time for this update is the same as that for updating $\Phi_{i,e}(t)$. Finally, summing over all join trees used, each having a distinct relation as its root, the overall update cost is $O(N \log N)$, namely, the amortized update cost is $O(\log N)$.

Batch Generate. We define a generalized procedure for generating an $\Omega(1)$-dense batch $\Delta J \supseteq \Delta Q(\mathcal{R}, t)$ for any tuple $t \in R_e$ or $t \in \pi_{\mathsf{key}(e)} R_e$, as described in Algorithm 6. The density will depend on the query size, which is taken as a constant, but not on the data size. If tuple t is inserted into R_e, the fist call is BatchGenerate$(\mathcal{T}, e, t)$, where $\mathcal{T}$ is the join tree rooted at node e. In each recursive call, Algorithm 6 distinguishes three cases:

- **Case 1:** e is a leaf node and $t \in R_e$. We simply return t as ΔJ. This batch is 1-dense and $|\Delta J| = 1$.
- **Case 2:** e is an internal node and $t \in R_e$. In this case, $\Delta Q(\mathcal{R}, t)$ can be decomposed into the Cartesian product of $\Delta Q(\mathcal{R}, \pi_{\mathsf{key}(e_i)} t)$ for each child $e_i \in C_e$. The batch ΔJ also follows the same way. We recursively generate a batch for $\pi_{\mathsf{key}(e_i)} t$ in $\mathcal{T}_{e_i}$ for each $e_i \in C_e$, and return their cross product as ΔJ.
- **Case 3:** $t \in \pi_{\mathsf{key}(e)} R_e$. We recursively generate a batch for every tuple $t' \in R_e \ltimes t$ in $\mathcal{T}_e$ and concatenate these batches with $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] - \mathsf{cnt}[\mathcal{T}, e, t]$ dummy elements at the end as ΔJ.

It can be easily shown by induction that

$$|\Delta J| = \begin{cases} \prod_{e_i \in C_e} \tilde{\mathsf{cnt}}[\mathcal{T}, e_i, \pi_{\mathsf{key}(e_i)} t] & \text{if } t \in R_e \\ \tilde{\mathsf{cnt}}[\mathcal{T}, e, t] & \text{if } t \in \pi_{\mathsf{key}(e)} R_e \end{cases}$$

where the second case follows the definition of $\mathsf{cnt}(\cdot)$. Hence, $|\Delta J|$ can be returned in $O(1)$ time. We next prove by induction that

$$\Delta J \text{ is } \phi\text{-dense, where } \phi = \begin{cases} (\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 1} & \text{if } t \in \pi_{\mathsf{key}(e)} R_e \\ (\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 2} & \text{if } t \in R_e \end{cases}$$

```
Algorithm 7: RETRIEVE($\mathcal{T}, e, t, z$)
  Input  : A join tree $\mathcal{T}$ for $Q$, a node $e$ and a tuple $t \in R_e$ or $t \in \pi_{\mathsf{key}(e)} R_e$, an integer $z \geqslant 0$;
  Output : The element at position $z$ in the batch generated for $t$ by BATCHGENERATE($\mathcal{T}, e, t$);
1  $x \leftarrow \mathsf{supp}(t)$;
2  if $e$ is a leaf node then
3      if $z \geqslant \mathsf{cnt}[\mathcal{T}, e, t]$ then return $\perp$;
4      else return the element at position $z$ in $R_e \ltimes t$;
5  if $e = x$ then
6      $C_e \leftarrow \{e_1, e_2, \cdots, e_m\}$;
7      foreach $i \in [1 \dots m]$ do $t_i \leftarrow \pi_{\mathsf{key}(e_i)} t$;
8      Find $(z_1, z_2, \cdots, z_m) \in \times_{i=1}^{m} \tilde{\mathsf{cnt}}[\mathcal{T}, e_i, t_i]$ such that $z = \sum_{i \in [1 \dots m]} \left( z_i \cdot \prod_{j > i} \tilde{\mathsf{cnt}}[\mathcal{T}, e_j, t_j] \right)$;
9      foreach $i \in [1 \dots m]$ do
10         $t'_i \leftarrow$ RETRIEVE($\mathcal{T}, e_i, t_i, z_i$);
11         if $t'_i = \perp$ then return $\perp$;
12     return $(t'_1, t'_2, \cdots, t'_m)$;
13 else
14     if $z \geqslant \mathsf{cnt}[\mathcal{T}, e, t]$ then return $\perp$;
15     Find $i$ such that $\sum_{i' \leqslant i-1} \varphi_{i',e}(t) < z + 1 \leqslant \sum_{i' \leqslant i} \varphi_{i',e}(t)$;
16     $j \leftarrow \left\lfloor \left( z - \sum_{i' \leqslant i-1} \varphi_{i',e}(t) \right) / 2^i \right\rfloor$;
17     $\ell \leftarrow z - \sum_{i' \leqslant i-1} \varphi_{i',e}(t) - 2^i \cdot j$;
18     $t' \leftarrow$ the element at position $j$ in $\Phi_{i,e}(t)$;
19     return RETRIEVE($\mathcal{T}, e, t', \ell$);
```

This holds trivially for **Case 1**. For **Case 2**, we assume that the batch generated for tuple $\pi_{\mathsf{key}(e_i)} t$ is $(\frac{1}{2})^{2 \cdot |\mathcal{T}_{e_i}| - 1}$-dense. Implied by Lemma 3.6, this ΔJ is $(\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 2}$-dense, since

$$(\frac{1}{2})^{|C_e| - 1} \cdot \prod_{e_i \in C_e} (\frac{1}{2})^{2 \cdot |\mathcal{T}_{e_i}| - 1} = (\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 3} \geqslant (\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 2}$$

For **Case 3**, we assume the batch generated for each tuple $t' \in R_e \ltimes t$ is $(\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 2}$-dense. Their concatenation is also $(\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 2}$-dense, implied by Lemma 3.5. The ΔJ is $(\frac{1}{2})^{2 \cdot |\mathcal{T}_e| - 1}$-dense, since $\mathsf{cnt}[\mathcal{T}, e, t] \geqslant \frac{1}{2} \cdot \tilde{\mathsf{cnt}}[\mathcal{T}, e, t]$ and then we invoke Lemma 3.7.

Retrieve. Finally, we describe how to retrieve the join result at position z in the batch generated by BATCHGENERATE. As described in Algorithm 7, RETRIEVE follows the same recursive structure as that of BATCHGENERATE. If tuple t is inserted into R_e, the first call is RETRIEVE($\mathcal{T}, e, t, z$), where $\mathcal{T}$ is join tree rooted at node e. In each recursive call, we also distinguish three cases:

- **Case 1:** e is a leaf node. We simply return the element at position z in the batch accordingly. This takes $O(1)$ time.

- **Case 2:** e is an internal node and $t \in R_e$. In this case, we decompose the index z into a m-coordinate $(z_1, z_2, \cdots, z_m)$ as defined in line 8, then retrieve the element at position z_i in the batch generated for tuple $\pi_{\mathsf{key}(e_i)} t$ for each $e_i \in C_e$ recursively (line 9), and return their combinations as the final result (line 12). The value of $z_1, z_2, \cdots, z_m$ can be computed in $O(1)$ time.
- **Case 3:** e is an internal node and $t \in \pi_{\mathsf{key}(e)} R_e$. In this case, we first locate the bucket into which the element at position z falls, say i. We then locate the index of the tuple whose batch contains the element at position z, say j, and find the specific tuple t'. We also need to compute the index of the target element in the batch generated for t', say ℓ. Finally, the element at position z in the batch generated for t can be found by Retrieve$(\mathcal{T}, e, t', \ell)$. The value of i, j, ℓ can be computed in $O(\log N)$ time.

It is not hard to see that the retrieve operation takes $O(\log N)$ time, by summing the time cost for each recursive invocation.

4.4 Optimizations

We next discuss some optimization techniques for our algorithm. Although they do not improve the complexity results, they significantly reduce the constant factor, as verified in Section 6. More details are given in the full version [2].

Grouping. In a join tree $\mathcal{T}$, for an non-root internal node R_e with its children nodes $\{e_1, e_2, \cdots, e_m\}$, let $\bar{e} = \mathsf{key}(e) \cup \mathsf{key}(e_1) \cup \cdots \cup \mathsf{key}(e_m)$ denote the join attributes. We can *group* tuples in R_e by attributes $\bar{e}$, if $e - \bar{e} \neq \emptyset$. Let $R_{\bar{e}} = \pi_{\bar{e}} R_e$. We replace e with $\bar{e}$ in $\mathcal{T}$. For each tuple $t \in R_{\bar{e}}$, we maintain $\mathsf{feq}[\mathcal{T}, \bar{e}, t] = |R_e \ltimes t|$ and $\tilde{\mathsf{feq}}[\mathcal{T}, \bar{e}, t] = 2^{\lceil \log_2 \mathsf{feq}[\mathcal{T}, \bar{e}, t] \rceil}$. Then, $\mathsf{cnt}[\mathcal{T}, \bar{e}, t]$ is defined as:

$$\mathsf{cnt}[\mathcal{T}, \bar{e}, t] = \sum_{t' \in R_{\bar{e}} \ltimes t} \tilde{\mathsf{feq}}[\mathcal{T}, \bar{e}, t'] \cdot \prod_{e_i \in C_{\bar{e}}} \tilde{\mathsf{cnt}}\left[\mathcal{T}, e_i, \pi_{\mathsf{key}(e_i)} t'\right]$$

Grouping can bring much benefit in index update. More specifically, in line 9-11 of Algorithm 5, instead of propagating update for every tuple $t' \in R_{p_e} \ltimes t_e$, we now propagate update only for every tuple $t \in R_{\bar{p_e}} \ltimes t_e$, where $R_{\bar{p_e}}$ is the projection of relation R_{p_e} onto join attributes in p_e. Hence, we can see a significant reduction in the number of propagated updates.

Foreign-keys. When foreign-key join exists, similar to [31], we simply combine the corresponding sub-join as a whole relation. More specifically, for $R_i \bowtie_X R_j$, where X is the primary key of R_j, we combine R_i, R_j together as a new relation $R_{ij} = R_i \bowtie R_j$. This combination can be recursively done until no more foreign-key join exists. When a tuple t_i is inserted into R_i, we check if there exists a matching tuple $t_j \in R_j$ with the value $\pi_X t_i$. If t_j exists, we insert $t_{ij} = t_i \bowtie t_j$ into R_{ij}. However, when a tuple t_j is inserted into R_j, we need to identify all tuples in R_i that can joined with t_j, and insert $t_{ij} = t_i \bowtie t_j$ into R_{ij}.

5 EXTENSION TO CYCLIC JOINS

We next show how to handle cyclic joins using our algorithm in Section 4, by resorting to the classic GHD decomposition framework [17]. It has been shown [7] that for a join query $Q = (\mathcal{V}, \mathcal{E})$ and any instance of input size N, the maximum join size is $\Theta(N^{\rho^*(Q)})$, where $\rho^*(Q)$ is the fractional edge covering number of Q. Please see an example in Figure 3.

Definition 5.1 (Fractional Edge Covering Number). Given a join query $Q = (\mathcal{V}, \mathcal{E})$, a fractional edge covering is a function $W : \mathcal{E} \to [0, 1]$ such that $\sum_{e \in \mathcal{E}: x \in e} W(e) \geqslant 1$ for every attribute $x \in \mathcal{V}$. The fractional edge covering number $\rho(Q)$ is defined as minimum value of $\sum_{e \in \mathcal{E}} W(e)$ over all possible fractional edge coverings W.

Definition 5.2 (Generalized Hypertree Decomposition). Given a join query $Q = (\mathcal{V}, \mathcal{E})$, a GHD of Q is a pair $(\mathcal{T}, \lambda)$, where $\mathcal{T}$ is a tree as an ordered set of nodes and $\lambda : \mathcal{T} \to 2^{\mathcal{V}}$ is a labeling

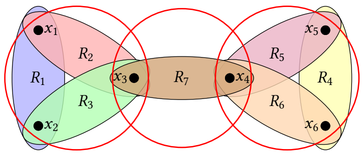


Fig. 3. The dumbbell join $Q = R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3) \bowtie R_4(x_5, x_6) \bowtie R_5(x_4, x_5) \bowtie R_6(x_4, x_6) \bowtie R_7(x_3, x_4)$ with GHD illustrated as the red circle. It has fractional hypertree width $\mathrm{w}(Q) = 1.5$ since the triangle join $R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3)$ and $R_5(x_4, x_5) \bowtie R_6(x_4, x_6) \bowtie R_7(x_3, x_4)$ have the fractional edge covering number $\rho^* = 1.5$.

function, which associates to each vertex $u \in \mathcal{T}$ a subset of attributes in $\mathcal{V}$, such that the following conditions are satisfied:

- For each $e \in \mathcal{E}$, there is a node $u \in \mathcal{T}$ such that $e \subseteq \lambda(u)$;
- For each $x \in \mathcal{V}$, the set of nodes $\{u \in \mathcal{T} : x \in \lambda(u)\}$ forms a connected subtree of $\mathcal{T}$.

Given a join query $Q = (\mathcal{V}, \mathcal{E})$, a GHD $(\mathcal{T}, \lambda)$ and a node $u \in \mathcal{T}$, the width of $\mathcal{T}$ is defined as the optimal fractional edge covering number of its derived subquery $Q_u = (\lambda_u, \mathcal{E}_u)$, where $\mathcal{E}_u = \{e \cap \lambda_u : e \in \mathcal{E}\}$. Given a join query and a GHD $(\mathcal{T}, \lambda)$, the width of $(\mathcal{T}, \lambda)$ is defined as the maximum width over all nodes in $\mathcal{T}$. Then, the fractional hypertree width of a join query follows:

Definition 5.3 (Fractional Hypertree Width [17]). . The fractional hypertree width of a join query Q, denoted as $\mathrm{w}(Q)$, is $\mathrm{w}(Q) = \min_{(\mathcal{T},\lambda)} \max_{u \in \mathcal{T}} \rho^*(Q_u)$, i.e., the minimum width over all GHDs.

Our algorithm builds upon a GHD $(\mathcal{T}, \lambda)$ for the input join query Q, and considers every version of $\mathcal{T}$ by rooting $\mathcal{T}$ at one distinct node. Given an instance $\mathcal{R}$, we define the sub-instance for each node $u \in \mathcal{T}$ as $\mathcal{R}_u = \{\pi_{e \cap \lambda_u} R_e : e \in \mathcal{E}, e \cap \lambda_u \neq \emptyset\}$. Suppose a tuple t is inserted into relation R_e. For each node $u \in \mathcal{T}$ with $e \cap \lambda_u \neq \emptyset$, we add tuple $\pi_{e \cap \lambda_u} t$ to relation $\pi_{e \cap \lambda_u} R_e$ in instance $\mathcal{R}_u$ and update the results of Q_u. We pick an arbitrary node $u \in \mathcal{T}$ with $e \subseteq \lambda_u$. Let $\Delta_u = Q_u(\mathcal{R}_u) \ltimes t$ be the delta join results of t. For every tuple $t' \in \Delta_u$, we just execute line 5-7 in Algorithm 4.

The correctness follows the fact that $Q(\mathcal{R}) \ltimes t = \biguplus_{t' \in \Delta_u} Q(\mathcal{R}) \ltimes t'$, where $\biguplus$ is the disjoint union operator.

Time Complexity. We next analyze the time complexity. For each tuple t inserted into R_e, the join result of Q_u for each node u can be updated in $\mathrm{AGM}(Q_u, \mathcal{R}_u \ltimes t)$ time. Summing over all inserted tuples and nodes in $\mathcal{T}$, the time complexity is

$$\sum_{u \in \mathcal{T}} \sum_{e \in \mathcal{E}} \sum_{t \in R_e} \mathrm{AGM}(Q_u, \mathcal{R}_u \ltimes t) \leqslant \mathrm{AGM}(Q_u, \mathcal{R}_u) = O\left(N^{\mathrm{w}(Q)}\right).$$

Moreover, we can also bound the size of $\Delta_u(t)$ by $\mathrm{AGM}(Q_u, \mathcal{R}_u \ltimes t)$. Hence, the size of the simulated input stream over the GHD $(\mathcal{T}, \lambda)$ is bounded by $\sum_{e \in \mathcal{E}} \sum_{t \in R_e} \mathrm{AGM}(Q_u, \mathcal{R}_u \ltimes t) = O\left(N^{\mathrm{w}(Q)}\right)$.

Space Usage. Our index for cyclic joins builds upon a GHD $(\mathcal{T}, \lambda)$ for Q. Note that the total number of input tuples inserted into each node of $\mathcal{T}$ is $O(N^{\mathrm{w}})$, where w is the fractional hypertree width of Q. Following the same analysis of acyclic joins in Section 4, the space used by our index is proportional to the total number of tuples in each node of $\mathcal{T}$, i.e., $O(N^{\mathrm{w}})$ for cyclic joins.

Putting everything together, we obtain:

THEOREM 5.4. *Given any join 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 $Q(\mathcal{R}^i)$, uses $O(N^w)$*

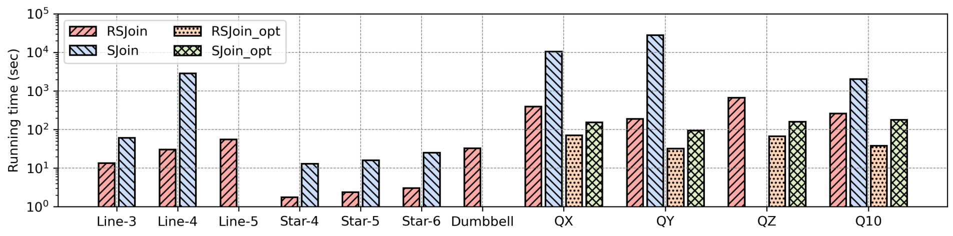


Fig. 4. Running time over different join queries

space and runs in $O(N^w \cdot \log N + k \cdot \log N \cdot \log \frac{N}{k})$ *expected time, where* w *is the fractional hypertree width of* Q.

6 EXPERIMENTS

6.1 Setup

Implementation. We compare our algorithm (denoted as RSJoin) as well as the optimized version when foreign-key join exists (denoted as RSJoin_opt), with the algorithm in [31] (denoted as SJoin) and its optimized version when foreign-key join exists (denoted as SJoin_opt), which is also the state-of-the-art method for supporting random sampling over joins under updates. We mentioned that the symmetric hash join algorithm [3] was proposed for computing the (delta) join results for the basic two-table join over data streams. In [31], symmetric join was combined with reservoir sampling for supporting maintaining uniform samples over joins and also tested as a baseline solution, but its performance is overall dominated by [31], hence we do not include it in our experiments. We implement our algorithms in C++, and conduct experiments on a machine equipped with two Intel Xeon 2.1GHz processors with 24 cores and 251 GB of memory, running CentOS 7. We repeat each experiment 10 times (with timeout as 12 hours) and report the average running time. All code is available at [1].

Datasets and Queries. We evaluate algorithms on graph and relational datasets/queries. All queries in SQL can be found in the version [2].

We use the Epinions dataset that contains 508,837 edges from SNAP (Stanford Network Analysis Project) [23] 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 finds all pairs of triangle that is connected by an edge). There is no foreign-key join in graph queries.

We use two relational datasets. One is TPC-DS dataset [4], which models several generally applicable aspects of a decision support system. We evaluate the same QX, QY, and QZ queries as [31] on TPC-DS, which include the foreign-key joins, and follow the same setup as [31], such that small dimension tables (such as date_dim and household_demographics are pre-loaded, while the rest of the tables are loaded in a streaming fashion. The other is LDBC Social Network Benchmark (LDBC-SNB) [26], which focuses on join-heavy complex queries with updates. We tested Q10 query from the Business Intelligence (BI) workload 10. Similar as before, the static tables (such as tag and city) are pre-loaded, and the dynamic tables are loaded in a streaming fashion.

6.2 Experiment Results

Running time. Figure 4 shows the running time 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

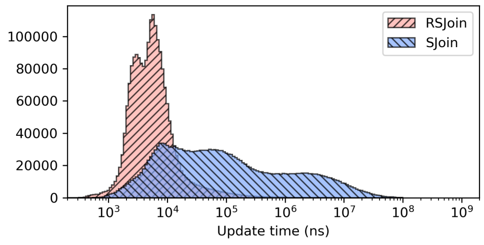


Fig. 5. Update time distribution

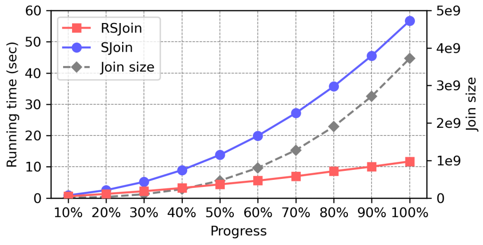


Fig. 6. Running time v.s. input size and join size

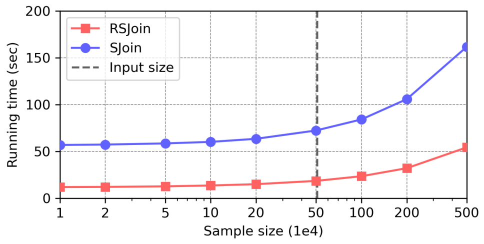


Fig. 7. Running time v.s. sample size

Optimizations	#Execution	Run-time (sec)
N/A	172010370	678.864
Foreign-key	132175648	204.614
Foreign-key + Grouping	597557	68.047

Fig. 8. Optimizations on QZ over TPC-DS dataset

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 5 shows the result on line-4 join. Most of the update time required is roughly 10 μs, with an average of 13 μs. Some tuple may incur 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 update time. 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 6 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.

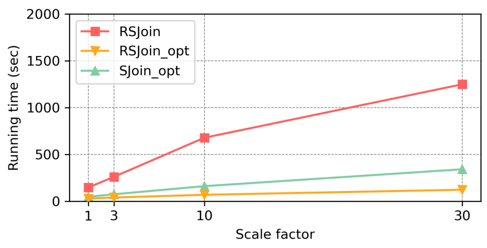


Fig. 9. Running time v.s. scale factor

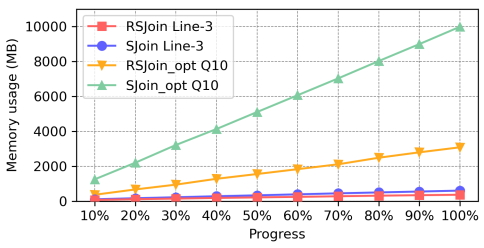


Fig. 10. Memory usage v.s. input size

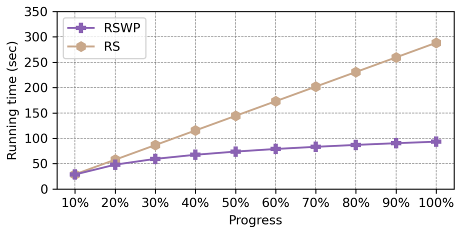


Fig. 11. Running time v.s. input size

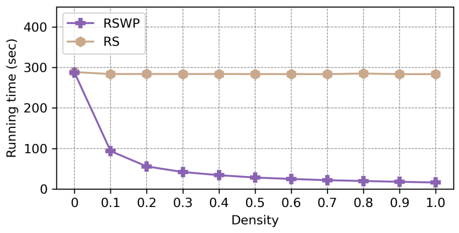


Fig. 12. Running time v.s. density

Sample size. We next study how the sample size k affect the total execution time of both algorithms. Figure 7 shows the running time 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 slow. 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 increases 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 rapid increase in the total execution time. SJoin follows a similar trend. Moreover, when the sample size reaches $k = 10,000$, the running time required by SJoin is even more than that required by RSJoin for the case when sample size is as large as $k = 5,000,000$.

Scalability. To examine the scalability of both methods, we evaluate the QZ query on TPC-DS dataset with scale factors of 1,3,10, and 30. The results are shown in Figure 9. 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 scale factor as 1. We observe that even without applying foreign-key optimization, RSJoin achieves linear growth in the running time as the scale factor increases, which indicates that RSJoin is scalable and practical even when dealing with significantly huge input size.

Memory usage. In addition, we explore the memory usage of all methods. Figure 10 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 10. 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.

Optimizations. We evaluate the effectiveness of our optimizations by counting the number of loop execution lines 9-11 in Algorithm 5. Table 8 records the count as well as the total running time of our method for QZ query over the TPC-DS dataset (scale factor 10 and sample size 1,000,000). It is clear to see that when applying foreign-key optimizations, the number of propagation decreases as well as the total execution time. If applying the grouping optimizations, we can further decrease the total execution time, achieving roughly 10x speedup over the RSJoin without optimization.

6.3 Reservoir Sampling with Predicate

At last, we compare our new reservoir sampling algorithm with predicate (denoted as RSWP) with the classic reservoir sampling algorithm (denoted as RS) on data streams. We generate a data stream as follows. We fix a random string of 1024 characters, referred 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 data stream whose edit distance from the query string is less than or equal to 16.

In Figure 11, 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 running time 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 running time of RSWP grows slower and slower, which is consistent with our theoretical result that it takes $O\left(\frac{k}{r_i+1}\right)$ expected time to process the i-th item.

In figure 12, we measure the running time 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 running time only depends on the input size, instead of the density of input stream. In contrast, the running time 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 running time 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.

7 RELATED WORK

In addition to the directly related work mentioned in Section 2.2, the following is also relevant to our work:

Streaming Subgraphs Sampling. The problem of sampling subgraph patterns from a graph whose edges come as an input stream has also been considered (where the space usage is important). For example, Paven et. al [25] designed an algorithm that uses $O(\frac{N^{3/2}}{\mathrm{OUT}})$ space, where OUT is the number of triangles in the graph. In the field of property testing (where sub-linear number of query accesses to the graph is important), Eden et al [16] studied the problem of almost uniform sampling of edges, and Biswas et. al [11] studied the problem of sampling subgraphs.

Maintaining Conjunctive Queries under Updates. It has been shown [10, 18] that a very restrictive class of queries, known as q-hierarchical query can admit an index with $O(1)$ update time.

However, any non-q-hierarchical query, a lower bound of $\Omega(N^{\frac{1}{2}-\epsilon})$ has also been proved on the update time, for any small constant $\epsilon > 0$. This result is rather negative, since q-hierarchical queries are a very restricted class; for example, the line-3 join. Meanwhile, [18] showed an index for acyclic joins that can be updated in $O(N)$ time. Later, Kara et al. [19] designed optimal data structures that can be updated in $O(\sqrt{N})$ time while supporting $O(1)$-delay enumeration for line-3 join, triangle join, length-4 cycle join, etc. Moreover, Kara et al. [20] also investigated the tradeoff between update time and delay for hierarchical queries. Wang et al. [28, 29] worked on instance-dependent complexity by relating the update time to the enclosureness of update sequences.

8 CONCLUSION

In this paper, we propose a general reservoir sampling algorithm that supports a predicate. We design a dynamic data structure that supports efficient updates and direct access of 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 as open, such as uniform sampling over join-project queries over data streams.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16205420, 16205422, and 16204223.

REFERENCES

[1] Code. https://github.com/hkustDB/Reservoir-Sampling-over-Joins

[2] Reservoir Sampling over Joins. https://arxiv.org/pdf/2404.03194.pdf

[3] Symmetric hash join. https://en.wikipedia.org/wiki/Symmetric_hash_join

[4] TPC-DS dataset. https://www.tpc.org/tpcds/

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Vol. 8. Addison-Wesley Reading.

[6] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data.* 275–286.

[7] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science.* IEEE, 739–748.

[8] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic.* Springer, 208–222.

[9] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. 1983. On the desirability of acyclic database schemes. *JACM* 30, 3 (1983), 479–513.

[10] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems.* 303–318.

[11] AS Biswas, T Eden, and R Rubinfeld. 2021. Towards a Decomposition-Optimal Algorithm for Counting and Sampling Arbitrary Motifs in Sublinear Time. *RANDOM 2021* (2021).

[12] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 393–409.

[13] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On random sampling over joins. *ACM SIGMOD Record* 28, 2 (1999), 263–274.

[14] Yu Chen and Ke Yi. 2020. Random Sampling and Size Estimation Over Cyclic Joins. In *23rd International Conference on Database Theory.*

[15] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. On Join Sampling and Hardness of Combinatorial Output-Sensitive Join Algorithms. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2023).*

[16] Talya Eden and Will Rosenbaum. 2018. On Sampling Edges Almost Uniformly. In *1st Symposium on Simplicity in Algorithms (SOSA 2018).*

[17] Georg Gottlob, Gianluigi Greco, Francesco Scarcello, et al. 2014. Treewidth and hypertree width. *Tractability: Practical Approaches to Hard Problems* 1 (2014), 20.

[18] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 1259–1274.

[19] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Counting Triangles under Updates in Worst-Case Optimal Time. In *22nd International Conference on Database Theory.* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[20] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 375–392.

[21] Kyoungmin Kim, Jaehyun Ha, George Fletcher, and Wook-Shin Han. 2023. Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 113–125.

[22] Donald Ervin Knuth. 1997. *The art of computer programming.* Vol. 3. Pearson Education.

[23] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[24] Kim-Hung Li. 1994. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *ACM Transactions on Mathematical Software (TOMS)* 20, 4 (1994), 481–493.

[25] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1870–1881.

[26] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (dec 2022), 877–890. https://doi.org/10.14778/3574245.3574270

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

[28] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1046–1058.

[29] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1225–1239.

[30] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.

[31] Zhuoyue Zhao, Feifei Li, and Yuxi Liu. 2020. Efficient join synopsis maintenance for data warehouse. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2027–2042.

Received October 2023; accepted January 2024