RESEARCH-ARTICLE

Reservoir Sampling over Joins

BINYANG DAI, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

XIAO HU, University of Waterloo, Waterloo, ON, Canada

KE YI, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

Published: 10 March 2026
Online AM: 07 January 2026
Accepted: 16 December 2025
Revised: 03 September 2025
Received: 23 April 2025

Reservoir Sampling over Joins

BINYANG DAI, The Hong Kong University of Science and Technology School of Engineering, Hong Kong, Hong Kong
XIAO HU, Computer Science, University of Waterloo, Waterloo, Canada
KE YI, The Hong Kong University of Science and Technology School of Engineering, Hong Kong, Hong Kong

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 article, 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. 2026. Reservoir Sampling over Joins. *ACM Trans. Datab. Syst.* 51, 2, Article 9 (March 2026), 35 pages. https://doi.org/10.1145/3787855

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

Binyang Dai and Xiao Hu contributed equally to this research.
This work has been supported by HKRGC under grants 16205420, 16205422, 16204223, and the Natural Sciences and Engineering Research Council of Canada – Discovery Grant.
Authors' Contact Information: Binyang Dai (corresponding author), The Hong Kong University of Science and Technology School of Engineering, Hong Kong, Hong Kong; e-mail: bdaiab@ust.hk; Xiao Hu, Computer Science, University of Waterloo, Waterloo, Ontario, Canada; e-mail: xiaohu@uwaterloo.ca; Ke Yi, The Hong Kong University of Science and Technology School of Engineering, Hong Kong, Hong Kong; e-mail: yike@ust.hk.

ACM 0362-5915/2026/03-ART9
https://doi.org/10.1145/3787855

and then randomly access the table, but this loses the performance benefit of sampling. In as early as 1999, two prominent papers [7, 14] asked the intriguing question, of whether a sample can be obtained without computing the full join. As observed in by [14], the main barrier is that the sampling operator cannot be pushed down, i.e., $\text{sample}(R \bowtie S) \neq \text{sample}(R) \bowtie \text{sample}(S)$. 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 2.3) that can be built in $O(N)$ time, where N is the number of tuples in the database, which can then be used to draw a sample of the join results in $O(1)$ time [13, 35]. 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 can we efficiently and continuously maintain a uniform sample of the join results produced by tuples seen so far? One naive solution would be to rebuild the index and re-draw the samples after each tuple has arrived, but this results in a total running time of $O(N^2)$ to process a stream with N tuples. Recently, Zhao et al. [36] applied the reservoir sampling algorithms [27, 31] 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 article presents a new reservoir sampling algorithm for maintaining a sample over joins with a near-linear running time of $O(N \log N + k \log N \log \frac{N^{\rho^*}}{k})$,[1] where k is the given sample size and ρ^* is the fractional edge cover number of the join (formal definition given in Section 2.3). 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 [26], maintains a sample of size k in $O(N)$ time over a stream of N items, which is already optimal. Assuming there is a $\text{skip}(i)$ 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 [27, 31]. In this article, 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 $O(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i+1}\right))$, where r_i the number of items in the first $i-1$ items that pass the predicate. Note that when $\theta(\cdot) \equiv \texttt{true}$, we have $r_i = i-1$ and the bound simplifies to $O(k \log \frac{N}{k})$, matching the classical result. Meanwhile, the complexity degrades gracefully as the stream becomes sparser, i.e., fewer items pass the predicate. Intuitively, sparse streams are more difficult, as it is not safe to skip items aggressively. In the extreme case where only one item passes the predicate, that item must be included in the sample, and we have to check every item in order not to miss it.

However, the assumption that $\text{skip}(i)$ takes $O(1)$ time is usually not true: One must at least use a counter to track how many items have been skipped, which already requires $O(i)$ time. Interestingly, the reservoir sampling over joins problem provides a notable scenario where this assumption holds true, albeit with a time complexity slightly higher than constant. It is known [8] that the join results of N tuples can be as many as N^{ρ^*}. Thus, the stream of input tuples implicitly defines a polynomially longer, but conceptual, stream of join results, which we want to sample from. As there is a good structure in the latter, there is no need to materialize this join result stream. As such, it is possible to skip its items more efficiently without counting them one by one.

[1]The bound claimed in the conference version is $O(N \log N + k \log N \log \frac{N}{k})$ which is not entirely correct. Note that we define $\log(x) = 1$ for $x \leqslant 2$.

Dynamic sampling from joins. Let Q be an acyclic join query, and $Q(\mathcal{R})$ the query results of Q on database instance $\mathcal{R}$. 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 (i.e., tuples not included in the join result), 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 still holds when $Q(\mathcal{R})$ is replaced by the delta query $\Delta Q(\mathcal{R}, t) := Q(\mathcal{R} \cup \{t\}) - Q(\mathcal{R})$, for any tuple $t \notin \mathcal{R}$.

N is the total number of tuples added to $\mathcal{R}$, but our index does not need the knowledge of N in advance. Introducing dummy tuples seems to make the problem more complicated because we need to filter out these dummy tuples to ensure they do not enter the reservoir and become part of the sample. However, we will show that while dummy tuples require a small amount of additional work in sampling, they also significantly reduce the overhead of updating the index structure, leading to an algorithm with lower overall complexity.

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 a 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 [13, 35], which are inherently static. Furthermore, operations (1) and (2) together already provide a solution for the reservoir sampling over joins 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 $O(N \log N + Nk \log N) = O(Nk \log N)$, but still not near-linear.

To achieve near-linear time, we use operation (3) in conjunction with our reservoir-sampling-with-a-predicate 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), we obtain a bounded time complexity.

The contributions of this article are summarized as follows, which also serve as a roadmap of the article:

— **(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 $O(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i+1}\right))$, which we also show is instance-optimal.
— **(Section 4)** We provide a high-level overview of our approach through an example of maintaining a sample over the line-3 join.
— **(Section 5)** We present a dynamic index for acyclic joins that supports drawing a sample from either the full query or the delta query efficiently. Combined with our reservoir sampling with predicate algorithm, we show how the reservoir sampling over join problem can be solved in $O(N \log N + k \log N \log \frac{N\rho^*}{k})$ time. We further present some optimization techniques when key constraints are present.
— **(Section 6)** We extend the algorithm to non-full free-connex queries with the same asymptotic complexity, thus allowing us to handle all **selection-projection-join** (**SPJ**) queries.

— **(Section 7)** We extend our algorithm to arbitrary cyclic queries using the tree decomposition techniques. The running time becomes $O(N^{\mathrm{w}} \log N + k \log N \log \frac{N^{\rho^*}}{k})$, where w is the free-connex fractional hypertree width of the query.
— **(Section 8)** As applications, we show how our reservoir sampling algorithm can be used to perform mean estimation, selectivity estimation, and query cardinality estimation continuously over a stream of tuples.
— **(Section 9)** We have implemented our algorithm and evaluated it over both graph and relational data. The experimental results show that our algorithm significantly outperforms the state-of-the-art solution [36].

2 Preliminaries

2.1 Problem Definition

We first recap some standard notation in relational algebra [6]. 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 that have the same value(s) on attribute(s) $e \cap \mathrm{supp}(t)$ 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 $\mathcal{R}^0 = \varnothing$. 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 Refs. [7, 13–15, 18, 25, 35]. 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 [27, 31, 36], where we wish to maintain a sample of size k 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 [6] 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, so we may assume that the input stream has no duplicates.

2.2 Previous Results

Sampling over joins. Chaudhuri et al. [14] 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. [7] 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 [13, 35]. 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 [15], which has recently been improved to $O\left(\frac{N^{\rho^*}}{|Q(\mathcal{R})|}\right)$ [18, 25], 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 [26] solves the problem in $O(N)$ time. Assuming that skip takes $O(1)$ time, this can be reduced to $O(k \log \frac{N}{k})$ [27, 31]. Zhao et al. [36] study 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 sampling index [13, 35] at each time step.

Hardness results. In this article, we will focus on acyclic joins over an insertion-only stream. Both restrictions are 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 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 [9]. 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 any 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 [11]. This means that the reservoir sampling problem requires at least $\Omega(N^{1.5})$ time over a fully-dynamic stream.

Prior publication. This article is an extended version of a previous conference article [17]. In addition to improved presentations and filling missing proofs, this version has the following new technical contents: (1) The algorithm in the conference version only supports full queries. In this version, we have extended our algorithm to support non-full queries (Sections 6 and 7). (2) We present two applications of our reservoir sampling algorithm for any SPJ query: mean estimation and cardinality estimation continuously over a stream of tuples (Section 8), together with additional experiments to validate their practical performance (Section 9).

2.3 Background on Join and Join-Project Queries

For a join query $Q = (\mathcal{V}, \mathcal{E})$, and a subset of attributes $S \subseteq \mathcal{V}$, we use $Q[S] = (S, \mathcal{E}[S])$ to denote the subjoin induced by S, where $\mathcal{E}[S] = \{e \cap S : e \in \mathcal{E}\}$. A *fractional edge covering* is a function $\rho : \mathcal{E} \to [0, 1]$ such that $\sum_{e:x\in e} \rho(e) \geqslant 1$ for each $x \in \mathcal{V}$. The *fractional edge covering number* of Q, denoted as $\rho^*(Q)$, is defined as the minimum sum of weight over all possible fractional edge coverings ρ for Q, i.e., $\rho^*(Q) = \min_\rho \sum_{e\in\mathcal{E}} \rho(e)$. The AGM bound [8] states that the maximum number of join results produced by a join query Q over any instance of input size N is $\Theta(N^{\rho^*(Q)})$.

Definition 2.1 (Tree Decomposition (TD)*).* A *tree decomposition* of a join query $Q = (\mathcal{V}, \mathcal{E})$ is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree and $\chi : \text{nodes}(\mathcal{T}) \to 2^{\mathcal{V}}$ is a mapping from the nodes of $\mathcal{T}$ to subsets of $\mathcal{V}$, that satisfies the following:

- **(cover property)** For each relation $e \in \mathcal{E}$, there is a node $u \in \text{nodes}(\mathcal{T})$ such that $e \subseteq \chi(u)$.
- **(connect property)** For each attribute $x \in \mathcal{V}$, the set $\{u \in \text{nodes}(\mathcal{T}) : x \in \chi(u)\}$ forms a connected sub-tree of $\mathcal{T}$.

Each set $\chi(u)$ is called a *bag* of the TD. The *width* of a TD $(\mathcal{T}, \chi)$ is defined as

$$\text{width}(\mathcal{T}, \chi) \quad = \quad \max_{u \in \text{nodes}(\mathcal{T})} \quad \rho^* \left(Q[\chi(u)]\right)$$

i.e., the maximum fractional edge covering number of the sub-queries induced by all bags in $\mathcal{T}$.

Definition 2.2. A join query is acyclic if it has a width-1 TD .

We also extend our scope from join queries to join-project queries (a.k.a. conjunctive queries), for which not all attributes are outputted. A join-project query is denoted as $\pi_{\mathbf{y}} Q$, where $Q = (\mathcal{V}, \mathcal{E})$ is the underlying join query, and $\mathbf{y} \subseteq \mathcal{V}$ is the set of output attributes. Given an instance $\mathcal{R}$ for Q, the query results of $\pi_{\mathbf{y}} Q$ over $\mathcal{R}$ is defined as

$$\pi_{\mathbf{y}} Q(\mathcal{R}) = \left\{ t \in \prod_{x \in \mathbf{y}} \text{dom}(x) \mid \exists t' \in \prod_{x \in \mathcal{V}} \text{dom}(x), \pi_{\mathbf{y}} t' = t, \forall e \in \mathcal{E}, \exists t_e \in R_e, \pi_e t' = t_e \right\}. \quad (2)$$

Note that "project" refers to a distinct projection under this definition.

The *fractional edge covering number* of $\pi_{\mathbf{y}} Q$, is defined as the fractional edge covering number of the join query induced by output attributes, i.e., $\rho^*(Q[\mathbf{y}])$. To simplify notation, we may use ρ^* to denote $\rho^*(Q)$ or $\rho^*(Q[\mathbf{y}])$ when the context is clear. Similarly, the maximum number of query results produced by $\pi_{\mathbf{y}} Q$ over any instance of input size N is $\Theta(N^{\rho^*(Q[\mathbf{y}])})$.

Still, a TD $(\mathcal{T}, \chi)$ of a join-project query $\pi_{\mathbf{y}} Q$ is defined on the underlying join query Q. $(\mathcal{T}, \chi)$ is *free-connex* for $\pi_{\mathbf{y}} Q$ if there is a connected subtree $\mathcal{E}_{\text{con}}$ of $\mathcal{T}$ such that $\bigcup_{u \in \text{nodes}(\mathcal{E}_{\text{con}})} \chi(u) = \mathbf{y}$, i.e., the union of attributes appearing in $\mathcal{E}_{\text{con}}$ is exactly the output attributes.

Definition 2.3. A join-project query $\pi_{\mathbf{y}} Q$ is free-connex if it has a width-1 free-connex TD with respect to $\mathbf{y}$.

For an arbitrary join-project query $\pi_{\mathbf{y}} Q$, the *free-connex fractional hypertree width* is defined as

$$\text{fn-fhtw}(\pi_{\mathbf{y}} Q) = \min_{(\mathcal{T}, \chi) \text{ is a free-connex TD for } \pi_{\mathbf{y}} Q} \text{width}(\mathcal{T}, \chi) \quad (3)$$

i.e., the minimum width over all possible free-connex tree decompositions.

3 Reservoir Sampling with Predicate

3.1 Reservoir Sampling Revisited

Reservoir sampling [27, 31] 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 Ref. [26], 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.

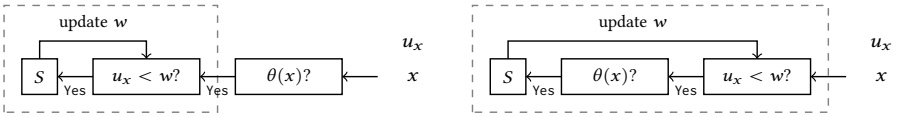


Fig. 1. Comparison between the naive (left) and our algorithm (right) on reservoir sampling with the predicate.

ALGORITHM 1: ReservoirPred(D, k, θ)

Input : An input stream D of items, an integer $k > 0$, and a predicate θ;
Output: A set S maintaining a random sample of size k without replacement of items on which θ evaluates to true;

```
1  S ← ∅, w ← 1;
2  while |S| < k do
3      x ← D.next();
4      if x = null then break;
5      if θ(x) then S ← S + {x};
6  w ← rand()^{1/k};
7  q ← ⌊ln(rand())/ln(1 − w)⌋;
8  while true do
9      x ← D.skip(q);
10     if x = null then break;
11     if θ(x) then
12         y ← a randomly chosen item from S;
13         S ← S − {y} ∪ {x};
14         w ← w · rand()^{1/k};
15     q ← ⌊ln(rand())/ln(1 − w)⌋;          // Note that q ∼ Geo(w)
```

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 [27]. It is based on the fact that, in a set of N independent random numbers drawn from the uniform distribution $\mathsf{Uni}(0,1)$, the indices of the smallest k random numbers are a sample without replacement from the index set $\{1, 2, \cdots, N\}$. The algorithm works as follows. **(Step 1)** It initializes S as before, and set $w = u^{1/k}$ for $u \sim \mathsf{Uni}(0,1)$. **(Step 2)** It draws a random number q from the geometric distribution $\mathsf{Geo}(w)$, and skip the next q items. It then replaces a random item from S with x_i, and updates w to $w \cdot u^{1/k}$ for $u \sim \mathsf{Uni}(0,1)$. It can be shown [27] 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 are dummy). We assume that θ can be evaluated in $O(1)$ time. Note that the $O(N)$-time algorithm easily supports a predicate: We just evaluate θ on each item and feed the real items to the algorithm as illustrated in Figure 1. It is more nontrivial to adapt the $O(k \log \frac{N}{k})$ algorithm since the skip operation skips an unknown number of real items.

We adapt the reservoir sampling algorithm [27] to Algorithm 1. The insight is that the random variable u_x, which is drawn from $\mathsf{Uni}(0,1)$ when item x is visited, is independent of $\theta(x)$. This

ALGORITHM 2: NaiveReservoirPred(D, k, θ)

Input : An input stream D of elements, an integer $k > 0$ and a predicate θ;
Output: A set S maintaining a random sample of size k without replacement of items that pass the predicate θ;

```
1  S ← ∅, w ← 1;
2  while |S| < k do
3      x ← D.next();
4      if x = null then break;
5      if θ(x) then S ← S ∪ {x};
6  w ← rand()^{1/k};
7  while true do
8      x ← D.next();
9      if x = null then break;
10     u_x ← rand();
11     if u_x < w then
12         if θ(x) then
13             y ← a randomly chosen element from S;
14             S ← S − {y} ∪ {x};
15             w ← w · rand()^{1/k};
```

property allows us to swap the order of the two comparisons ($u_x < w$ and whether $\theta(x)$ is true), as illustrated in Figure 1. Note that we do not need to explicitly assign a random number to each item as the number of failures before the next success (i.e., $u_x < w$) follows a geometric distribution parameterized by w. As in Ref. [27], we draw a random variable $q \sim \text{Geo}(w)$ to determine the number of items to skip. After skipping q items, we fetch the next item x and evaluate $\theta(x)$. If it satisfies θ, we insert it into the reservoir S and update w to $w \cdot u^{1/k}$, where $u \sim \text{Uni}(0, 1)$. Otherwise, we keep w unchanged and draw the next random variable $q \sim \text{Geo}(w)$.

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 [27], we have made two changes: (lines 2–5) when the reservoir is not full, we only add real items to it; (lines 11–14) we only update the reservoir and the parameter w when the algorithm stops at a real item.

Theorem 3.1. *Algorithm 1 maintains a sample of size k of all items on which θ evaluates to true.*

Proof. Algorithm 1 runs essentially the same process as Algorithm 2, since the next random value drawn from $\text{Uni}(0, 1)$ being smaller than w follows the geometric distribution parameterized by w. Furthermore, in Algorithm 2, it is safe to exchange the if condition in line 11 with the if condition in line 12, since these two conditions are independent of each other. After this exchange, it simply runs the classic reservoir sampling only over real elements, whose correctness is proved in [27]. Together, the correctness of Algorithm 1 is proved by the correctness of the classic reservoir sampling algorithm. □

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 3.2. *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 next$(\cdot)$ *and* skip$(\cdot)$*, respectively.*

Proof. For 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 an index larger than or equal to p. We analyze the number of invocations of skip$(\cdot)$ by Algorithm 1. Each time Algorithm 1 calls skip$(\cdot)$ and returns some item x, we say it stops at x. Note that Algorithm 1 only stops at items from S'. We consider the equivalent version shown in Algorithm 2 for the ease of analysis. The number of stops is the same as the number of items passing the check in line 11 in Algorithm 2. So we also say Algorithm 2 stops at item x_j if x_j passes the check in line 11. Let w_j be the value of parameter w when it visits item x_j and $i_1, i_2, \cdots, i_{r_{N+1}}$ be the indices of the real items in the stream. Correspondingly, let $\mathbf{w} = \langle w_{i_1}, w_{i_2}, \cdots, w_{i_{r_{N+1}}} \rangle$ be the state. We observe that (1) $w_j \in [0, 1]$ for all j; (2) $w_j \geqslant w_{j'}$ if $j < j'$; (3) $w_j = 1$ if $j < p$. For a dummy item x_j, we use function $\pi(j)$ to denote the smallest index $j' > j$ such that $x_{j'}$ is real. Given $\mathbf{w}$, it stops at a real item x_{i_j} if and only if $w_{i_j} > w_{i_{j+1}}$, i.e., the value of w has changed when it visits the next real item. Let $\mathbb{I}(\cdot)$ be an indicator function. Hence,

$$\mathbb{I}(\text{stop on } x_{i_j} | \mathbf{w}) = \begin{cases} 1 & \text{if } w_{i_j} > w_{i_{j+1}}, \\ 0 & \text{otherwise} \end{cases}$$

Let W be the set of all possible states. The expected number of stops in S' is

$$\begin{aligned}
\mathbb{E}[\#\text{stops in } S'] &= \sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot \mathbb{E}[\#\text{stops in } S' | \mathbf{w}] \\
&= \sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot \sum_{j=p}^{N} \mathbb{E}[\#\text{stop on } x_j | \mathbf{w}] \\
&= \sum_{j \in [p,N] \cap \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot \mathbb{I}(\text{stop on } x_j | \mathbf{w}) \\
&+ \sum_{j \in [p,N] - \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot \Pr(\text{stop on } x_j | \mathbf{w}) \\
&= \sum_{j \in [p,N] \cap \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \frac{k}{r_j + 1} + \sum_{j \in [p,N] - \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot w_{\pi(j)} \\
&= \sum_{j \in [p,N] \cap \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \frac{k}{r_j + 1} + \sum_{j \in [p,N] - \{i_1, i_2, \cdots, i_{r_{N+1}}\}} \frac{k}{r_j + 1} = \sum_{j=p}^{N} \frac{k}{r_j + 1}
\end{aligned}$$

where the second last equality follows the fact that $\sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot \mathbb{I}(\text{stop on } x_j | \mathbf{w})$ is exactly the probability that the real item x_j enters the reservoir ever and the last equality follows the fact that $w_j = w_{\pi(j)}$ for dummy item x_j and $\sum_{\mathbf{w} \in W} \Pr(\mathbf{w}) \cdot w_{\pi(j)}$ is exactly the probability that the real item $x_{\pi(j)}$ enters the reservoir.

Next, we move to the number of invocations of next() by Algorithm 1. For the while loop at 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 finish 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 [27, 31]. 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 3.3. *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 $i \in [1, N]$. 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 no algorithm can 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 $O(N)$, it is closer to the former as long as the stream is dense enough.

Definition 3.4 (Dense stream). A stream $S = \langle x_1, x_2, \cdots, x_n \rangle$ is *ϕ-dense* for $0 < \phi \leqslant 1$, if $r_i \geqslant \phi \cdot (i-1)$ for all i.

Combining Theorem 3.2 and Definition 3.4 we obtain:

Corollary 3.5. *For any ϕ-dense stream where ϕ is a constant, Algorithm 1 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 3.6 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 3.8. The more dummy items padded, the sparser the stream becomes. Lemma 3.7 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 3.6, Lemma 3.7, and Lemma 3.8. It is easy to see that a stream S is ϕ-dense if $q_i \geqslant \phi \cdot i$ for all i.

Lemma 3.6. *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 $i \in [1, m]$. 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 $j \in [1, m]$, we have proved that $q_j \geqslant \min\{\phi_1, \phi_2\} \cdot j$ for every $j \in [1, m+n]$. □

Lemma 3.7. *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.*

ALGORITHM 3: BatchReservoirPred(D, k, θ)

Input : An input stream D of item-disjoint batches, an integer $k > 0$, and a predicate θ;
Output: A set S maintaining a random sample of size k without replacement of items on which θ evaluates to true;

```
S ← ∅, w ← 1, q ← 0;
foreach batch B ∈ D do
    (S, w, q) ← BatchUpdate(S, k, B, q, w, θ);
```

Proof. Consider the stream $S_1 \times S_2 = \langle z_1, z_2, \dots z_{mn} \rangle$ 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 $\langle z_{jn+1}, z_{jn+2}, \dots z_{jn+n} \rangle$ be the row_j, where $j = 0, 1, \dots i_1 - 1$. 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 resulting 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 $\phi_1 \cdot \phi_2 \cdot n \cdot i_1$ items 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 3.8. *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 resulting 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 a random sample of size k 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 3, which calls BatchUpdate (Algorithm 4) for every batch. BatchUpdate essentially runs Algorithm 1 on the 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 4.

ALGORITHM 4: BatchUpdate(S, k, B, q, w, θ)

Input : A set S maintaining a random sample, an integer $k > 0$, a new batch B with the first q items to be skipped, parameter w and a predicate θ;
Output : Updated S, w and q;

```
1  while |S| < k and B.remain() > 0 do
2      x ← B.next();
3      if θ(x) then S ← S + {x};
4  if |S| < k then return S, w, q;
5  if w = 1 then
6      w ← rand()^{1/k};
7      q ← ⌊(ln(rand())/ln(1 − w))⌋;            // Note that q ∼ Geo(w)
8  while B.remain() > q do
9      x ← B.skip(q);
10     if θ(x) then
11         y ← a randomly chosen item from S;
12         S ← S − {y} ∪ {x};
13         w ← w · rand()^{1/k};
14     q ← ⌊(ln(rand()/ln(1 − w))⌋;            // Note that q ∼ Geo(w)
15 return S, w, q − B.remain();
```

Moreover, we note that lines 6–7 of Algorithm 1 will be invoked only once, 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 1 at the beginning (line 1 of Algorithm 3), so that lines 6–7 of Algorithm 4 will be invoked the first time when the reservoir S is filled with k items.

The samples maintained by Algorithm 3 are the same as those maintained by Algorithm 1 over items in batches, so correctness follows immediately. Below, we analyze its running time.

Theorem 3.9. *Given an input stream of batches each of which is ϕ-dense for some constant ϕ, Algorithm 3 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 3 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 number of invocations to next$(\cdot)$ and skip$(\cdot)$ in Algorithm 3 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 3 makes m calls to BatchUpdate in total. Then, it follows Corollary 3.5. □

4 Reservoir Sampling Over the Line-3 Join

In this section and the next, we present our algorithms for maintaining a uniform sample of size k over a join query $Q(\mathcal{R})$, where a stream of tuples are being continuously inserted into the database instance $\mathcal{R}$. We first use the line-3 join $R_1(A,B) \bowtie R_2(B,C) \bowtie R_3(C,D)$ to demonstrate our high-level ideas, before considering general acyclic joins in Section 5.

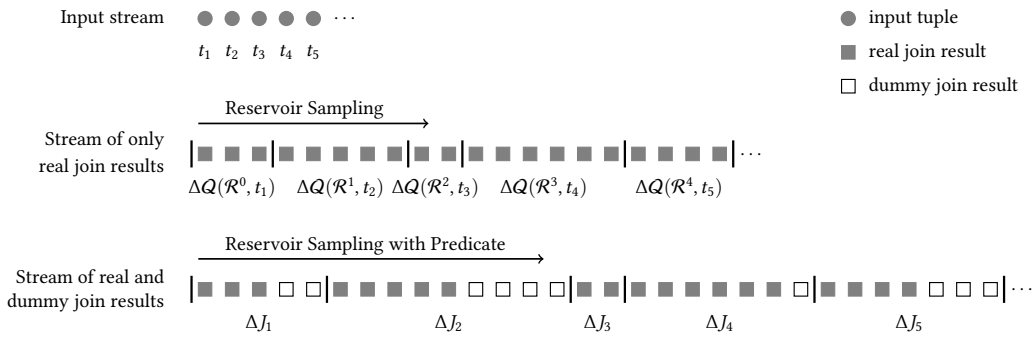


Fig. 2. An illustration of reservoir sampling over join.

4.1 Previous Approaches

The problem can be solved by a straightforward combination of incremental view maintenance and reservoir sampling, as illustrated in the middle row of Figure 2: We compute the delta query $\Delta Q(\mathcal{R}, t)$ for each incoming tuple t, and feed all the new join results to the reservoir sampling algorithm. However, as the line-3 join can generate up to $O(N^2)$ results in the worst case, this approach incurs a quadratic cost overall.

SJoin [36] avoids materializing $\Delta Q(\mathcal{R}, t)$ by maintaining a data structure that can be used to take samples from $\Delta Q(\mathcal{R}, t)$. However, this still incurs $O(N^2)$ cost in the worst case, as illustrated in Figures 3–4. In the figures, we use a common domain $\{1, 2, \ldots, 7\}$ for all the attributes, so that the database instance can be represented as a graph on a grid of 7×4 vertices, where the columns correspond to the four attributes A, B, C, D and the rows correspond to the values in the domain. Each relation, e.g., R_1, can then be represented as a bipartite graph between two columns, e.g., A and B; each join result is thus a length-3 path from some vertex in column A to some vertex in column D.

Figure 3 shows the insertion of tuple $(2, 1)$ into R_1, highlighted in red. The delta query $\Delta Q(\mathcal{R}, t)$ for this tuple includes the 7 new paths highlighted in blue. While SJoin does not enumerate $\Delta Q(\mathcal{R}, t)$ in full, to be able to sample from $\Delta Q(\mathcal{R}, t)$, it still needs to track its size, as well as all the sub-join sizes, which are the counters attached to the vertices. Note that the counter at a vertex is exactly the number of paths originating from this vertex and terminating somewhere in column D. These counters allow us to sample from $\Delta Q(\mathcal{R}, t)$ efficiently by a random walk: Starting from $t = (2, 1) \in R_1$, we walk towards column D while picking each next vertex with probability proportional to its counter value. However, tracking these counters is expensive. For example, in Figure 4, inserting $(2, 2)$ into R_3 will update the 3 reachable counters shown in red. In the worst case, a single insertion may update $O(N)$ counters, resulting in a total cost of $O(N^2)$.

4.2 Our Approach

Our approach beats the quadratic cost via two key techniques: approximate counters and dummy join results.

Approximate counters. Instead of trying to maintain the exact counters after each insertion, we maintain a 2-approximation. Specifically, we start from column C. For each value c taken by C, in addition to its exact counter $\mathsf{cnt}(C.c)$, we also maintain an approximate counter $\tilde{\mathsf{cnt}}(C.c) = 2^{\lceil \log_2 \mathsf{cnt}(C.c) \rceil}$, which rounds $\mathsf{cnt}(C.c)$ up to the nearest power of 2. In Figure 5, we show the corresponding approximate counters (in squares) next to the exact counters (in circles). For

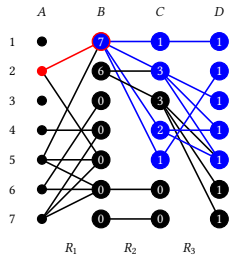


Fig. 3. Inserting $(2, 1)$ into R_1 by SJoin.

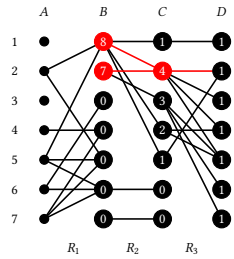


Fig. 4. Inserting $(2, 2)$ into R_3 by SJoin.

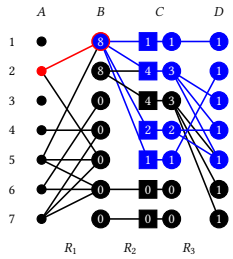


Fig. 5. Inserting $(2, 1)$ into R_1 by our approach.

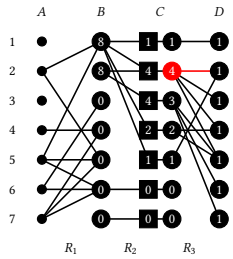


Fig. 6. Inserting $(2, 2)$ into R_3 by our approach.

example, the approximate counter for $\mathsf{cnt}(C.2) = 3$ is $\tilde{\mathsf{cnt}}(C.2) = 4$. Next, for each value b taken by attribute B, its "exact" counter $\mathsf{cnt}(B.b)$ is the sum of the approximate counters in C that are connected with b. For example, $\mathsf{cnt}(B.1) = \tilde{\mathsf{cnt}}(C.1) + \tilde{\mathsf{cnt}}(C.2) + \tilde{\mathsf{cnt}}(C.4) + \tilde{\mathsf{cnt}}(C.5) = 8$. Note that $\mathsf{cnt}(B.b)$ is not necessarily equal to the true counter value for $B.b$; it is "exact" only in the sense that it is the exact sum of the corresponding approximate counters in C. For the line-3 query, the counters shown in Figure 5 are all we need to be able to sample from $\Delta Q(\mathcal{R}, t)$. For more complex queries, this approximation-summation scheme will continue recursively, e.g., we would maintain 2-approximations of $\mathsf{cnt}(B.b)$, denoted $\tilde{\mathsf{cnt}}(B.b)$, and then add them up to get $\mathsf{cnt}(A.a)$, as will be described in full generality in Section 5.

The counters in Figure 5 can be maintained very efficiently. Suppose we insert $(2, 2)$ into R_3, as shown in Figure 6. After the insertion, $\mathsf{cnt}(C.2)$ increases from 3 to 4, but $\tilde{\mathsf{cnt}}(C.2) = 2^{\lceil \log_2 \mathsf{cnt}(2) \rceil}$ remains 4, so no further counter updates are needed. Now we argue that all the counters are updated at most $O(N \log N)$ times after N insertions. First, it is easy to see that all the $\mathsf{cnt}(C.c)$ counters are updated at most N times in total, since each insertion in R_3 updates exactly one $\mathsf{cnt}(C.c)$ counter. The approximate counter $\tilde{\mathsf{cnt}}(C.c)$ is at most twice the value of $\mathsf{cnt}(C.c)$. Therefore, each single $\tilde{\mathsf{cnt}}(C.c)$ is updated at most $O(\log N)$ times since each update doubles its value. Next, any $\mathsf{cnt}(B.b)$ is updated only when some $\tilde{\mathsf{cnt}}(C.c)$ connected with b changes. Hence, the total number of updates to all the $\mathsf{cnt}(B.b)$ counters is $|R_2| \cdot O(\log N) = O(N \log N)$.

Dummy Join Results. While approximate counters can be maintained efficiently, they introduce a difficulty for sampling. As before, suppose we want to sample from $\Delta Q(\mathcal{R}, t)$ for $t = (2, 1) \in R_1$. Now our counter value is $\mathsf{cnt}(B.1) = 8$, which is higher than the true size $|\Delta Q(\mathcal{R}, t)| = 7$. This over counting is in turn due to the over counting in $\tilde{\mathsf{cnt}}(C.2) = 4$. This situation can be modeled as dummy join results, i.e., our approximate counters include $(2, 1, 2, \perp)$ as a dummy result in $\Delta Q(\mathcal{R}, t)$. This is exactly how batched reservoir sampling with predicate fits into the picture, where the predicate filters out dummy tuples. As illustrated in the last row of Figure 2, each new tuple t defines a batch of join results containing both real join results and dummy ones. Since the approximate counters are at most 2 times the true counter values, each batch is at least $\frac{1}{2}$-dense. These counters also allow us to implement $\mathsf{next}(\cdot)$ and $\mathsf{skip}(\cdot)$ in $O(\log N)$ time, and $\mathsf{remain}(\cdot)$ in $O(1)$ time, which will yield the desired running time. The same argument applies for more complex queries, except that the density will depend on the number of relations in the query, as analyzed in Section 5 in detail.

5 Reservoir Sampling Over Acyclic Joins

The framework for reservoir sampling over acyclic joins is presented in Algorithm 5, which adapts the batched reservoir sampling framework of Algorithm 3. The key difference is that batches are

ALGORITHM 5: 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 a random sample of size k without replacement for the join results of Q over tuples seen so far;

```
Initialize index L, S ← ∅, w ← 1, q ← 0, θ ← isReal(·);
while true do
    t ← D.next();
    if t = null then break;
    L ← IndexUpdate(L, t);
    ΔJ ← BatchGenerate(Q, L, t);
    (S, w, q) ← BatchUpdate(S, k, w, q, ΔJ, θ);
```

generated by the BatchGenerate procedure rather than directly from the input stream. For each tuple t in the input stream D, we generate a batch ΔJ that contains all the join results in $\Delta Q(\mathcal{R}, t)$, plus some dummy results. We then process it using the batched reservoir sampling algorithm. Materializing each batch is impractical, as its size could reach $O(N^{\rho^*})$, where ρ^* is the fractional edge cover number of Q. Instead, we maintain a linear-size index $\mathcal{L}$ that supports a retrieve(z) operation to access the item at position z in ΔJ. The index also returns $|\Delta J|$, the size of ΔJ.

Within each batch, we track the current position using a variable pos. Then the three operations required by the batched reservoir sampling algorithm can be implemented as follows:

- remain() returns $|\Delta J| -$ pos, the number of remaining items in the batch;
- skip(i) increments pos by i and returns retrieve(pos), i.e., skips the next i items and access the $(i+1)$th item);
- next() simply returns skip(0).

To apply batched reservoir sampling to any join query Q, it suffices to maintain a linear-size index $\mathcal{L}$ that efficiently supports the retrieve(z) operation and returns $|\Delta J|$. Note that we should not sample from the dummy join results in ΔJ. This is exactly the reason why we must use a reservoir sampling algorithm that supports a 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 participating tuples are real, and dummy otherwise (i.e., at least one participating tuple or partial join result is dummy). Finally, it guarantees that each ΔJ is dense so as to apply Theorem 3.9.

Specifically, we present an index for acyclic joins that achieves the following:

Theorem 5.1. *Given any acyclic join query Q and an initially empty database $\mathcal{R}$. we can maintain an index $\mathcal{L}$ on $\mathcal{R}$ that supports 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 also holds when $Q(\mathcal{R})$ is replaced by the delta query $\Delta Q(\mathcal{R}, t)$ for any $t \notin \mathcal{R}$.

where N is the total number of tuples added to $\mathcal{R}$, but the index does not need the knowledge of N in advance.

Using operations (1) and (2) above, the index immediately solves the dynamic sampling over join problem with an update time of $O(\log N)$ and sampling time of $O(\log N)$, by repeatedly drawing $J[j]$ for a random j until it is a real tuple. With the help of operation (3) and the density

guarantee, it also solves the reservoir sampling over join problem by plugging into Theorem 3.9 with $\alpha = \gamma = O(\log N)$ and $\beta = O(1)$. The number of batches is $m = N$, while the N in Theorem 3.9, which corresponds to the length of the stream of join results, now becomes N^{ρ^*}. Note that ρ^* only depends on the query and is thus considered a constant. The total running time is therefore $O\left(N \log N + k \log N \cdot \log \frac{N^{\rho^*}}{k}\right)$. Putting everything together, we obtain:

Theorem 5.2. *Given any acyclic join Q, an initially empty database $\mathcal{R}$, a sample size k, and a stream of N tuples, Algorithm 5 maintains a uniform sample of size k without replacement for each $Q(\mathcal{R}^i)$, runs in $O\left(N \log N + k \log N \log \frac{N^{\rho^*}}{k}\right)$ expected time, and uses $O(N)$ space.*

Operation (2) can be reduced to operation (3) as follows. Suppose we are given an input join query $Q = (\mathcal{V}, \mathcal{E})$. We construct a new join query $Q' = (\mathcal{V}', \mathcal{E}')$ as follows. We introduce an additional attribute $x_0 \notin \mathcal{V}$ whose domain only contains a special value $\varnothing$, and an additional relation $e_0 \notin \mathcal{E}$ that only contains x_0. Meanwhile, we also choose an arbitrary relation $e_1 \in \mathcal{E}$, and introduce a new relation $e_1' = e_1 \cup \{x_0\}$. Let $\mathcal{V}' = \mathcal{V} \cup \{x_0\}$ and $\mathcal{E}' = \mathcal{E} - \{e_1\} + \{e_1'\} + \{e_0\}$. Given the instance $\mathcal{R}$ for Q, we construct another instance $\mathcal{R}'$ for Q' as follows. For each tuple $t \in R_e$ inserted into $\mathcal{R}$, if $e \in \mathcal{E} - \{e_1\}$, we also insert t into $\mathcal{R}'$; otherwise, we insert $t \circ (\varnothing)$ into $\mathcal{R}'$ for relation e_1', where $\circ$ denotes tuple concatenation and $(\varnothing)$ is the unary tuple containing the value $\varnothing$. At some timestamp, we insert $(\varnothing)$ into $\mathcal{R}'$ for relation e_0. It can be easily checked that there is a one-to-one correspondence between $Q(\mathcal{R})$ and $Q'(\mathcal{R}')$. Moreover, $Q'(\mathcal{R}') = \Delta Q'(\mathcal{R}')$. Hence, an index that supports operation (3) for Q' can also support operation (2) for Q. Henceforth, we will only focus on an index that supports both operations (1) and (3) for general acyclic join queries.

5.1 Index Structure for Two-Table Join

We start with the simple two-table join $R_1(A, B) \bowtie R_2(B, C)$. For this query, the index simply consists of two arrays $X_b = R_1 \ltimes b$ and $Y_b = R_2 \ltimes b$, as well as their sizes, for every $b \in \mathrm{dom}(B)$. Summing over all $b \in \mathrm{dom}(B)$, the whole index uses $O(N)$ space. Without loss of generality, suppose tuple t is inserted into R_1 and let $b = \pi_B t$. Then operation (1) can be easily supported in $O(1)$ time: We just add tuple t to the array X_b. For operation (3), we set $\Delta J = t \bowtie Y_b$. Clearly, ΔJ is 1-dense as there are no dummy tuples, and any $\Delta J[j]$ can be retrieved in $O(1)$ time.

5.2 Index Structure for Line-3 Join

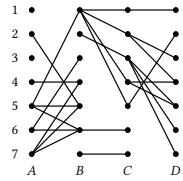
When moving to the line-3 join $R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D)$, the situation is more complicated. In fact, even maintaining an index for just finding the delta query sizes is difficult: It is still an open problem if there is a better algorithm than computing each delta query size from scratch, which takes $O(N)$ time. This is where we need to introduce dummy join results.

Index Structure. For each $b \in \pi_B R_1$, we maintain the degree of b in R_1, i.e., $\mathrm{cnt}(b) = |R_1 \ltimes b|$ and its approximation $\tilde{\mathrm{cnt}}(b) = 2^{\lceil \log_2 \mathrm{cnt}(b) \rceil}$ by rounding $\mathrm{cnt}(b)$ up to the nearest power of 2. Similarly, we maintain $\mathrm{cnt}(c) = |R_3 \ltimes c|$ and $\tilde{\mathrm{cnt}}(c) = 2^{\lceil \log_2 \mathrm{cnt}(c) \rceil}$ for each $c \in \pi_C R_3$. Note that $\tilde{\mathrm{cnt}}(b)$ and $\tilde{\mathrm{cnt}}(c)$ changes at most $O(\log N)$ times for each b and c.

For each value $b \in \pi_B R_2$, we organize the tuples $R_2 \ltimes b$ into at most $\log N$ buckets according to the approximate degree of c, where the ith bucket is

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

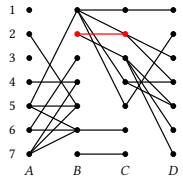
Let $\mathcal{L}_b$ be the list of non-empty buckets. Define $\varphi_i(b) = 2^i \cdot |\Phi_i(b)|$. We also maintain $N_b = \sum_{i \in [\log N]} \varphi_i(b)$ for each value $b \in \pi_B R_2$, which is an upper bound on the number of new join results



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

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

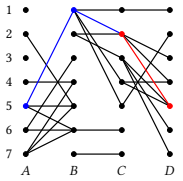
Fig. 7. Initial database instance and state.



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

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

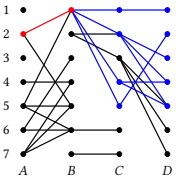
Fig. 8. Inserting $(2,2)$ into R_2.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	$(1,1)(1,5)$	2	8
	1	$(1,4)$	2	
	2	$(1,2)$	4	
2	2	$(2,2)(2,3)$	8	8

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

Fig. 9. Inserting $(2,5)$ into R_3.



b	i	$\Phi_i(b)$	$\varphi_i(b)$	N_b
1	0	$(1,1)(1,5)$	2	8
	1	$(1,4)$	2	
	2	$(1,2)$	4	
2	2	$(2,2)(2,3)$	8	8

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

Fig. 10. Inserting $(2,1)$ into R_1.

if some tuple (a,b) is added to R_1. Symmetrically, for each $c \in \pi_C R_2$, we maintain such a list $\mathcal{L}_c$, as well as the count $N_c = \sum_{i \in [\log N]} \varphi_i(c)$.

Example 5.3. Figures 7–10 show examples of our index structure, where the database instance in Figure 10 is identical to Figure 5. Consider Figure 10 and $b = 1$. In relation $R_2(B,C)$, there are four tuples having a B value of 1, namely $(1,1)$, $(1,2)$, $(1,4)$, and $(1,5)$. The corresponding $\tilde{\text{cnt}}(c)$ values are shown as the approximate counter $\tilde{\text{cnt}}(C.c)$ in Figure 5. These tuples are partitioned into three buckets based on $\tilde{\text{cnt}}(c)$: $\Phi_0(1) = \{(1,1),(1,5)\}$, $\Phi_1(1) = \{(1,4)\}$, and $\Phi_2(1) = \{(1,2)\}$. Thus, we obtain $\varphi_0(1) = 2$, $\varphi_1(1) = 2$, and $\varphi_2(1) = 4$, resulting in $N_1 = \varphi_0(1) + \varphi_1(1) + \varphi_2(1) = 2+2+4 = 8$, which matches the counter for $B.1$ in Figure 5.

Space Usage. As there are $O(N)$ values in $\pi_B R_1$ and $\pi_C R_3$, we need to maintain $O(N)$ degrees and their approximations in total. For each $b \in \pi_B R_2$, 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 $b \in \pi_B R_2$, the total size is $O(N)$. A similar argument applies to $c \in \pi_C R_2$.

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 $i = \log_2 \tilde{\text{cnt}}(c)$ and to $\Phi_j(c)$ for $j = \log_2 \tilde{\text{cnt}}(b)$, which just takes $O(1)$ time.

If $t = (c,d) \in R_3$ (the $t \in R_1$ case is similar), we increase $\text{cnt}(c)$ by 1, and update $\tilde{\text{cnt}}(c)$ if needed. If $\tilde{\text{cnt}}(c)$ changes, for each $(b,c) \in (R_2 \ltimes c)$, we remove (b,c) from $\Phi_{i-1}(b)$ and add (b,c) to $\Phi_i(b)$, where $i = \log_2 \tilde{\text{cnt}}(c)$. Whenever some $\Phi_i(b)$ changes, we update N_b and $\varphi_i(b)$ in $O(1)$ time accordingly. The time for updating N_b is the same as that for $\Phi_i(b)$. We point out that this update is only triggered when $\tilde{\text{cnt}}(c)$ changes, which happens at most $O(\log N)$ times. Thus, the total update cost is $O(N \log N)$, and the amortized cost is $O(\log N)$.

Example 5.4. Consider Figure 9 as an example. Inserting $(2,5)$ into R_3 increases $\text{cnt}(2)$ from 2 to 3, which in turn changes $\tilde{\text{cnt}}(2)$ from 2 to 4. Consequently, we move $(1,2)$ from $\Phi_1(1)$ to $\Phi_2(1)$ and $(2,2)$ from $\Phi_1(2)$ to $\Phi_2(2)$, updating $\varphi_1(1)$, $\varphi_2(1)$, $\varphi_1(2)$, $\varphi_2(2)$, N_1, and N_2 accordingly. The changes of the index structures in Figures 8-10 are highlighted in red.

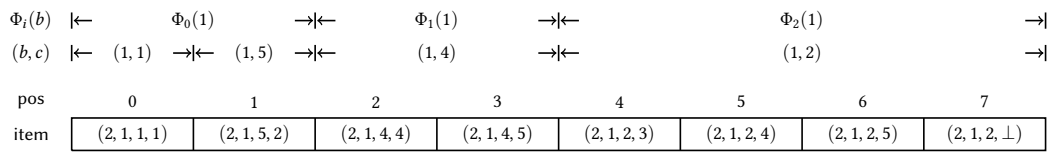


Fig. 11. An illustration of ΔJ generated in Figure 10.

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

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

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, with size $|\Delta J| = \mathsf{cnt}(b) \cdot \mathsf{cnt}(c)$. Next, consider $t = (a,b) \in R_1$. Consider bucket $\Phi_i(b) \in \mathcal{L}_b$ in ascending order of i. For each $(b,c) \in \Phi_i(b)$, define a mini-batch that consists of all tuples in $R_3 \ltimes c$, followed by $\tilde{\mathsf{cnt}}(c) - \mathsf{cnt}(c)$ dummy tuples. We concatenate these mini-batches to form the mini-batch for the bucket. We join each mini-batch with the tuple (a,b) and concatenate the results to form ΔJ. This ΔJ is $\frac{1}{2}$-dense, since each mini-batch is $\frac{1}{2}$-dense and then we invoke Lemma 3.6. Moreover, $|\Delta J| = N_b$ and can be returned in $O(1)$ time. The case $t \in R_3$ is similar.

Example 5.5. In Figures 8–10, real join results in each ΔJ are highlighted in blue. In Figure 8, since no tuple in R_1 has a B value of 2, inserting $(2,2)$ into R_2 yields an empty ΔJ. In Figure 9, a batch of size 1 is generated, containing the join result $(5,1,2,5)$.

The batch ΔJ from Figure 10 is detailed in Figure 11. After inserting $(2,1)$ into R_1, we bind A to 2 and B to 1. We have $\mathcal{L}_1 = \{\Phi_0(1), \Phi_1(1), \Phi_2(1)\}$. For $\Phi_0(1)$, it contains two tuples: $(1,1)$ and $(1,5)$. Since $\tilde{\mathsf{cnt}}(1) = \mathsf{cnt}(1) = 1$, tuple $(1,1)$ produces the mini-batch $\langle(1,1)\rangle$ with no dummy tuples. Similarly, $(1,5)$ produces $\langle(5,2)\rangle$. Concatenating these yields the mini-batch $\langle(1,1),(5,2)\rangle$ for $\Phi_0(1)$. Combining with $(2,1)$, we obtain $\langle(2,1,1,1),(2,1,5,2)\rangle$. For tuple $(1,2) \in \Phi_2(1)$, with $\tilde{\mathsf{cnt}}(2) = 4$ and $\mathsf{cnt}(2) = 3$, the mini-batch is $\langle(2,3),(2,4),(2,5),(2,\perp)\rangle$. By concatenating the mini-batches for $\Phi_0(1)$, $\Phi_1(1)$, and $\Phi_2(1)$, we derive ΔJ, as shown in Figure 11.

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 $(z_1, z_2) \in [|R_1 \ltimes b|] \times [|R_3 \ltimes c|]$ 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 $R_1 \ltimes b$ 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 $\sum_{i' \leqslant i-1} \varphi_{i'}(b) < z + 1 \leqslant \sum_{i' \leqslant i} \varphi_{i'}(b)$.

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

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

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 $\Phi_i(b) \neq \varnothing$, the value of i, j, ℓ can be computed in $O(\log N)$ time. So the retrieve operation takes $O(\log N)$ time.

Example 5.6. Suppose we want to retrieve the tuple at position pos $= 1$ from ΔJ in Figure 11. Setting $z = 1$, we first derive $i = 0$ as $\varphi_0(1) = 2$. This indicates that the involved (b, c) tuple is from $\Phi_0(1)$. Next, we get $j = 1$ from the formula, so the tuple t' is the second tuple in $\Phi_0(1)$, i.e., $(1, 5)$. Then, we derive $l = 0$ and retrieve the first tuple from $R_3 \ltimes t'$, which is $(5, 2)$. Combining these, the join result is $(2, 1, 5, 2)$. Now consider retrieving the tuple at position pos $= 7$. Setting $z = 7$, we follow the same steps to derive $i = 2$, $j = 0$, and $l = 3$. Since $|R_3 \ltimes t'| = 3$, we determine that the tuple at position pos $= 7$ is a dummy tuple.

5.3 Index Structure for Acyclic Joins

Finally, we generalize the line-3 algorithm to an arbitrary acyclic join $Q = (\mathcal{V}, \mathcal{E})$. As observed by [34], Q has a special width-1 TD $(\mathcal{T}, \chi)$ such that there is a one-to-one correspondence between the relations in $\mathcal{E}$ and the bags in $\mathcal{T}$. Such a TD is also called a *join tree* of Q. For simplicity, we also use $e \in \mathcal{E}$ to denote the node $u \in \text{nodes}(\mathcal{T})$ corresponding to e, i.e., $\chi(u) = e$. Note that $\mathcal{T}$ 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 $\text{key}(e) = e \cap p_e$ 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 rooted at 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 $t \in \text{dom}(x)$, we can get the list of tuples $R_e \ltimes t$ in $O(1)$ time. For each node $e \in \text{nodes}(\mathcal{T})$ and tuple $t \in \pi_{\text{key}(e)} R_e$, we maintain a upper bound $\text{cnt}[\mathcal{T}, e, t]$ 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 $\text{key}(e)$ matches t:

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

Note that this definition depends on $\tilde{\text{cnt}}(\cdot)$, which is recursively defined as $\tilde{\text{cnt}}[\mathcal{T}, e, t] = 2^{\lceil \log_2 \text{cnt}[\mathcal{T}, e, t] \rceil}$ by rounding $\text{cnt}[\mathcal{T}, e, t]$ up to the nearest power of 2. We point out an important property of $\tilde{\text{cnt}}(\cdot)$ in Lemma 5.7, which indicates that $\tilde{\text{cnt}}[\mathcal{T}, e, t]$ is a constant-approximation of the degree of t in $\mathcal{T}_e$.

Lemma 5.7. *For a join tree $\mathcal{T}$, node e and tuple $t \in \pi_{key(e)} R_e$,*

$$\tilde{\text{cnt}}[\mathcal{T}, e, t] \leqslant 2^{|\text{nodes}(\mathcal{T}_e)|} \cdot \left| \left(\ltimes_{e' \in \text{nodes}(\mathcal{T}_e)} R_{e'} \right) \ltimes t \right|.$$

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

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

ALGORITHM 6: IndexUpdate$(\mathcal{T}, e, t, d_0)$

Input : A join tree $\mathcal{T}$ for Q, a node e and tuple $t \in R_e$, an approximate degree d_0 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 $d_1 \leftarrow \prod_{e' \in C_e} \tilde{\mathsf{cnt}}[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t]$;
3 **if** $d_0 > 0$ **then**
4 $\quad i \leftarrow \log_2(d_0)$;
5 $\quad \Phi_{i,e}(t_e) \leftarrow \Phi_{i,e}(t_e) - \{t\}$;
6 $i \leftarrow \log_2(d_1)$;
7 $\Phi_{i,e}(t_e) \leftarrow \Phi_{i,e}(t_e) \cup \{t\}$;
8 $j \leftarrow \tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$;
9 $\mathsf{cnt}[\mathcal{T}, e, t_e] \leftarrow \mathsf{cnt}[\mathcal{T}, e, t_e] + d_1 - d_0$;
10 $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e] \leftarrow 2^{\lceil \log_2 \mathsf{cnt}[\mathcal{T}, e, t_e] \rceil}$;
11 **if** $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t_e]$ *changes and* p_e *is not the root* **then**
12 $\quad$ **foreach** $t' \in R_{p_e} \ltimes t_e$ **do**
13 $\quad\quad d_2 \leftarrow j \cdot \prod_{e' \in C_{p_e} - \{e\}} \tilde{\mathsf{cnt}}[\mathcal{T}, e', \pi_{\mathsf{key}(e')} t']$;
14 $\quad\quad$ IndexUpdate$(\mathcal{T}, p_e, t', d_2)$;

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

Together with the fact that $\left|\left(\ltimes_{e' \in \mathsf{nodes}(\mathcal{T}_e)} R_{e'}\right) \ltimes t\right| \leqslant \left|\ltimes_{e' \in \mathsf{nodes}(\mathcal{T}_e)} R_{e'}\right| \leqslant N^{|\mathsf{nodes}(\mathcal{T}_e)|}$, we obtain $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t] \leqslant (2N)^{|\mathsf{nodes}(\mathcal{T}_e)|}$, which implies that $\tilde{\mathsf{cnt}}[\mathcal{T}, e, t]$ can only change 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 $|\mathsf{nodes}(\mathcal{T}_e)| \cdot \log 2N$ 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 $i \in [|\mathsf{nodes}(\mathcal{T}_e)| \cdot \log 2N]$. We also maintain $N_t = \sum_{i \in [|\mathsf{nodes}(\mathcal{T}_e)| \cdot \log 2N]} \varphi_{i,e}(t)$ 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. 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 lookup in line 12 of Algorithm 6. 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 6, the procedure IndexUpdate takes as input the join tree $\mathcal{T}$, a node e, tuple $t \in R_e$ and an integer $d_0 \geqslant 0$ (indicating the approximate degree of t in $\mathcal{T}_e$ before the update). The update proceeds recursively. We first compute the approximate degree of t in $\mathcal{T}_e$ after update, denoted as d_1. For simplicity, denote $t_e = \pi_{\mathsf{key}(e)} t$. We then remove t from the old bucket if it exists (lines 3–5) and insert t into the new bucket (lines 6–7). We increase $\mathsf{cnt}[\mathcal{T}, e, t_e]$ by $d_1 - d_0$ (line 9), and update

ALGORITHM 7: BatchGenerate$(\mathcal{T}, e, t)$

Input : A join tree $\mathcal{T}$ for Q, a node e and a tuple $t \in R_e$ or $t \in \pi_{\text{key}(e)} R_e$;
Output: A $O(1)$-dense batch $\Delta J \supseteq \Delta Q(\mathcal{R}, t)$;

```
1  x ← 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 ∈ C_e do
5          B_i ← BatchGenerate(T, e_i, π_key(e_i) t);
6      return {t} × (×_{e_i ∈ C_e} B_i);
7  for t' ∈ R_e ⋉ t do B_{t'} ← BatchGenerate(T, e, t');
8  d ← c̃nt[T, e, t] − cnt[T, e, t];
9  return concatenation of B_{t'} for t' ∈ R_e ⋉ t, followed by d dummy tuples;
```

$\tilde{\text{cnt}}[\mathcal{T}, e, t_e]$ if needed. If $\tilde{\text{cnt}}[\mathcal{T}, e, t_e]$ changes, we might need to propagate the updates upward (lines 12–14). If p_e is not the root, for each tuple $t' \in R_{p_e} \ltimes t_e$, we compute the approximate degree of t' in $\mathcal{T}_{p_e}$ before update (line 13), denoted as d_2 and invoke this whole procedure recursively (line 14).

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. Whenever some $\Phi_{i,e}(t)$ changes, we update N_t and $\varphi_{i,e}(t)$ accordingly. The time for this update is the same as that for updating $\Phi_{i,e}(t)$. The key observation is that this update is only triggered when $\tilde{\text{cnt}}[\mathcal{T}, e', \pi_{\text{key}(e')} t]$ changes for some $e' \in C_e$, which happens at most $O(\log N)$ times. Thus, the total update cost is

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

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 $t \in \pi_{\text{key}(e)} R_e$, as described in Algorithm 7. 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 first call is BatchGenerate$(\mathcal{T}, e, t)$, where $\mathcal{T}$ is the join tree rooted at node e. In each recursive call, Algorithm 7 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 $\Delta Q(\mathcal{R}, \pi_{\text{key}(e_i)} t)$ for each child $e_i \in C_e$. The batch ΔJ also follows the same way. We recursively generate a batch for $\pi_{\text{key}(e_i)} t$ in $\mathcal{T}_{e_i}$ for each $e_i \in C_e$, and return their cross product as ΔJ.
- **Case 3:** $t \in \pi_{\text{key}(e)} R_e$. We recursively generate a batch for every tuple $t' \in R_e \ltimes t$ in $\mathcal{T}_e$ and concatenate these batches with $\tilde{\text{cnt}}[\mathcal{T}, e, t] - \text{cnt}[\mathcal{T}, e, t]$ dummy elements at the end as ΔJ.

It can be easily shown by induction that

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

ALGORITHM 8: 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$(\mathcal{T}, e, t)$;

$x \leftarrow \mathsf{supp}(t)$;
if *e is a leaf node* **then**
  if $z \geqslant \mathsf{cnt}[\mathcal{T}, e, t]$ **then return** dummy;
  else return the element at position z in $R_e \ltimes t$;
if $e = x$ **then**
  $C_e \leftarrow \{e_1, e_2, \cdots, e_m\}$;
  foreach $i \in [1, m]$ **do** $t_i \leftarrow \pi_{\mathsf{key}(e_i)} t$;
  Find $(z_1, z_2, \cdots, z_m) \in \times_{i=1}^{m} [\tilde{\mathsf{cnt}}[\mathcal{T}, e_i, t_i]]$ such that $z = \sum_{i \in [1 \ldots m]} \left(z_i \cdot \prod_{j>i} \tilde{\mathsf{cnt}}[\mathcal{T}, e_j, t_j] \right)$;
  foreach $i \in [1, m]$ **do**
    $t_i' \leftarrow$ RETRIEVE$(\mathcal{T}, e_i, t_i, z_i)$;
    if *t_i' is dummy* **then return** dummy;
  return $t \bowtie t_1' \bowtie t_2' \bowtie \cdots \bowtie t_m'$;
else
  if $z \geqslant \mathsf{cnt}[\mathcal{T}, e, t]$ **then return** dummy;
  Find i such that $\sum_{i' \leqslant i-1} \varphi_{i',e}(t) < z + 1 \leqslant \sum_{i' \leqslant i} \varphi_{i',e}(t)$;
  $j \leftarrow \left\lfloor \left(z - \sum_{i' \leqslant i-1} \varphi_{i',e}(t) \right) / 2^i \right\rfloor$;
  $\ell \leftarrow z - \sum_{i' \leqslant i-1} \varphi_{i',e}(t) - 2^i \cdot j$;
  $t' \leftarrow$ the element at position j in $\Phi_{i,e}(t)$;
  return RETRIEVE$(\mathcal{T}, e, t', \ell)$;

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

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

This holds trivially for **Case 1**. For **Case 2**, we assume that the batch generated for tuple $\pi_{\mathsf{key}(e_i)} t$ is $(\frac{1}{2})^{2 \cdot |\mathsf{nodes}(\mathcal{T}_{e_i})| - 1}$-dense. Implied by Lemma 3.7, this ΔJ is $(\frac{1}{2})^{2 \cdot |\mathsf{nodes}(\mathcal{T}_e)| - 2}$-dense, since

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

For **Case 3**, we assume the batch generated for each tuple $t' \in R_e \ltimes t$ is $(\frac{1}{2})^{2 \cdot |\mathsf{nodes}(\mathcal{T}_e)| - 2}$-dense. Their concatenation is also $(\frac{1}{2})^{2 \cdot |\mathsf{nodes}(\mathcal{T}_e)| - 2}$-dense, implied by Lemma 3.6. The ΔJ is $(\frac{1}{2})^{2 \cdot |\mathsf{nodes}(\mathcal{T}_e)| - 1}$-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 3.8.

Retrieve. Finally, we describe how to retrieve the join result at position z in the batch generated by BATCHGENERATE. As described in Algorithm 8, 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 $\pi_{\text{key}(e_i)}t$ 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 $t \in \pi_{\text{key}(e)}R_e$. 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.

5.4 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 9.

Grouping. In a join tree $\mathcal{T}$, consider a non-root internal node R_e with child nodes $\{e_1, e_2, \ldots, e_m\}$. Let $e_\dagger = \text{key}(e) \cup \text{key}(e_1) \cup \cdots \cup \text{key}(e_m)$ denote the set of join attributes. If $e - e_\dagger \neq \emptyset$, we can *group* tuples in R_e by $e_\dagger$.

Specifically, we introduce a new node $e_\dagger$ with a corresponding relation $R_{e_\dagger}$. We replace node e with $e_\dagger$ in $\mathcal{T}$ to obtain a new join tree $\mathcal{T}'$. The node e becomes a child of $e_\dagger$, and all previous children of e are reattached to $e_\dagger$. We enforce the invariant $R_{e_\dagger} = \pi_{e_\dagger}R_e$, which requires only an additional check upon insertions into R_e.

Since $R_e \bowtie R_{e_\dagger} = R_e$, it follows that $Q \bowtie R_{e_\dagger} = Q$. Running our algorithm on $Q \bowtie R_{e_\dagger}$ using $\mathcal{T}'$ as the join tree ensures that the maintained reservoir S remains a uniform random sample of size k from the result of Q.

This optimization can bring much benefit in index update. Let e' be the parent of e in $\mathcal{T}$ and we apply this optimization on e'. In the new join tree $\mathcal{T}'$, the parent of e is now $e'_\dagger$. When applying Algorithm 6 to process updates on e, in lines 12–14, instead of propagating updates for every tuple $t' \in R_{e'} \ltimes t_e$, we now propagate updates only for tuples in $R_{e'_\dagger} \ltimes t_e$, where $R_{e'_\dagger} = \pi_{e'_\dagger}R_{e'}$. Since $|\pi_{e'_\dagger}R_{e'} \ltimes t_e| \leqslant |R_{e'} \ltimes t_e|$, we may see a significant reduction in the number of propagated updates.

Foreign-keys. When foreign-key join exists, similar to [36], 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 join with t_j, and insert $t_{ij} = t_i \bowtie t_j$ into R_{ij}.

6 Extension to Free-Connex Queries

Beyond join queries, we next move to join-project queries, where "project" refers to a distinct projection. Our sampling index proposed for acyclic joins can be extended to the class of *free-connex* queries. For our development, we use an equivalent definition of free-connex queries in [32]. A free-connex query $\pi_y Q$ has a special width-1 TD $(\mathcal{T}, \chi)$ such that for each node $u \in \text{nodes}(\mathcal{T})$, there

always exists some relation $e \in \mathcal{E}$ such that $\chi(u) \subseteq e$. Recall in Definition 2.1 that for each relation $e \in \mathcal{E}$, there always exists a node $u \in \text{nodes}(\mathcal{T})$ such that $e \in \chi(u)$. Hence, we can distinguish each node $u \in \text{nodes}(\mathcal{T})$ as *original* if there exists some relation $e \in \mathcal{E}$ such that $\chi(u) = e$, and *generalized* otherwise. We use r to denote the root, $\mathcal{T}_u$ for the subtree rooted at node u, C_u for the set of children of node u, and p_u for the parent of node u. If u is a leaf, $C_u = \emptyset$; for the root r, $p_r = \emptyset$. Let $\text{key}(u) = \chi(u) \cap \chi(p_u)$ be the *join key* between node u and p_u. As pointed out [32], $(\mathcal{T}, \chi)$ satisfies the additional properties as follows:

- **(cover property)** each input relation in $\mathcal{E}$ corresponds to a distinct node in $\mathcal{T}$; moreover, each leaf node of $\mathcal{T}$ corresponds to an input relation in $\mathcal{E}$;
- **(guard property)** for a generalized node u, $\chi(u) \subseteq \chi(u')$ for every child node u' of u;
- **(above property)** an original node does not appear above any generalized node;
- **(connex property)** there exists a connected subset $\mathcal{E}_{\text{con}}$ of $\mathcal{T}$ such that (i) $r \in \mathcal{E}_{\text{con}}$; (ii) $\text{key}(u) \subseteq \mathbf{y}$ for all $u \in \mathcal{E}_{\text{con}}$; (iii) $\mathbf{y} \subseteq \bigcup_{u \in \mathcal{E}_{\text{con}}} \chi(u)$;

Note that under this definition, $\mathcal{E}_{\text{con}}$ is slightly different form our previous definition in Section 2.3.

Our index for free-connex queries is a combination of the sampling index proposed for acyclic joins in Section 5 and the index proposed for maintaining free-connex queries under insertion-only updates in Ref. [32].

Index Structure. Given a free-connex query $\pi_{\mathbf{y}}\boldsymbol{Q}$, we choose a width-1 free-connex TD $(\mathcal{T}, \chi)$ for $\pi_{\mathbf{y}}\boldsymbol{Q}$ as characterized by [32].

Our index consists of two parts. **Part I.** For each node $u \in \text{nodes}(\mathcal{T})$, it maintains two views: a semi-join view $V_s(u)$ and a projection view $V_p(u)$, defined recursively as follows. Every non-root node $u \in \text{nodes}(\mathcal{T})$ has a *projection view* $V_p(u) := \pi_{\text{key}(u)} V_s(u)$. To define the *semi-join view* $V_s(u)$, it distinguishes three cases. (1) If u is a leaf node, there exists some input relation $e \in \mathcal{E}$ such that $\chi(u) = e$. Then, $V_s(u) := R_e$. (ii) If u is an internal original node, i.e., there exists some input relation $e \in \mathcal{E}$ such that $\chi(u) = e$, then $V_s(u) := R_e \ltimes V_p(u_1) \ltimes \cdots \ltimes V_p(u_\ell)$, where $C_u = \{u_1, u_2, \ldots, u_\ell\}$ are the children of u. (iii) If u is an internal generalized node, then $V_s(u) := V_p(u_1) \cap \cdots \cap V_p(u_\ell)$ since all the $V_p(u_i)$'s have the same attributes $\text{key}(u_i) = \chi(u_i) \cap \chi(u) = \chi(u)$ for every $i \in [1, \ell]$. **Part II.** For each $u \in \mathcal{E}_{\text{con}}$, we define a new relation R^* corresponding to $e^* = \chi(u) \cap \mathbf{y}$. We then build the index and apply our algorithms for acyclic joins in Section 5 over the join of all R^* relations.

Index Update. We distinguish the following three cases for maintaining the **Part I** of our index:

- **Case 1:** when there is a tuple t inserted into $V_s(u)$ for some u, we will insert it into $V_p(u)$ if $\pi_{\text{key}(u)} t \notin V_p(u)$. If an insertion happens to $V_p(u)$, we will propagate an update in **Case 2**.
- **Case 2:** when there is a tuple t inserted into some $V_p(u_i)$, where u_i be a child of u, we update $V_s(u)$ by distinguishing two more cases. If u is a generalized node, and $t \in V_p(u_i)$ for each $i \in [1, \ell]$, we add t into $V_s(u)$. Otherwise, u is an original node. Let $e \in \mathcal{E}$ be the corresponding relation. For each tuple $t' \in R_e \ltimes t$, if $\pi_{\text{key}(u_j)} t' \in V_p(u_j)$ for each $j \in [1, \ell]$, we add t' into $V_s(u)$. Whenever an insertion happens to $V_s(u)$, we will propagate it in **Case 1**.
- **Case 3:** when there is a tuple t inserted to an input relation R_e, we will add t to $V_s(u)$ if $\pi_{\text{key}(u_i)} t \in V_p(u_i)$ for each $i \in [1, \ell]$, where u is the node corresponding to e. If an insertion happens to $V_s(u)$, we will propagate an update in **Case 1**.

Whenever a tuple t is inserted into $V_s(u)$ for some node $u \in \mathcal{E}_{\text{con}}$, if $\pi_{\chi(u) \cap \mathbf{y}} t \notin \pi_{\chi(u) \cap \mathbf{y}} V_s(u)$ before the insertion of t, we simply invoke the update procedure for maintaining **Part II** of our index with a tuple $\pi_{\mathbf{y}} t$ being inserted into R^*, where R^* is the created relation corresponding to u.

Example 6.1. Consider an example of free-connex query $\pi_{\mathbf{y}}\boldsymbol{Q} :=$

$$\pi_{X_3, X_4, X_5, X_6} R_1(X_1, X_2) \bowtie R_2(X_2, X_3, X_4) \bowtie R_3(X_4, X_5) \bowtie R_4(X_5, X_6) \bowtie R_5(X_6, X_7),$$

which is equivalent to the following SQL query:

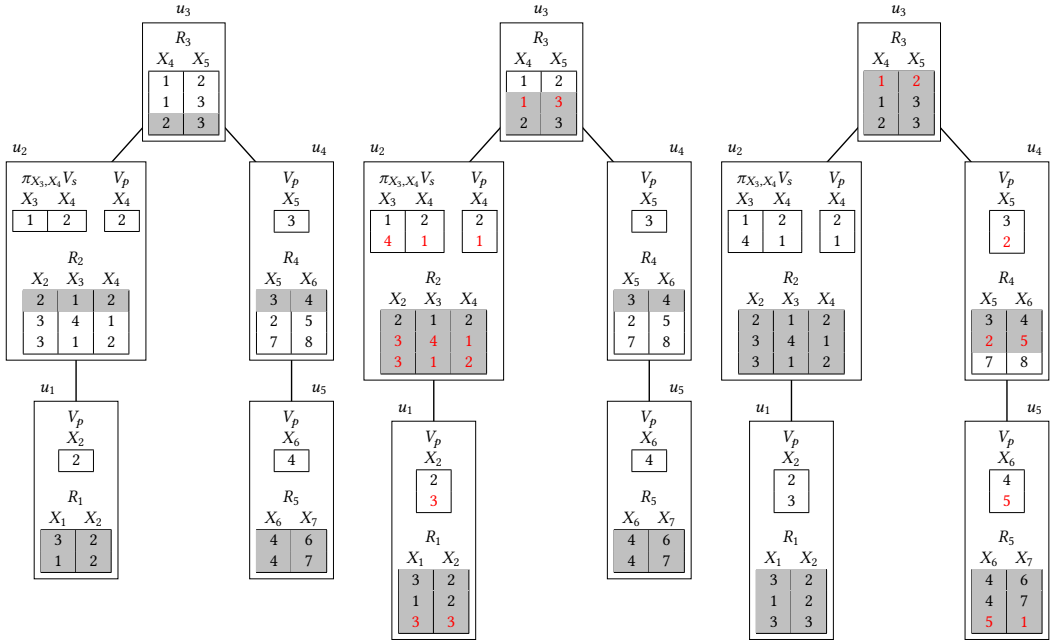


Fig. 12. Initial state.

Fig. 13. Inserting $(3, 3)$ into R_1.

Fig. 14. Inserting $(5, 1)$ into R_5.

```
SELECT DISTINCT X3, X4, X5, X6
FROM R1 NATURAL JOIN R2 NATURAL JOIN R3 NATURAL JOIN R4 NATURAL JOIN R5
```

Figure 12 shows the join tree for maintaining the **Part I** with $\mathcal{E}_{\text{con}} = \{u_2, u_3, u_4\}$, and the initial database instance. The tuples included in the semi-join views are shaded. In this example, we create three new relations, $R_2^*(X_3, X_4)$, $R_3^*(X_4, X_5)$, and $R_4^*(X_5, X_6)$. We apply the algorithms in Section 5 to the full join query $R_2^* \bowtie R_3^* \bowtie R_4^*$.

Figure 13 shows the updated state after inserting $(3, 3)$ into R_1, with changes marked in red. After the insertion, $(4, 1)$ enters $\pi_{X_3,X_4}V_s(u_2)$ and $(1, 3)$ enters $V_s(u_3)$. We then update the index structure using the method from Section 5, treating $(4, 1)$ and $(1, 3)$ as if they were directly inserted into R_2^* and R_3^*, respectively. Figure 14 illustrates the state after inserting $(5, 1)$ to R_5. As a result, $(2, 5)$ enters $V_s(u_4)$ and $(1, 2)$ enters $V_s(u_3)$. We update the index structure again, following the same approach as if $(2, 5)$ and $(1, 2)$ were inserted into R_4^* and R_3^*, respectively.

Analysis. It has been proved that the index in **Part I** uses $O(N)$ space. Note that each relation in $\mathcal{E}_{\text{con}}$ has at most $O(N)$ tuples. Hence, the index in **Part II** also uses $O(N)$ space. and can be updated in $O(1)$ amortized time for each input tuple. Moreover, the index in **Part I** can be updated in $O(1)$ amortized time for each insertion. It has been shown that there are $O(N)$ tuples in all relations from $\mathcal{E}_{\text{con}}$. Putting everything together, we obtain:

Theorem 6.2. *Given any free-connex query $\pi_{\mathbf{y}}\mathbf{Q}$, an initially empty database $\mathcal{R}$, a sample size k, and a stream of N tuples, our algorithm can maintain a uniform sample of size k without replacement for each $\pi_{\mathbf{y}}\mathbf{Q}(\mathcal{R}^i)$, uses $O(N)$ space and runs in $O\left(N \log N + k \log N \log \frac{N^{\rho^*}}{k}\right)$ expected time, where ρ^* is the fractional edge covering number of $\mathbf{Q}[\mathbf{y}]$.*

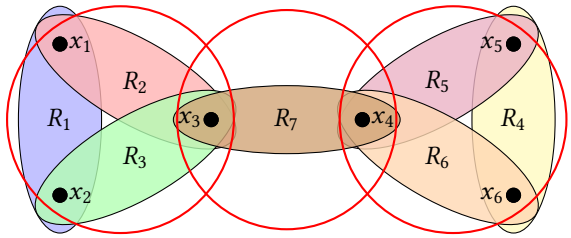


Fig. 15. The dumbbell join $Q = R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3) \bowtie R_4(x_5, x_6) \bowtie R_5(x_4, x_5) \bowtie R_6(x_4, x_6) \bowtie R_7(x_3, x_4)$ with a free-connex TD $(\mathcal{T}, \chi)$ illustrated as the red circle. It has width$(\mathcal{T}, \chi) = 1.5$ since the triangle join $R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3)$ and $R_4(x_5, x_6) \bowtie R_5(x_4, x_5) \bowtie R_6(x_4, x_6)$ have the fractional edge covering number as $\rho^* = 1.5$.

7 Extension to Cyclic Queries

In this section, we show how to handle arbitrary cyclic join-project queries using our algorithm in Section 6 via tree decompositions. Please see an example in Figure 15. Given a join-project query $\pi_{\mathbf{y}} Q$, let $(\mathcal{T}, \chi)$ be an arbitrary free-connex TD . Each node $u \in \text{nodes}(\mathcal{T})$ derives a subquery $\pi_{\mathbf{y} \cap \chi(u)} Q[\chi(u)]$. Given an instance $\mathcal{R}$, we define the sub-instance for each node $u \in \text{nodes}(\mathcal{T})$ as $\mathcal{R}_u = \{\pi_{e \cap \chi(u)} R_e : e \in \mathcal{E}, e \cap \chi(u) \neq \emptyset\}$. We then build the index for the free-connex query $\pi_{\mathbf{y}} Q_{\mathcal{T}}$ for $Q_{\mathcal{T}} = (\mathcal{V}, \{\chi(u) : u \in \text{nodes}(\mathcal{T})\})$ over the instance $\mathcal{R}_{\mathcal{T}} = \{\mathcal{R}_u : u \in \text{nodes}(\mathcal{T})\}$, as described in Section 6.

Index Update. Suppose a tuple t is inserted into relation R_e. For each node $u \in \text{nodes}(\mathcal{T})$ with $e \cap \chi(u) \neq \emptyset$, let $t_u = \pi_{e \cap \chi(u)} t$. If $t_u \notin \pi_{e \cap \chi(u)} R_e$, we add the tuple t_u to the instance $\mathcal{R}_u$ and update the query result at u, i.e., from $Q[\chi(u)](\mathcal{R}_u)$ to $Q[\chi(u)](\mathcal{R}_u \cup \{t_u\})$. Let $\Delta_u(t) = Q[\chi(u)](\mathcal{R}_u) \ltimes t_u$ be the delta of $Q[\chi(u)]$ induced by t. If $\Delta_u(t) \neq \emptyset$, for each tuple $t' \in \Delta_u(t)$, we invoke the update procedure on the index built for the free-connex query $Q_{\mathcal{T}}$ over instance $\mathcal{R}_{\mathcal{T}}$, as described in Section 6.

Time Complexity. We next analyze the time complexity. For each tuple t inserted into R_e, the size of the delta results induced by t, as well as the time required for computing these delta results, can be bounded by $O\left(|\mathcal{R}_u \ltimes t_u|^{\rho^*(Q[\chi(u)])}\right)$ time. Summing over all inserted tuples and nodes in $\mathcal{T}$, the time complexity of is

$$\sum_{u \in \text{nodes}(\mathcal{T})} \sum_{e \in \mathcal{E}} \sum_{t_u \in \pi_{e \cap \chi(u)} R_e} |\mathcal{R}_u \ltimes t_u|^{\rho^*(Q[\chi(u)])} \leqslant \sum_{u \in \text{nodes}(\mathcal{T})} \sum_{e \in \mathcal{E}} |\mathcal{R}_u|^{\rho^*(Q[\chi(u)])} = O\left(N^{\text{width}(\mathcal{T}, \chi)}\right),$$

where the first inequality is implied by the facts that $\rho^*(Q[\chi(u)]) \geqslant 1$, and $\sum_{t_u \in \pi_{e \cap \chi(u)} R_e} |\mathcal{R}_u \ltimes t_u| = |\mathcal{R}_u|$, and the second inequality is implied by the fact that $|\mathcal{R}_u| = O(N)$ and the query size is $O(1)$. Moreover, the input size for the derived instance $\mathcal{R}_{\mathcal{T}}$ is $O\left(N^{\text{width}(\mathcal{T}, \chi)}\right)$, which is also the size of the simulated input stream to the free-connex query $Q_{\mathcal{T}}$. Plugging into Theorem 6.2, we can obtain the cost of maintaining the index built for $Q_{\mathcal{T}}$ over instance $\mathcal{R}_{\mathcal{T}}$.

Space Usage. Note that the total number of input tuples inserted into each node of $\mathcal{T}$ is $O(N^{\text{width}(\mathcal{T}, \chi)})$. Following the same analysis of free-connex queries in Section 6, the space used by our index is proportional to the total number of tuples in each node of $\mathcal{T}$, i.e., $O(N^{\text{width}(\mathcal{T}, \chi)})$.

By taking the TD with the minimum width, we obtain:

Theorem 7.1. *Given an arbitrary join-project query $\pi_{\mathbf{y}} Q$, an initially empty database $\mathcal{R}$, a sample size k, and a stream of N tuples, Algorithm 5 maintains a sample of size k without replacement for each $\pi_{\mathbf{y}} Q(\mathcal{R}^i)$, uses $O\left(N^{fn\text{-}fhtw}\right)$ space and runs in $O\left(N^{fn\text{-}fhtw} \cdot \log N + k \cdot \log N \cdot \log \frac{N^{\rho^*}}{k}\right)$ expected*

time, where fn-fhtw is the free-connex fractional hypertree width of Q and ρ^* is the fractional edge covering number of $Q[\mathbf{y}]$.

Finally, if a relation has a selection condition, we can easily filter the tuples in the stream with the condition in $O(1)$ time per tuple. Therefore, our complete algorithm can maintain a reservoir sample for any SPJ query over a stream of tuples with the claimed running time above.

8 Applications

Random sampling is an important tool in data analytics with many applications. Since our reservoir sampling algorithm maintains a sample continuously, it can be used in a variety of continuous estimation problems over data defined by SPJ queries. In this section, we present three such examples: mean estimation, selectivity estimation, and query cardinality estimation.

8.1 Mean Estimation

Given an SPJ query $\pi_{\mathbf{y}}Q$, let $w : \mathrm{dom}(\mathbf{y}) \to \mathbb{R}$ define a numerical *weight* of t. In the continuous mean estimation problem, we wish to estimate

$$\mu_i = \frac{1}{|\pi_{\mathbf{y}}Q(\mathcal{R}^i)|} \sum_{t \in \pi_{\mathbf{y}}Q(\mathcal{R}^i)} w(t)$$

continuously over all i.

There are two general approaches to mean estimation over a join. Ripple join [21] takes a sample from each relation, and joins the samples. Although the join of the samples is not a sample of the join [14], the former can still be used to derive an unbiased estimator of the mean [21]. The other approach is to directly sample from the join, and then use the sample mean as an estimator of the mean. Both approaches can be applied in the streaming setting: The first approach can be implemented by running the classical reservoir sampling algorithm on each relation, and then using incremental view maintenance [22, 29, 32] to maintain the join of the samples.

The second approach is directly supported by our algorithm. Specifically, suppose S_i is a reservoir sample of size k of $\pi_{\mathbf{y}}Q(\mathcal{R}_i)$ maintained by our algorithm. The sample mean is thus

$$\bar{\mu}_i = \frac{1}{k} \sum_{t \in S_i} w(t).$$

We can construct a confidence interval over the true mean μ_i. We first compute the sample variance

$$\bar{\sigma}_i^2 = \frac{1}{k-1} \sum_{t \in S_i} (w(t) - \bar{\mu}_i)^2.$$

Then the $(1-\alpha)$-confidence interval over μ_i is

$$\left(\bar{\mu}_i - z_{\alpha/2} \cdot \frac{\bar{\sigma}_i}{\sqrt{k}}, \bar{\mu}_i + z_{\alpha/2} \cdot \frac{\bar{\sigma}_i}{\sqrt{k}} \right), \tag{4}$$

where $z_{\alpha/2}$ is the z-value corresponding to $\alpha/2$.

The sample mean can be easily maintained in $O(1)$ time for every change in the sample. To maintain the sample variance efficiently, we rewrite it as

$$\bar{\sigma}_i^2 = \frac{1}{k-1} \left(\sum_{t \in S_i} w(t)^2 - 2\bar{\mu}_i \cdot \sum_{t \in S_i} w(t) + k\bar{\mu}_i^2 \right),$$

i.e., it boils down to $\sum_{t \in S_i} w(t)^2$ and $\sum_{t \in S_i} w(t)$, both of which can be maintained in $O(1)$ time per change.

8.2 Selectivity Estimation

Given an input stream of tuples $x_1, x_2, \ldots$, and a predicate θ, the selectivity of θ is the fraction of tuples that satisfy θ (i.e., real tuples). This is a special case of the mean estimation problem, where $w(t) = 1$ for the real tuples and $w(t) = 0$ for the dummy tuples. Thus, we can directly use the solution above. However, that solution requires $O(k)$ space to store the sample explicitly. Below, we show how Algorithm 1 can be modified to use only $O(1)$ space to maintain a good estimate of the number of real tuples in the stream, from which the selectivity of θ can be derived easily.

Recall that Algorithm 1 conceptually assigns a uniformly random number in $[0, 1]$ to each real tuple and then maintains the k tuples with the smallest random numbers. However, it does not generate all these random number explicitly (which would take $O(N)$ time); instead, only the k-th smallest random number is maintained in the variable w. As shown in [10], if we draw n uniformly random numbers in $[0, 1]$ and let w be the k-th smallest, then $\frac{k-1}{w}$ is a $\left(1 \pm \frac{1}{\sqrt{k}}\right)$-estimate of n with constant probability. Thus, the variable w in Algorithm 1 directly provides a good approximation of the number of real tuples in the stream so far. The sample S is thus not required, and the algorithm only needs $O(1)$ space.

8.3 Query Cardinality Estimation

In query cardinality estimation, we wish to estimate $|\pi_{\mathbf{y}} Q(\mathcal{R}_i)|$ continuously for a given SPJ query $\pi_{\mathbf{y}} Q$. This is a more general problem than selectivity estimation, where the query also involves joins and a projection onto $\mathbf{y}$. In fact, when the query has only a projection but no joins, the problem is exactly the *distinct count* problem, which has been studied extensively in the literature [10, 20, 28].

For a general SPJ query $\pi_{\mathbf{y}} Q$, our reservoir sampling algorithm conceptually assigns a uniformly random number in $[0, 1]$ to each tuple in $\pi_{\mathbf{y}} Q(\mathcal{R}_i)$ and maintains the kth smallest random number in a variable w, except that it uses the batch version (Algorithm 4). The same rationale works here, and $\frac{k-1}{w}$ provides a $\left(1 \pm \frac{1}{\sqrt{k}}\right)$-estimate of $|\pi_{\mathbf{y}} Q(\mathcal{R}_i)|$. However, for an SPJ query, we need to use the sampling index in Section 5 in conjunction with Algorithm 4, and the index requires linear space. In fact, linear space is required for any constant-factor approximation of the query cardinality when joins are present, since even checking if a two-way join is empty requires linear space, by a reduction from the communication complexity of set intersection [16].

9 Experiments

9.1 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 [36] (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 joins algorithm [3] was proposed for computing the (delta) join results for the basic two-table join over data streams. In Ref. [36], the 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 [36], 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 a timeout of 12 hours) and report the average running time. Our code repository is available at https://github.com/hkustDB/Reservoir-Sampling-over-Joins.

Datasets and Queries. We evaluate algorithms on graph and relational datasets/queries. All queries in SQL can be found at our code repository.

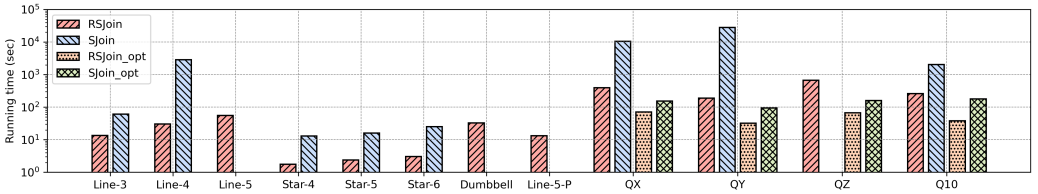


Fig. 16. Running time over different join queries.

We use the Epinions dataset that contains 508,837 edges from SNAP (Stanford Network Analysis Project) [4] 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), line-5-p join (which finds length-3 paths positioned in the middle of a length-5 path), star-k joins (which find all combinations of k edges sharing a common vertex), and dumbbell join (which find all pairs of triangles that are connected by an edge). There is no foreign-key join in graph queries.

We use two relational datasets. One is the TPC-DS dataset [5], which models several generally applicable aspects of a decision support system. We evaluate the same QX, QY, and QZ queries as [36] on TPC-DS, which include the foreign-key joins, and follow the same setup as [36], 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) [1], which focuses on join-heavy complex queries with updates. We tested Q10 query from the Business Intelligence (BI) workload 10. Similar to before, the static tables (such as tag and city) are pre-loaded, and the dynamic tables are loaded in a streaming fashion.

9.2 Experiment Results

Running time. Figure 16 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 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 the 12-hour time limit while SJoin cannot finish on the line-5 join and the QZ join. For the dumbbell join and line-5-p join, the results are missing for SJoin since it does not support cyclic queries or join-project 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 to mention the case when SJoin cannot finish in time. When a foreign-key join exists (i.e., QX, QY, QZ, and Q10), RSJoin_opt improves 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. Finally, for line-k queries, RSJoin demonstrates increasing performance advantages over SJoin as k grows (i.e., the join path gets longer). Specifically, RSJoin achieves a 4.6x speedup on the line-3 query and a 94.6x speedup on the line-4 query. For the line-5 query, SJoin even fails to complete within the time limit.

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 17 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, SJoin only guarantees an update cost of $O(N)$, and its update time ranges from 0.5 µs to 165 ms, with an average of 1.4 ms.

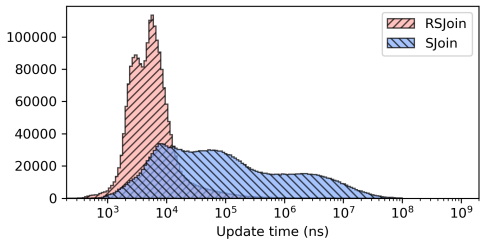


Fig. 17. Update time distribution.

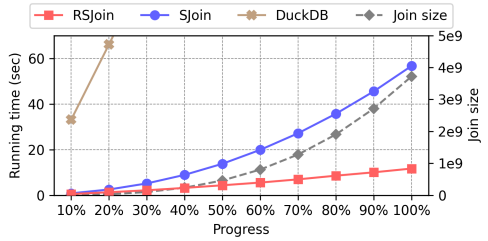


Fig. 18. Running time v.s. input size and join size.

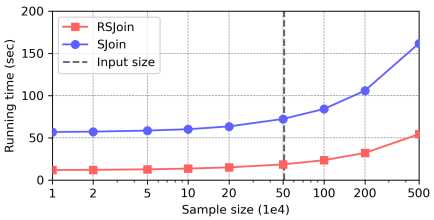


Fig. 19. Running time v.s. sample size.

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

Table 1. Optimizations on QZ Over TPC-DS Dataset

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 the line-3 join. Figure 18 shows the progress of the 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 $O\left(N \log N + k \log N \log \frac{N^{\rho *}}{k}\right)$, where the term $O(N \log N)$ almost dominates the total execution time in this case. In contrast, the total execution time of SJoin shows a clear increasing trend together with the increase in the join size, which is much larger than the input size. We also conduct this experiment in DuckDB [2], a state-of-the-art analytical processing system. After processing every 10% of the input data, we issue a query to recompute a sample from scratch in DuckDB. The results show that DuckDB takes over 60 seconds to process the first 20% of the input data. This suggests that reservoir sampling is more efficient for applications requiring continuously updated samples.

Sample size. We next study how the sample size k affects the total execution time of both algorithms. Figure 19 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 slowly. More specially, when k increases from 10, 000 to 500, 000, the total execution time of RSJoin only increases by a factor of 2. However, when the sample time overrides the input size, i.e., $k > N$, the total execution time of RSJoin starts to increase rapidly. This is also expected again as the theoretical complexity of RSJoin is $O\left(N \log N + k \log N \log \frac{N^{\rho *}}{k}\right)$. Hence, increasing the sample size results in a rapid increase in the total execution time. SJoin follows a similar trend. Moreover, when the sample size reaches $k = 10,000$, the 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 the TPC-DS dataset with scale factors of 1,3,10, and 30. The results are shown in Figure 20. The input size of QZ

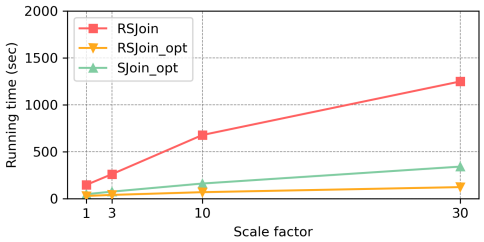


Fig. 20. Running time v.s. scale factor.

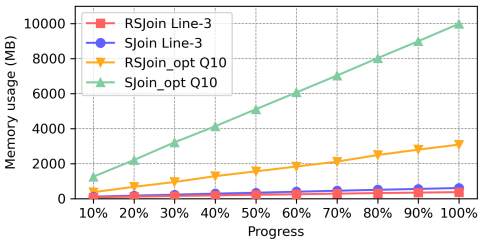


Fig. 21. Memory usage v.s. input size.

is approximately 226MB when the scale factor is 1, while the input size reaches around 6.6GB when the scale factor reaches 30. We do not include the results of SJoin here since it takes more than 4 hours to finish the execution even with a scale factor of 1. We observe that even without applying foreign-key optimization, RSJoin achieves linear growth in the 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 21 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 the 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 21. The memory usage of Q10 grows much faster than the line-3 join, as it is much more complex with a 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 12–14 in Algorithm 6. Table 1 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 propagations 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.

9.3 Applications

We evaluate the accuracy of our algorithm for two of the applications introduced in Section 8.

Query Cardinality Estimation. We evaluate our algorithm on the Epinions dataset using line-3 and line-5-p joins, estimating their query output sizes. The true join size is 3,721,042,797 for the line-3 join and 3,497,166,520 for the line-5-p join. We set $\epsilon = 0.05$ (allowing a relative error below 5%) and $\delta = 0.01$, resulting in $k = \frac{20}{0.05^2} = 8000$. Each trial repeats $\lceil 4 \log \frac{1}{0.01} \rceil = 27$ times, reporting the median as the final estimate. We conduct 100 trials per query and compute the relative error for each, shown in Figure 22 (line-3 join) and Figure 23 (line-5-p join). The errors are tightly concentrated around 0. Moreover, across all 200 trials, the relative error never exceeds the ± 0.05 bound. These results confirm that our algorithm consistently provides a $(1 \pm \epsilon)$-approximation of the true join size with high probability.

Mean Estimation. We use a line-3 join on the Epinions dataset, assigning each edge a random weight between 1 and 10,000. For each length-3 path $e_1 - e_2 - e_3$, the weight is computed as

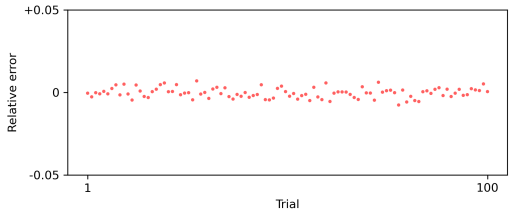


Fig. 22. Relative error in Line-3 join size estimation.

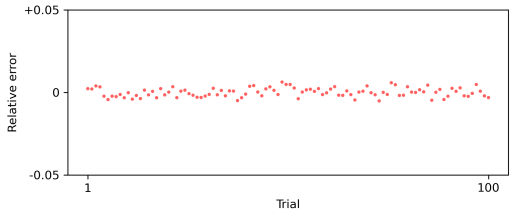


Fig. 23. Relative error in Line-5-P join size estimation.

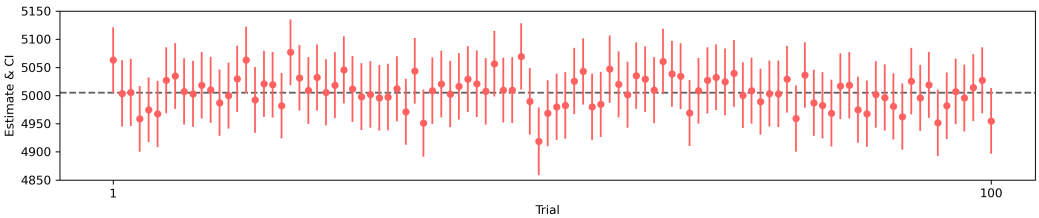


Fig. 24. Mean estimation for Line-3 join based on RSJoin.

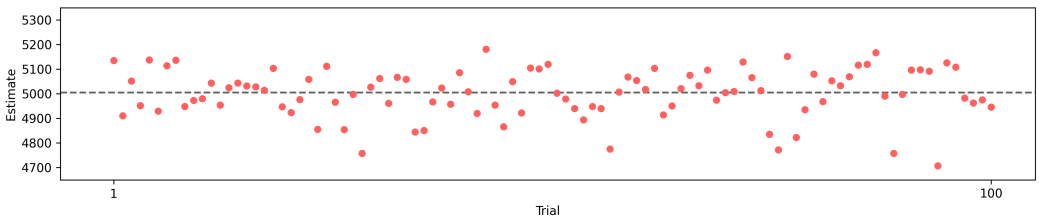


Fig. 25. Mean estimation for Line-3 join based on ripple join.

$0.7 \times w_1 + 0.2 \times w_2 + 0.1 \times w_3$, where w_1, w_2, and w_3 are the weights of edges e_1, e_2, and e_3, respectively. The average weight of all length-3 paths is 5004.95. We set $k = 5000$ and run 100 trials, each time reporting the estimated mean and its 95% confidence interval. Figure 24 shows the results, with a horizontal dashed line indicating the true mean. In 97 out of 100 trials, the confidence interval contains the true mean, and the average relative error is only $\pm 0.45\%$. It demonstrates that our algorithm accurately estimates the mean and provides a valid $(1 - \alpha) \cdot 100\%$ confidence interval.

We also evaluate the ripple join approach [21] for mean estimation. We maintain a sample of size 5,000 for each input relation using the classical reservoir sampling algorithm. Then, we use the join results of these sampled relations to estimate the mean. We ran this experiment 100 times, and the results are shown in Figure 25. We can see that, although the estimated mean still centers around the true value, the results are more dispersed compared with those of RSJoin. The average relative error is $\pm 1.6\%$, which is larger than RSJoin. This indicates that, given the same sample size, mean estimation based on RSJoin provides more accurate results.

9.4 Reservoir Sampling with Predicate

At last, we compare our new reservoir sampling algorithm with a 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 1,024 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

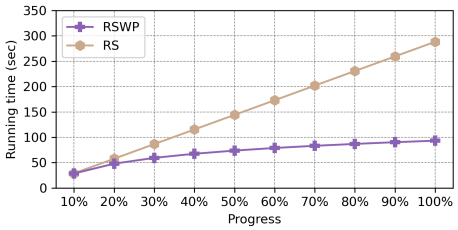


Fig. 26. Running time v.s. input size.

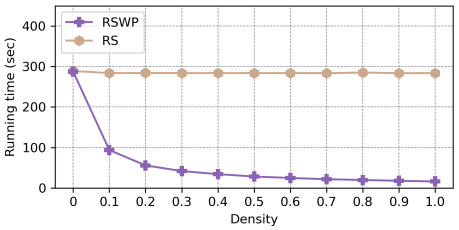


Fig. 27. Running time v.s. density.

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 26, 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 ith item.

In Figure 27, we measure the running time of both RSWP and RS over 11 streams of the 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 the input stream. In contrast, the running time of RSWP depends on the density of the 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.

10 Additional 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 space usage is important). For example, Paven et al. [30] 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 a sub-linear number of query accesses to the graph is important), Eden et al. [19] studied the problem of almost uniform sampling of edges, and Biswas et al. [12] studied the problem of sampling subgraphs.

Maintaining Conjunctive Queries under Updates. It has been shown [11, 22] 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, [22] showed an index for acyclic joins that can be updated in $O(N)$ time. Later, Kara et al. [23] 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, and so on. Moreover, Kara et al. [24] also investigated the tradeoff between update time and delay for hierarchical queries. Wang et al. [32, 33] worked on instance-dependent complexity by relating the update time to the enclosureness of update sequences.

11 Conclusion

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

References

[1] 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. In *Proceedings of the VLDB Endow* 16, 4 (December 2022), 877–890.

[2] Junyi Xie and Jun Yang. 2007. A survey of join processing in data streams. In *Data Streams: Models and Algorithms*, Charu C. Aggarwal (Ed.). Springer US, Boston, MA, 209–236.

[3] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection. Retrieved from http://snap.stanford.edu/data

[4] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. Association for Computing Machinery, Madison, Wisconsin, 582–587.

[5] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, Amsterdam, Netherlands, 1981–1984.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[7] 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* (Philadelphia, Pennsylvania, USA) *(SIGMOD'99)*. Association for Computing Machinery, New York, NY, USA, 275–286.

[8] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size bounds and query plans for relational joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS'08)*. IEEE Computer Society, USA, 739–748.

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.

[10] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science*, José D. P. Rolim and Salil Vadhan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–10.

[11] 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* (Chicago, Illinois, USA) *(PODS'17)*. Association for Computing Machinery, New York, NY, USA, 303–318.

[12] Amartya Shankha Biswas, Talya Eden, and Ronitt Rubinfeld. 2021. Towards a decomposition-optimal algorithm for counting and sampling arbitrary motifs in sublinear time. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 207)*, Mary Wootters and Laura Sanità (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 55:1–55:19.

[13] 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* (Portland, OR, USA) *(PODS'20)*. Association for Computing Machinery, New York, NY, USA, 393–409.

[14] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD'99)*. Association for Computing Machinery, New York, NY, USA, 263–274.

[15] Yu Chen and Ke Yi. 2020. Random sampling and size estimation over cyclic joins. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 155)*, Carsten Lutz and Jean Christoph Jung (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:18.

[16] Graham Cormode and Ke Yi. 2020. *Small Summaries for Big Data*. Cambridge University Press.

[17] Binyang Dai, Xiao Hu, and Ke Yi. 2024. Reservoir sampling over joins. *Proc. ACM Manag. Data* 2, 3, Article 118 (May 2024), 26 pages.
[18] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. On join sampling and the hardness of combinatorial output-sensitive join algorithms. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Seattle, WA, USA) *(PODS'23)*. Association for Computing Machinery, New York, NY, USA, 99–111.
[19] Talya Eden and Will Rosenbaum. 2018. On sampling edges almost uniformly. In *1st Symposium on Simplicity in Algorithms (SOSA 2018) (Open Access Series in Informatics (OASIcs), Vol. 61)*, Raimund Seidel (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:9.
[20] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), Article 10 (Jan 2007).
[21] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD'99)*. Association for Computing Machinery, New York, NY, USA, 287–298.
[22] Muhammad Idris, Martin 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* (Chicago, Illinois, USA) *(SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 1259–1274.
[23] 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 (ICDT 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 127)*, Pablo Barcelo and Marco Calautti (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:18.
[24] 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* (Portland, OR, USA) *(PODS'20)*. Association for Computing Machinery, New York, NY, USA, 375–392.
[25] Kyoungmin Kim, Jaehyun Ha, George Fletcher, and Wook-Shin Han. 2023. Guaranteeing the $\tilde{O}$(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* (Seattle, WA, USA) *(PODS'23)*. Association for Computing Machinery, New York, NY, USA, 113–125.
[26] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA.
[27] Kim-Hung Li. 1994. Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n))). *ACM Trans. Math. Softw.* 20, 4 (Dec. 1994), 481–493.
[28] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842.
[29] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD'18)*. Association for Computing Machinery, New York, NY, USA, 365–380.
[30] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1870–1881.
[31] Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57.
[32] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change propagation without joins. *Proc. VLDB Endow.* 16, 5 (Jan. 2023), 1046–1058.
[33] 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* (Portland, OR, USA) *(SIGMOD'20)*. Association for Computing Machinery, New York, NY, USA, 1225–1239.
[34] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB'81)*. VLDB Endowment, 82–94.
[35] 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* (Houston, TX, USA) *(SIGMOD'18)*. Association for Computing Machinery, New York, NY, USA, 1525–1539.
[36] 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* (Portland, OR, USA) *(SIGMOD'20)*. Association for Computing Machinery, New York, NY, USA, 2027–2042.

Received 23 April 2025; revised 3 September 2025; accepted 16 December 2025