Paper Presentation:
Reservoir Sampling over Joins

Yuxin Kang

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS

Background

(1) Fully materializing the join can be extremely expensive in both time and space. Uniform sample of the join results would suffice for many purposes, (eg. answer analytical queries, train ML models.

(2) Real-world data arrives continuously, so the database is always growing. We need a method that keeps a uniform random sample of the join results as new tuples stream in, rather than sampling after everything is collected.

Previous Results

N: the total size of all input relations

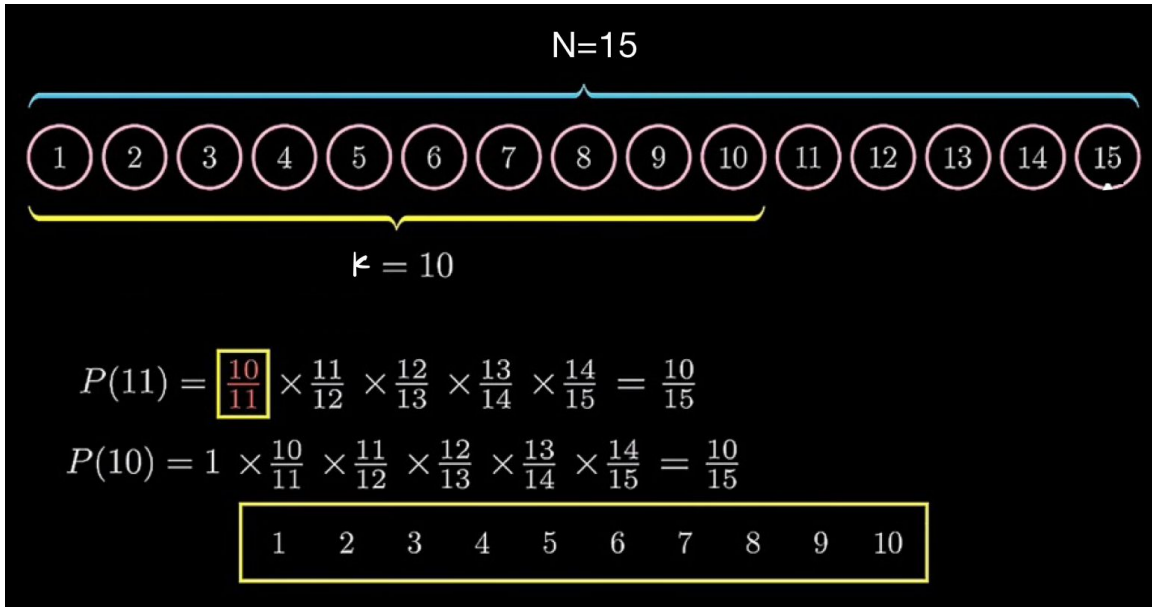
|Q(R)|: the number of tuples in the join result.

ρ*: the fractional edge cover number

(1) Static Join Sampling:
Acyclic joins : Build an index in O(N), draw one sample in O(1).

Cyclic joins: Build an index in O(N), draw one sample in $O\left(\frac{N^{\rho*}}{|Q(\mathcal{R})|}\right)$

(2) Reservoir Sampling (Classical):
The total number of elements in the stream is unknown; Each element can only be accessed once. We want each element has an equal probability of being sampled.



For item number t: If t ≤ k: store it directly in the sample.

If t > k:Draw i uniformly at random from {0, 1, ..., t-1}.

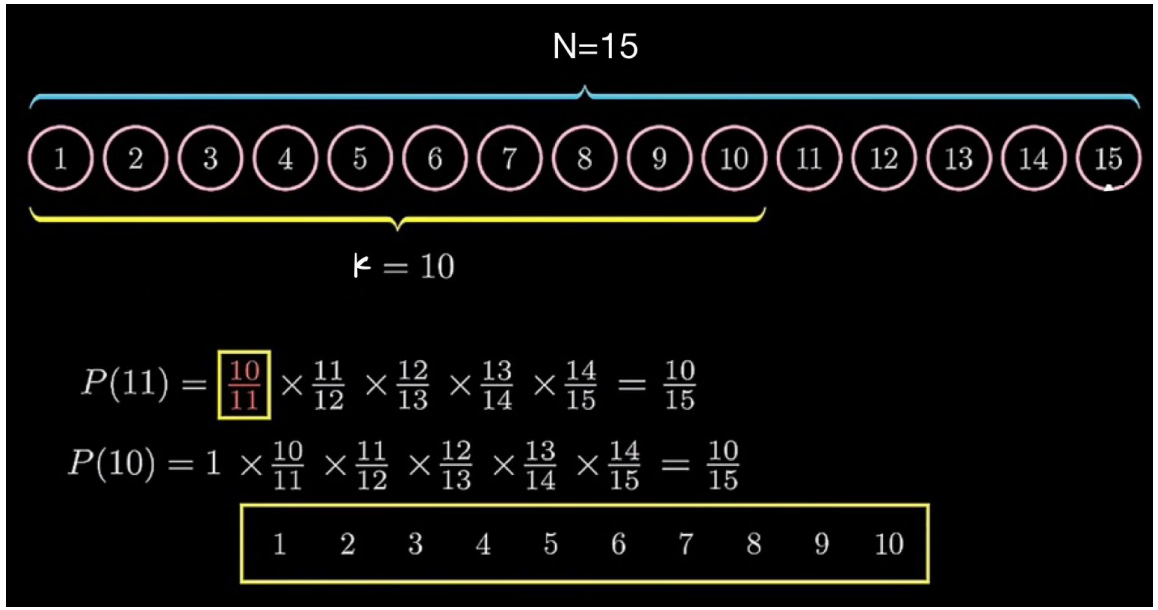
If i < k, replace sample[i] with the new item.

Otherwise, do nothing.

Each item has the probability:

$$\frac{k}{k+1} \times \frac{k+1}{k+2} \times \frac{k+2}{k+3} \times \cdots \times \frac{N-2}{N-1} \times \frac{N-1}{N} = \frac{k}{N}$$

Reservoir Sampling (Classical) , cont.



Without skip optimization: O(N)

With skip optimization: $O\left(k \log \frac{N}{k}\right)$

The skip optimization:

Suppose we have N > 100,000 and k = 10. The probability that the t = 100,000 th item will be accepted is: 0.0001.

99.99% of items will be rejected. So, checking each one is a waste of time.

The probability of keeping the t-th item in the reservoir is $p_t = \frac{k}{t}$.

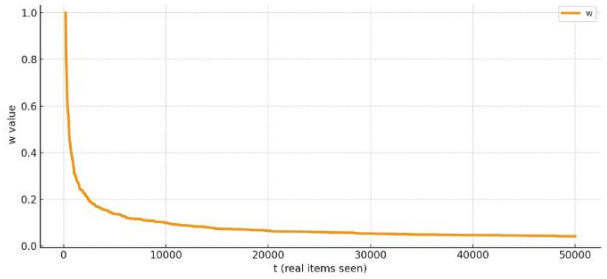
So, $\Pr(\text{skip exactly } q \text{ items}) = (1 - p_t)^q \cdot p_t$ —-> q ~Geom (pt)

The skip-based algorithm does not examine every item, so we can not compute t.

So the algorithm maintains a value of w such that

$$w \leftarrow w \cdot (u')^{1/k} \quad \text{where } u' \sim \text{Unif}(0, 1)$$

And w behave like $w \approx \frac{k}{t}$



Reservoir Sampling (Classical) — the skip optimization, cont.

Algorithm:

Initialize reservoir S with the first k items.
Initialize w after filling the reservoir.
Repeat for the rest of the stream:

Draw $q \sim \text{Geom}(w)$.
Skip the next q items in the stream.
Let x be the next item.

Draw $u \sim \text{Unif}(0,1)$.
If $u < w$, then:
replace a random item in S with x;
draw $u' \sim \text{Unif}(0,1)$ and update $w \leftarrow w \cdot (u')^{1/k}$.

drawing u from a uniform distribution and comparing it to w gives: Pr[u<w] = w.

The expectation of the acceptance of the t-th item is:

$$0 \cdot \frac{t-k}{t} + 1 \cdot \frac{k}{t} = \frac{k}{t}$$

The expected number of acceptances over the entire stream:

$$\begin{aligned} \sum_{t=k+1}^{N} \frac{k}{t} = k \sum_{t=k+1}^{N} \frac{1}{t} &= k\,(H_N - H_k) \\ &\approx k\,(\log N - \log k) \\ &= k \log\left(\frac{N}{k}\right). \end{aligned}$$

Each skip and replacement both take O(1) time. So the total expected time is :

$$O\left(k \log \frac{N}{k}\right)$$

Previous Results

(3) Attempts of Applying Reservoir Sampling to Joins: Zhao et al. investigated the problem over acyclic joins. But their solution takes $O(N^2)$ time in the worst case since their index needs costly maintenance when new tuples arrive. This is essentially no better than a naive approach that re-build the static join sampling index and re-draw the samples after each tuple has arrived ($N * O(N) = O(N^2)$).

Motivation

Can we build a new reservoir sampling algorithm that maintains a sample over joins with a near-linear running time?

Reservoir Sampling with Predicate

Goal: We want to maintain a reservoir sample of size k only over items that satisfy a predicate θ. (Items that satisfy θ are called real items. Items that do not satisfy θ are called dummy items.)

Key changes: The skip counts real items only. Dummy items do not count.

Repeat for the rest of the stream:

Draw a skip length $q \sim \text{Geom}(w)$.
Skip the next q items in the stream.
Let x be the next item.
If $\theta(x)$ is false, then continue to the next iteration.
Draw $u \sim \text{Unif}(0, 1)$.
If $u < w$, then:
Choose a random position in S and replace its item with x.
Draw $u' \sim \text{Unif}(0, 1)$ and update $w \leftarrow w \cdot (u')^{1/k}$.

Reservoir Sampling with Predicate

Why It Remains Correct:

We never adjust w when seeing dummy items, So,

$$w \approx \frac{t}{k} \Rightarrow w \approx \frac{t}{r}$$

where r is the counts of only the real items.

Each real item is accepted with probability $\frac{k}{r}$, ensuring every real item is equally likely to appear in the reservoir.

Upper bound: Reservoir Sampling with Predicate

Let: r_i= number of real items among the first (i−1) items

- If x_i is not dummy, then: It cannot enter the reservoir; It does not update w. So, the cost is 0.
- If x_i is real, it could enter the reservoir, The probability that a real item enters the sample at this moment is:

$$\Pr(\text{real } x_i \text{ enters}) = \frac{k}{r_{i+1}}$$

So, expected work at step i is:

$$\text{Cost}_i = \min\left(1, \frac{k}{r_{i+1}}\right)$$

(If we haven't filled the reservoir yet, cost is 1; once full, it becomes $\frac{k}{r_{i+1}}$)

Sum over all items:

$$\text{Total Time} = O\left(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i + 1}\right)\right)$$

If every item is real, then $r_i = i - 1$:

$$\sum_{i=k}^{N} \frac{k}{i} = O\left(k \log \frac{N}{k}\right)$$

Lower bound: Reservoir Sampling with Predicate

To output a uniform sample of size k over only the real items, a real item at position i must be given probability $\frac{k}{r_{i+1}}$ to enter the reservoir.

Therefore, any correct algorithm must stop and examine position i with probability at least $\min\left(1, \frac{k}{r_i + 1}\right)$

If it stops less often than this, it would miss some real items that should have been considered. Those items would have too small a chance of entering the reservoir. The resulting sample would not be uniform anymore.

$$\Omega\left(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i + 1}\right)\right)$$

Sampling over Acyclic Joins

Goal: We want to build a data structure that allows us to:

a. Incrementally update the join when new tuples arrive
b. Efficiently retrieve or skip join results later.

Example:

R_1

X	Y
a	2
b	3

R_2

Y	Z
2	P
2	Q
3	R

(1) Two-table join $R_1(X, Y) \bowtie R_2(Y, Z)$

y value	List $R_1 \bowtie y$	List $R_2 \bowtie y$
2	$\{(a, 2)\}$	$\{(2, P), (2, Q)\}$
3	$\{(b, 3)\}$	$\{(3, R)\}$

a.We maintain one bucket per value of Y

b. Each tuple is stored exactly once in its bucket. So, it uses linear space: $\sum_b (|R_1 \bowtie b| + |R_2 \bowtie b|) = O(N)$

Sampling over Acyclic Joins

(1) Two-table join —cont

c. If a new tuple t arrives:

- If t $\in$ R_1, insert it into bucket $R_1 \ltimes$ t.Y
- If t $\in$ R_2, insert it into bucket $R_2 \ltimes$ t.Y

This is just an append → O(1) time.

d. If t $\in$ R1, then new join results are:

$\Delta J = R2 \ltimes t.Y$

because the lists are directly stored, accessing any joined result is O(1)

Example:

y value	List $R_1 \ltimes y$	List $R_2 \ltimes y$
2	$\{(a, 2)\}$	$\{(2, P), (2, Q)\}$
3	$\{(b, 3)\}$	$\{(3, R)\}$

$t = (c, 2) \in R_1$

$$\boxed{\Delta J = \{(c, 2, P),\ (c, 2, Q)\}}$$

Sampling over Acyclic Joins

$$R_1(X,Y) \bowtie R_2(Y,Z) \bowtie R_3(Z,W)$$

X	Y
2	5
4	4
5	5
5	1
5	6
6	3
6	6
7	4
7	5
7	6

Y	Z
1	1
1	2
1	4
2	3
6	5
7	7

Z	W
1	1
2	3
2	4
3	1
3	5
3	6
3	7
4	4
4	5
6	6

(2) Line-3 Join

a. Index Rebuild: $O(\log N)$

Initialization:

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

For each tuple $(b,c) \in R_2$, we compute:

$\widetilde{cnt}(c) = 2^{\lceil \log_2 cnt(c) \rceil}$; $\widetilde{cnt}(c) = 2^i$

where $cnt(c) = |R_3 \ltimes c|$ is the number of match in R_3

For $b=1$:

$(1,1)$: $c=1$, $\widetilde{cnt}(1) = 1 = 2^0 \Rightarrow$ bucket $i=0$

$(1,2)$: $c=2$, $\widetilde{cnt}(2) = 2 = 2^1 \Rightarrow$ bucket $i=1$

$(1,4)$: $c=4$, $\widetilde{cnt}(4) = 2 = 2^1 \Rightarrow$ bucket $i=1$

For $b=2$:

$(2,3)$: $c=3$, $\widetilde{cnt}(3) = 4 = 2^2 \Rightarrow$ bucket $i=2$

b: A value from attribute Y in $R_2(Y,Z)$

c: A value from attribute Z in $R_2(Z,W)$

Sampling over Acyclic Joins

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

$\mathcal{L}_b$: the list of non-empty buckets for a given $b \in \pi_X(R_2)$

$\psi_i(b)$: i-th sub-bucket inside bucket b: $\psi_i(b) = 2^i \cdot |\mathcal{L}_{b,i}|$

N_b: Total number of items in bucket b

$\psi_0(1) = 2^0 \cdot 1 = 1$

$\psi_1(1) = 2^1 \cdot 2 = 4$

$\psi_2(2) = 2^2 \cdot 1 = 4$

$N_1 = 1+4 = 5$

$N_2 = 4$

Inserting (2,2) into R_2:

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

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

(2,2): $c=2$, $\widetilde{cnt}(2) = 2 = 2^0 \Rightarrow$ bucket $i=1$

(doesn't affect $cnt(c)$)

$\psi_1(2) = 2^1 \cdot 1 = 2$

$N_2 = 2+4 = 6$

Sampling over Acyclic Joins

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

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

(1,2) : $c=2$, $\widetilde{cnt}(2) = 4 = 2^2$ ($cnt(2)=3$)
$\Rightarrow$ bucket $i=2$

(2,2): $c=2$, $\widetilde{cnt}(2) = 4 = 2^2$
$\Rightarrow$ bucket $i=2$

$\psi_1(1) = 2^1 \cdot 1 = 2$
$\psi_2(1) = 2^2 \cdot 1 = 4$
$N_2 = 1+2+4 = 7$

$\psi_2(2) = 2^2 \cdot 2 = 8$
$N_2 = 8$

Every time we insert a new tuple into R_3
$\Downarrow$
The count of some c increase.
$\Downarrow$
(b, c) pairs in the index may move to a new bucket

The largest cnt is at most N. So, the number of buckets is at most $\log_2 N + 1$.

Therefore, each (b, c) moves across at most $\log N$ buckets in total.

$\therefore$ total index rebuild $O(\log N)$ amortized time

Sampling over Acyclic Joins

b. Get a sample:

Step 1: Sample (a,b) from R1 (but not uniformly)

We prefer (a, b) pairs that lead to more join results. → $O(\log N)$

Step 2 : Sample (b, c) from $R_2 \ltimes b$ → $O(\log N)$

Binary search over cumulative weights

Step 3: Sample (c,d) uniformly from $R_3 \ltimes c$ → $O(1)$

Total: $O(\log N)$

Reservoir Sampling over Joins

For acyclic:

(1) Update the join index

Cost per tuple: $O(\log N)$

Total over N tuples: $O(N \log N)$

(2)Possibly update the reservoir sample

Retrieve a random join result via the index: $O(\log N)$

Number of times we perform a reservoir replacement: $k \log \frac{N}{k}$

Total reservoir sampling cost: $O\left(k \log N \log \frac{N}{k}\right)$

$$\boxed{O(N \log N)} + \boxed{O(k \log N \log(N/k))} = \boxed{O(N \log N + k \log N \log(N/k))}$$

Reservoir Sampling over Joins

For cyclic :

Cyclic joins can create many more join results, much larger than N

By the Generalized Hypertree Decomposition:

Fractional Hypertree Width $w(Q)$ tells us how cyclic the query is,

If w = 1, the join is acyclic.

If w > 1, the join is cyclic.

The worst -case join size is $O(N^{w(Q)})$

$$O(N^{w(Q)} \log N + k \log N \log(N/k))$$

So the index maintenance time become: $O(N^{w(Q)} \log N)$

And the reservoir sampling part does not change.

Experimental Results

Dataset:
Graph dataset: Epinions;
Relational dataset: TPC-DS for decision-support joins ;
Relational dataset: LDBC-SNB for join-heavy complex queries.

Join queries tested:
Multiple acyclic joins (e.g., line-3, line-4, line-5, star joins).
Some cyclic joins to evaluate the extension that uses fractional hypertree width.

Baseline:
Zhao et al.

Experimental Results

Key Results:

(1) The proposed (RSJoin) update time is stable and stays around 10 microseconds, while the method from Zhao et al. (SJoin) update time is unpredictable and can jump to hundreds of milliseconds, which makes it unreliable in streaming environments.

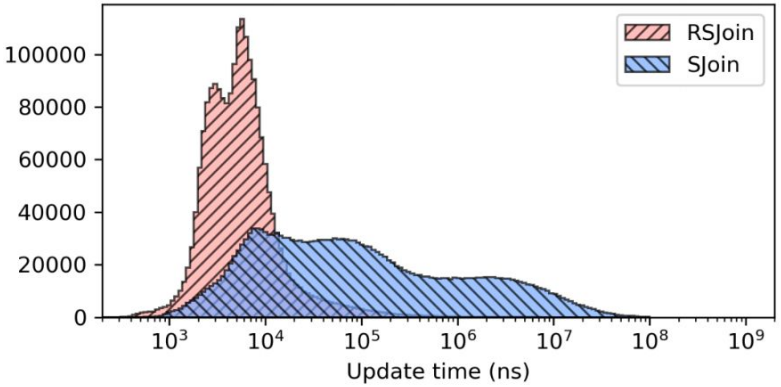


Fig. 6. Update time distribution

(2) Although the number of join results grows rapidly as more data is processed, RSJoin's runtime increases almost linearly with the input, while SJoin's runtime grows roughly in proportion to the join size.

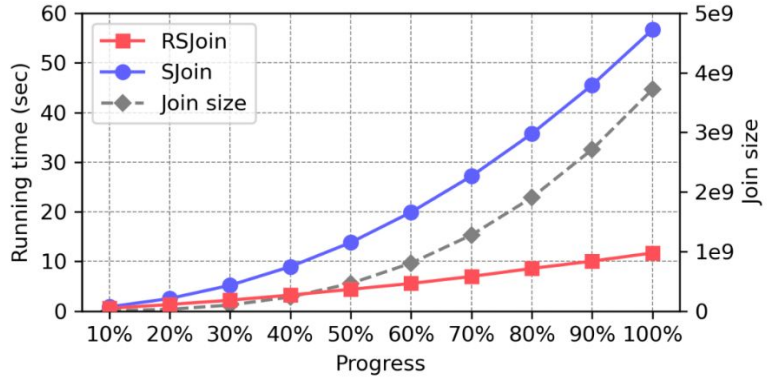


Fig. 7. Running time v.s. input size and join size

Experimental Results

Key Results:

(3) RSJoin achieves 4.6×–147.6× speedups over prior methods and successfully completes all join queries, including cyclic ones where SJoin may fail. It also does not depend on foreign-key constraints, making it more scalable and robust across both graph and relational workloads.

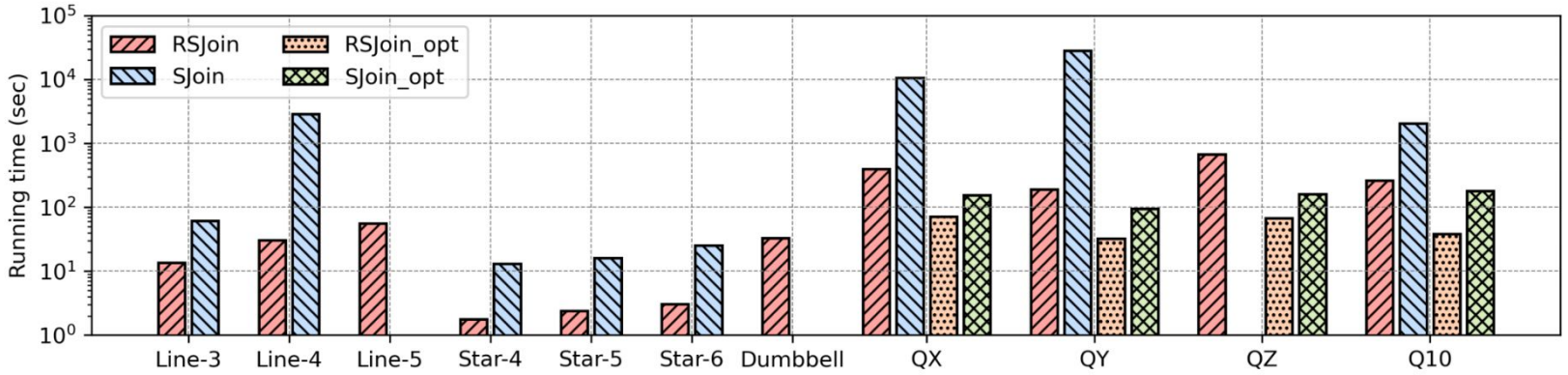


Fig. 5. Running time over different join queries

Experimental Results

Key Results:

(4) RSJoin uses much less memory than SJoin as input grows. For complex relational joins (e.g., Q10), SJoin's memory usage increases to tens of GB, while RSJoin remains low and stable.

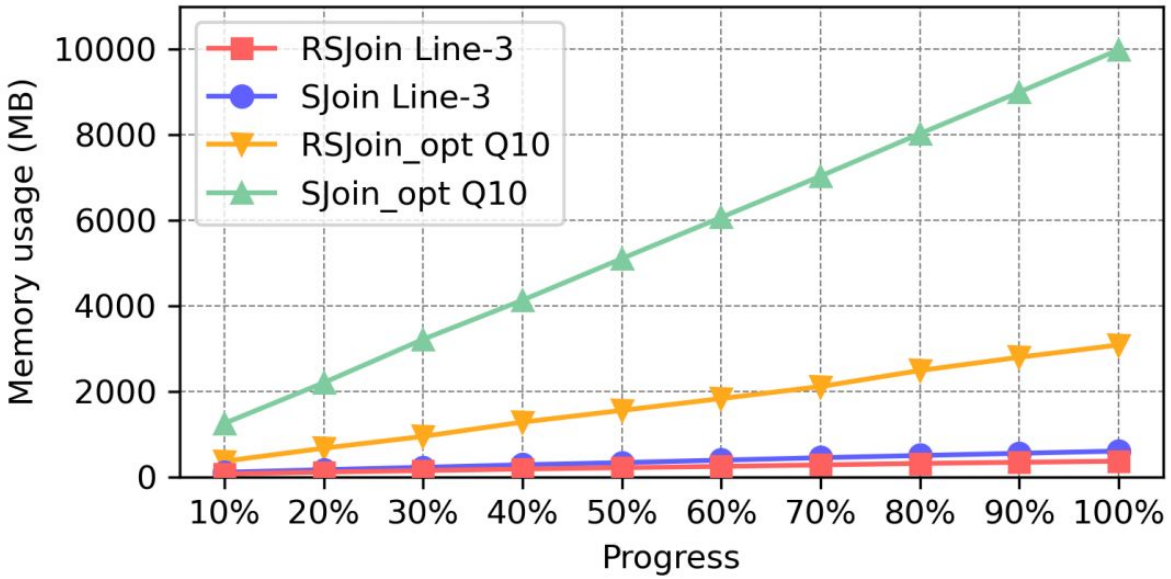


Fig. 11. Memory usage v.s. input size

Conclusion

(1) Introduce reservoir sampling with a predicate, and prove an instance-optimal running time of $O\left(\sum_{i=1}^{N} \min\left(1, \frac{k}{r_i+1}\right)\right)$

(2) Present a dynamic index for acyclic joins supporting O(log N) amortized updates and O(log N) sampling.

(3) Combine both to solve reservoir sampling over joins in $O(N \log N + k \log N \log \frac{N}{k})$.

(4) Extend to cyclic joins with fractional hypertree width , achieving $O(N^{w} \log N + k \log N \log \frac{N}{k})$

References

https://arxiv.org/abs/2404.03194

http://xhslink.com/o/47rnsirYAcF