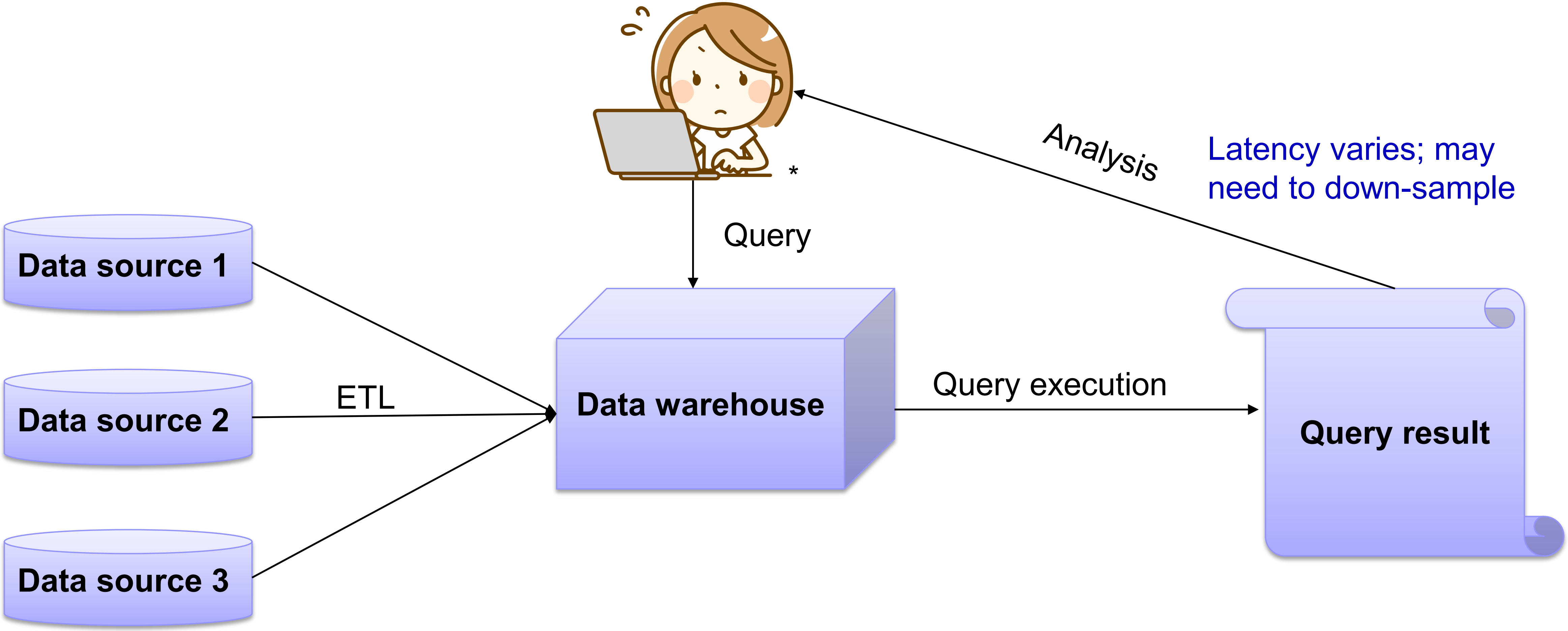
Efficient Join Synopsis Maintenance for Data Warehouse

Zhuoyue Zhao **Feifei Li** **Yuxi Liu**

University of Utah

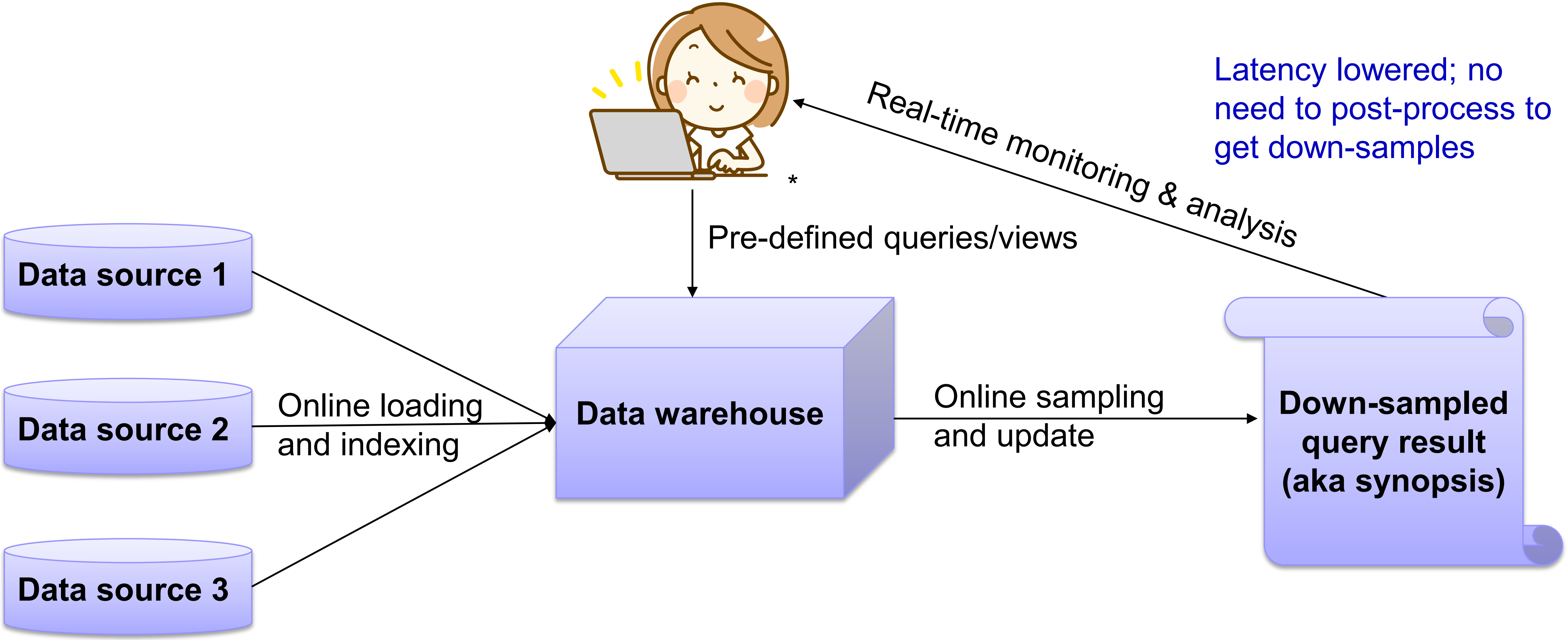
High latency in data analysis pipelines



Typical latency: hours ~ days

Tens of seconds for single-table queries
Hours or longer for many-to-many join queries

Alternatives to cut down latency



Typical latency: milliseconds ~ seconds   milliseconds ~ seconds

Background and challenges

- Existing systems work well for single-table/key-join queries
  - E.g., Apache Storm, Apache Flink, …
  - Sampling is easy to implement on the fly

- Difficulties with multi-table join queries, especially many-to-many joins
  - Even streaming join can be expensive (when join size is large)
  - Limited sampling/indexing support in existing systems
  - Existing *random* sampling algorithms for joins
    - have restrictions on the types of join/aggregations (e.g., [1, 2]);
    - depends on assumptions on data distribution (e.g., [3]);
    - or require offline scans (e.g., [4]).

[1] Tao et al. Random Sampling for Continuous Streams with Arbitrary Updates. In TKDE '06.
[2] Kandula et al. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In SIGMOD '16.
[3] Srivastava et al. Memory-limited Execution of Windowed Stream Joins. In VLDB '04.
[4] Zhao et al. Random Sampling over Joins Revisited. In SIGMOD '18.

Problem Formulation

Given a pre-specified SPJ query in the following form,

```
SELECT *
FROM R1, R2, …, Rn
WHERE <join-preds>
  AND <filter-preds>;
```

where a `<join-pred>` is in the form of,

- `Ri.A op Rj.B`
- `|Ri.A – Rj.B| < d`

(op is one of <, <=, =, >, >=; d is a constant)

maintain a readily available join synopsis (random sample) in a database with any insertions or deletions of tuples, for a user-specified synopsis type (fixed-size w/ replacement, fixed-size w/o replacement or Bernoulli).

- Baseline: SJ (Symmetric index/hash Join)
  - builds conventional tree or hash indexes on all the join columns
    - storage cost is $O(nN)$, where N is the size of the largest table.
  - incrementally maintains samples over a scan of the *full* join results upon insertion
    - insertion cost is at least linear to the join size (costly!)
  - rescans join upon deletion to replenish missing samples upon deletion (very costly!)

Overview of SJoin

- Our solution: SJoin (Synopsis Join)
  - features a specialized per-query index based on *a weighted join graph*, which
    - consists of aggregate indexes on all the join columns
    - provides random access and random sampling to join results
  - runs reservoir sampling style algorithms for the specified synopsis type, which
    - only retrieves the selected join results upon insertion or
    - replenishes missing samples using the weighted join graph index upon deletion
  - has a similar storage cost to SJ
    - $O(nN)$ in theory, and within $\pm 25\%$ in experiments
  - has asymptotically lower insertion cost in many-to-many joins
    - $O(2^n d)$ for a chain band-join with a half-width d, compared to $O(2^n d^n)$ in SJ
  - does not rescan join results upon deletion for missing samples

A running example

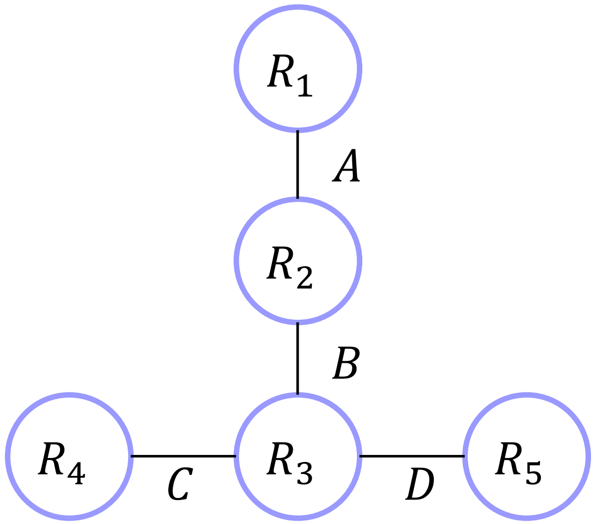
- Suppose we have a pre-specified SPJ query where there are $n = 5$ tables.

Query:

```
SELECT *
FROM R1, R2, R3, R4, R5
WHERE R1.A = R2.A
  AND R2.B = R3.B
  AND R3.C = R4.C
  AND R3.D = R5.D;
```

Synopsis type:

```
Fixed size synopsis of size 4 w/o replacement
```



Weighted join graph

R_1

Row ID	A
0	1
1	2
2	2
3	1

R_2

Row ID	A	B
0	1	1
1	1	2
2	2	2

R_3

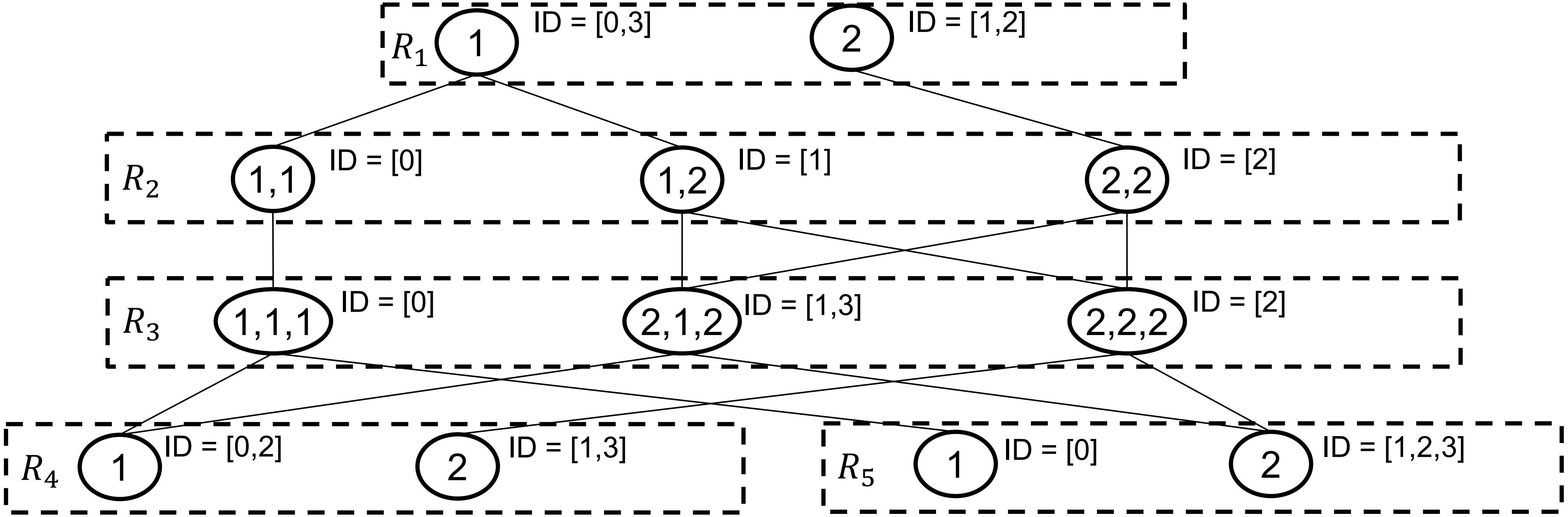
Row ID	B	C	D
0	1	1	1
1	2	1	2
2	2	2	2
3	2	1	2

R_4

Row ID	C
0	1
1	2
2	1
3	2

R_5

Row ID	D
0	1
1	2
2	2
3	2

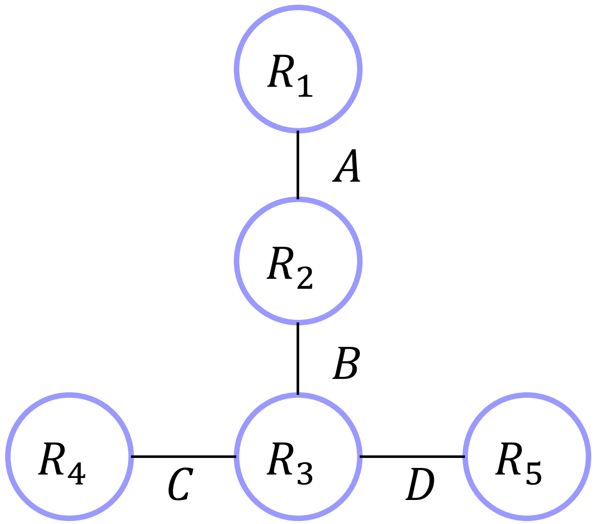


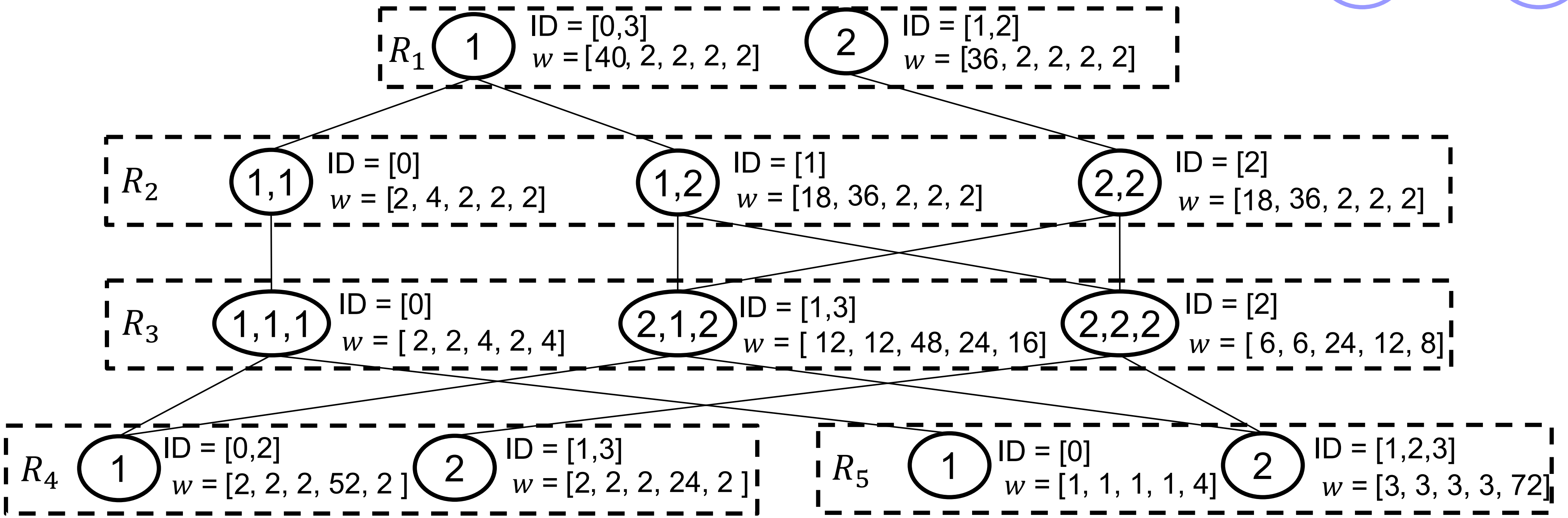
Weighted join graph

$$w_i(t_j) = \left| \bowtie (\mathbb{R}(j) \backslash \mathrm{R}_j) \bowtie \{t_j\} \right|, \qquad w_i(v_j) = \sum_{t_j \in \mathbb{T}(v_j)} w(t_j)$$

where $\mathbb{R}(j)$ is the set of tables in the subtree at R_j and $\mathbb{T}(v_j)$ is the set of tuples that v_j represent.

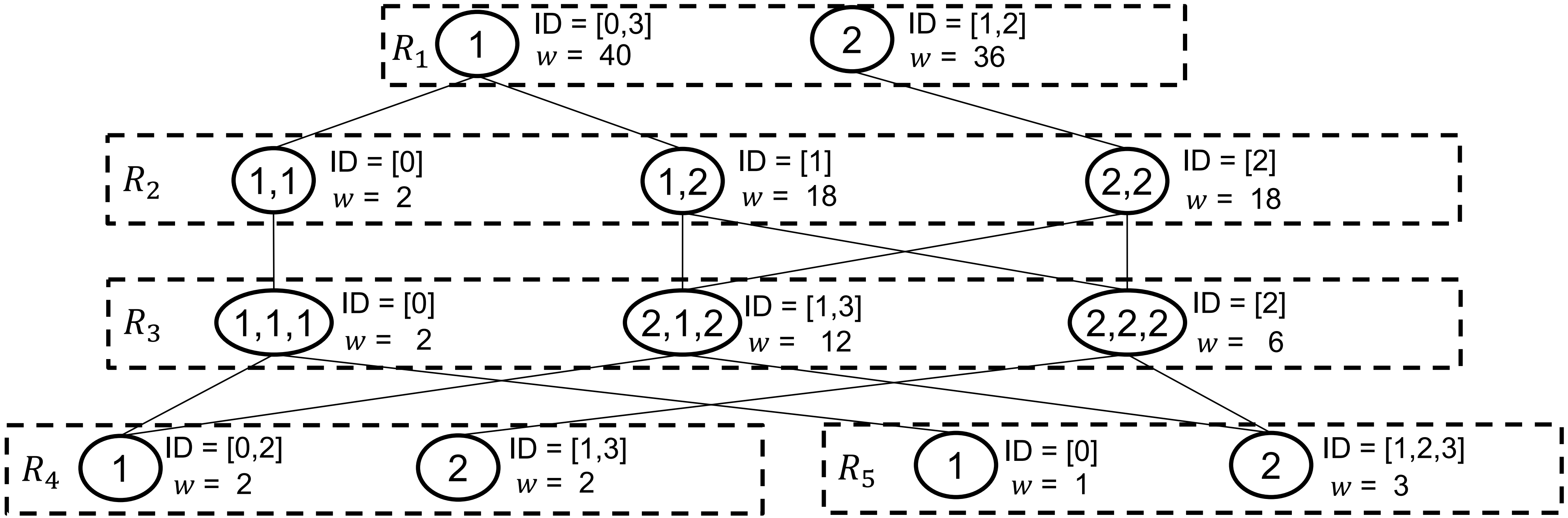
e.g., $w_1(t_1) = |\{t_1\} \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5|$





Drawing a single random join sample

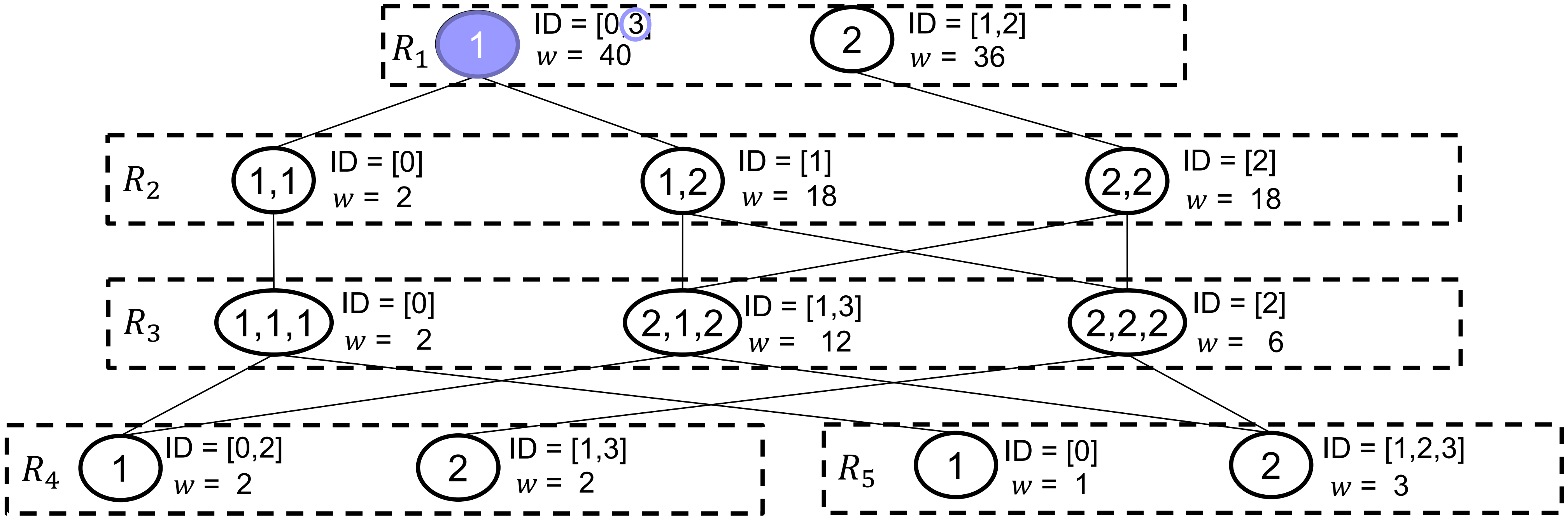
- How to draw random sample from a join *with just one random number*?
  - Fix a join order by choosing any relation R_i as the query tree root
    - Let's say we choose R_1
    - For simplicity, omit the subscript i in the weight functions for now
    - Sort the tuples in R_j based on its join attribute with its parent
      - R_1 is arbitrarily ordered, but we order it by its 1st attribute anyway



Drawing a single random join sample

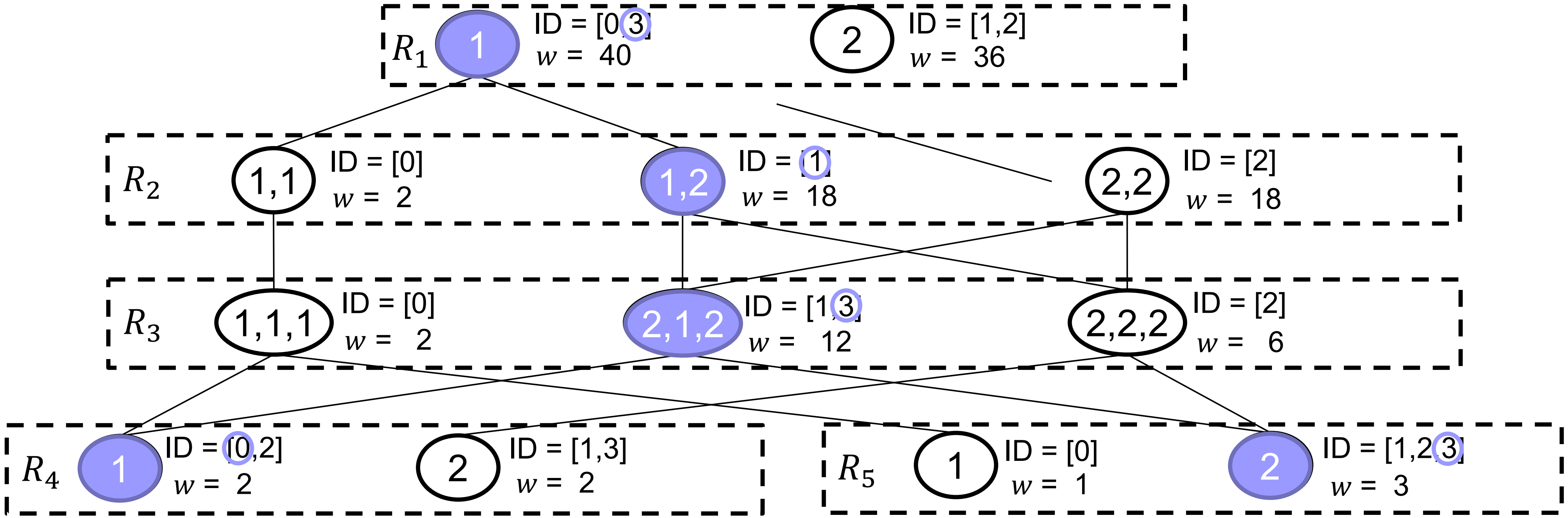
- How to draw random sample from a join *with just one random number*?
  - Generate a random number $l \in [0, W)$, where W is the join size
  - Starting from the root $j = 1$
    - Step 1: select $t_j \in R_j$ s.t. $\mathrm{L} = \sum_{t'_j < t_j} w(t'_j) \le l < \sum_{t'_j \le t_j} w(t_j)$; then, let $l \leftarrow l - L$

$l = 10$



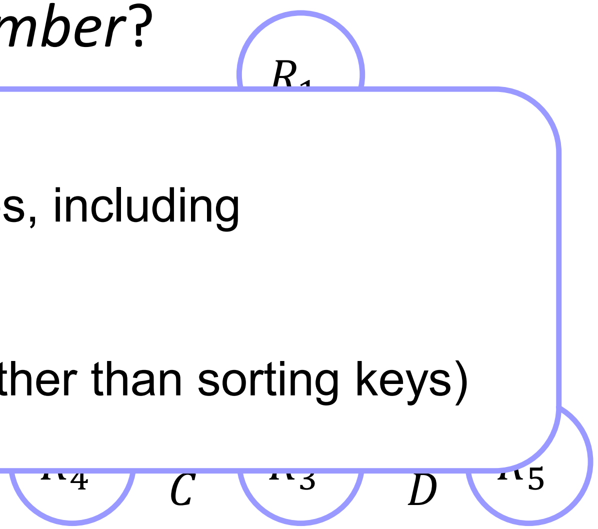
Drawing a single random join sample (cont'd)

- Step 2: for each immediate child R_k, recursively apply step 1 and 2, except that
  - Substitute R_j with $R_k[t_j]$, where $R_k[t_j]$ includes all tuples of R_k that join t_j
  - Use l mod W_k instead of l in the search, where $W_k = \sum_{t_k \in R_k[t_j]} w(t_k)$
  - Let $l \leftarrow l/W_k$ after each selection



Drawing a single random join sample (cont'd)

- How to draw random sample from a join *with just one random number*?



> - Suppose there are n tables in the join and the largest table has N tuples.
> - All ops can be implemented in $O(logN)$ time using n aggregate balanced trees, including
>   - Calculation of W and W_k
>   - Calculation of L and U
>   - Selection of "l^{th}" items (similar to std::lower_bound() but w.r.t. weights rather than sorting keys)

  - Generate a random number $l \in [0, W]$, where W is the join size
  - Starting from the root $j = 1$
    - Step 1: select $t_j \in R_j$ s.t. $\mathrm{L} = \sum_{t'_j < t_j} w(t'_j) \leq l < \sum_{t'_j \leq t_j} w(t_j)$; then, let $l \leftarrow l - L$
    - Step 2: for each immediate child R_k, recursively apply step 1 and 2, except that
      - □ Substitute R_j with $R_k[t_j]$, where $R_k[t_j]$ includes all tuples of R_k that join t_j
      - □ Use l mod W_k instead of l in the search, where $W_k = \sum_{t_k \in R_k[t_j]} w(t_k)$
      - □ Let $l \leftarrow l / W_k$ after each selection

From random sampling to reservoir sampling

- Reservoir sampling requires a unidirectional iterator over a stream
  - Need to support GetCurrent() or Skip(k)
- The algorithm for drawing a random sample
  - defines a one-to-one mapping from an index number to a join result.
  - For an inserted tuple $t_i \in R_i$, let R_i be the query tree root.
    - The batch of the new join results map from a consecutive range of

$$\sum_{t_i' < t_i} w(t_i') \leq l < \sum_{t_i' \leq t_i} w(t_i')$$

- Construct a stream of inserted join result by concatenating the batches
  - Maintain a l number in the current batch
  - Skip(k) is simply increasing l
  - GetCurrent() uses the one-to-one mapping process for random access

Optimizations

- Consolidating the tuples t_i with the same join attribute values into one vertex v_i
  - Reduces the index update cost to $\tilde{O}\big(h(v_i)\big)$
    - where $h(v_i)$ is the number of reachable vertices from v_i in the weighted join graph
    - $h(v_i) = O(d)$ when the graph has a fixed degree d
    - In contrast, symmetric join involves up to $O(d^n)$ index accesses

- Foreign-key subjoin optimization
  - Combining adjacent vertices that are connected by foreign-key join predicates
  - Save space for storing duplicate weight functions
  - See paper for details

Experiments

10GB of TPC-DS data. A 5-table many-to-many join query. Fixed-size synopsis of size 10,000 w/o replacement. All experiments use AVL trees for indexes. The synopsis is requested after every 50,000 updates.

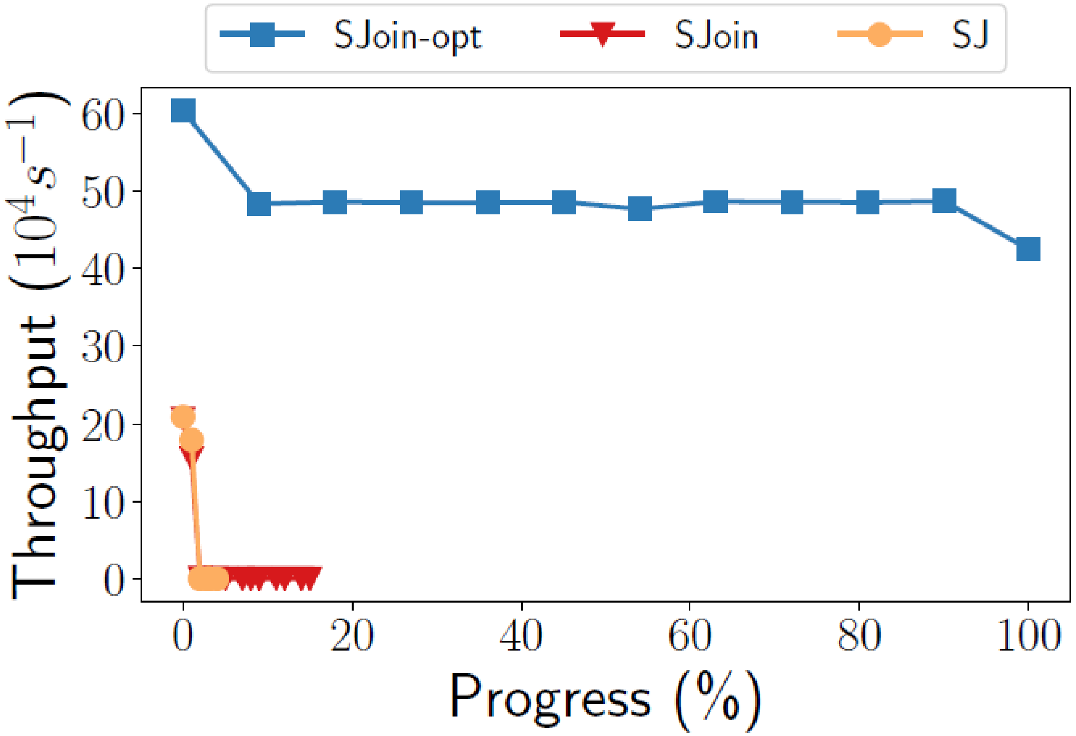


Fig 1. Insertion only.

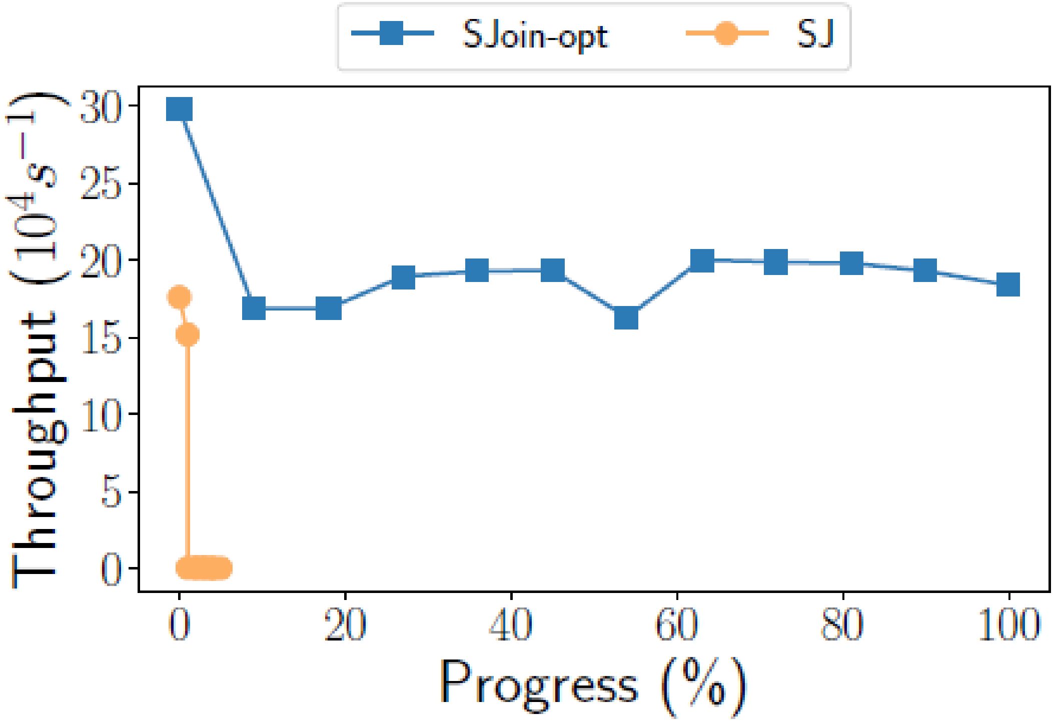


Fig 2. Insertion + deletion.

SJoin-opt: w/ foreign-key subjoin optimization. SJoin: w/o foreign-key subjoin optimization. SJ: baseline symmetric index join

Experiments

QX, QY, QZ are run on 10GB of TPC-DS data. QX, QY, QZ involve 5, 5, 7 tables respectively.
QB is run on a streaming dataset generated by Linear Road benchmark. It self-joins on 3 copies of the same table.

	SJoin-opt	SJ
QX (insertion only)	7.4 GB	8.4 GB
QY (insertion only)	3.9 GB	4.5 GB
QZ (insertion only)	4.2 GB	5.7 GB
QY (insertion and deletion)	5.6 GB	4.6 GB
QB ($d = 300$)	188 MB	151 MB

Table 3: Peak memory usage (base table + index).

SJoin-opt: w/ foreign-key subjoin optimization. SJoin: w/o foreign-key subjoin optimization. SJ: baseline symmetric index join

Conclusion

- We proposed SJoin, an efficient algorithm for maintaining join synopsis in a dynamically updated data warehouse.
- Theoretical analysis and experiments all show great performance improvements over the best-available baseline.
- We have in-memory implementation of SJoin and SJ in an experimental system.
  - will be open-sourced at https://github.com/InitialDLab

Thank you!
Q&A

Our solution

- Baseline: SJ (Symmetric index/hash Join)
  - Build conventional tree or hash indexes on all join columns
  - Incrementally maintain samples over a scan of the *full* join results
  - Up to $2n - 2$ unique indexes.
    - Storage cost is $O(nN)$, where N is the size of the largest table.
  - Maintenance cost is linear to the join size
- Our solution: SJoin (Synopsis Join)
  - Build a specialized per-query index based on *a weighted join graph*
  - Support sampling w/ or w/o replacement, or Bernoulli sampling with a *reservoir*
  - Similar storage cost ($O(nN)$ in theory, and within $\pm 25\%$ in experiments)
  - Asymptotically lower maintenance overhead in many-to-many joins
- In-memory implementation of both in an experimental system
  - Will be open-sourced at https://github.com/InitialDLab

Weighted join graph

- A join graph consists of
  - vertices that represent unique join attribute values
  - edges as a binary predicate indicating whether two join in the query

R_1

Row ID	A
0	1
1	2
2	2
3	1

R_2

Row ID	A	B
0	1	1
1	1	2
2	2	2

R_3

Row ID	B	C	D
0	1	1	1
1	2	1	2
2	2	2	2
3	2	1	2

R_4

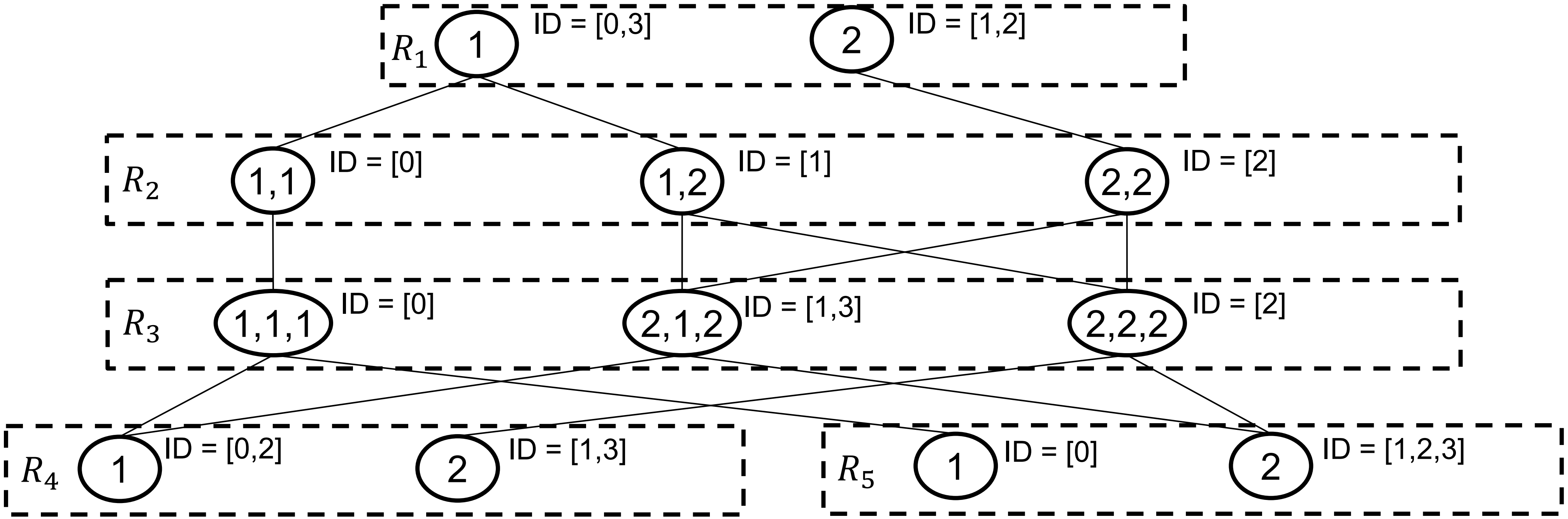
Row ID	C
0	1
1	2
2	1
3	2

R_5

Row ID	D
0	1
1	2
2	2
3	2

Weighted join graph

- A join graph consists of
  - vertices that represent unique join attribute values
  - edges as a binary predicate indicating whether two join in the query

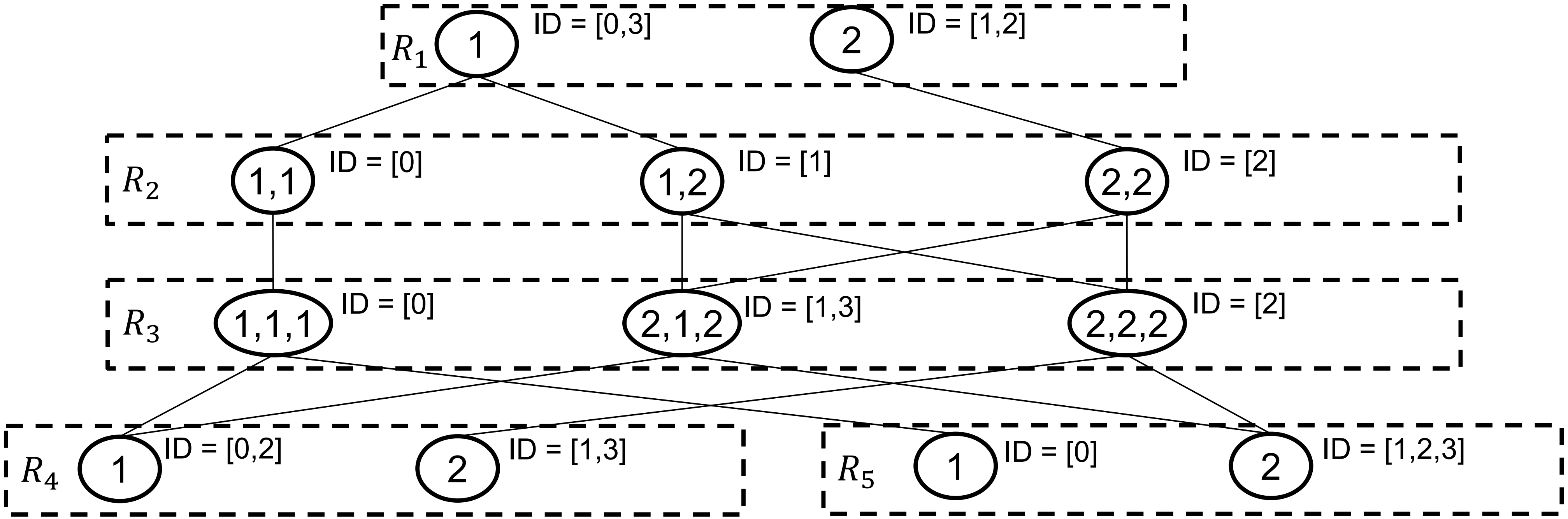


Weighted join graph

- A weighted join graph stores the unique weights that are the cardinalities of certain sub-join queries
  - Let R_i be the query tree root, we define the weights of a tuple $t_j \in R_j$ and a vertex $v_j \in R_j$ w.r.t. R_i as

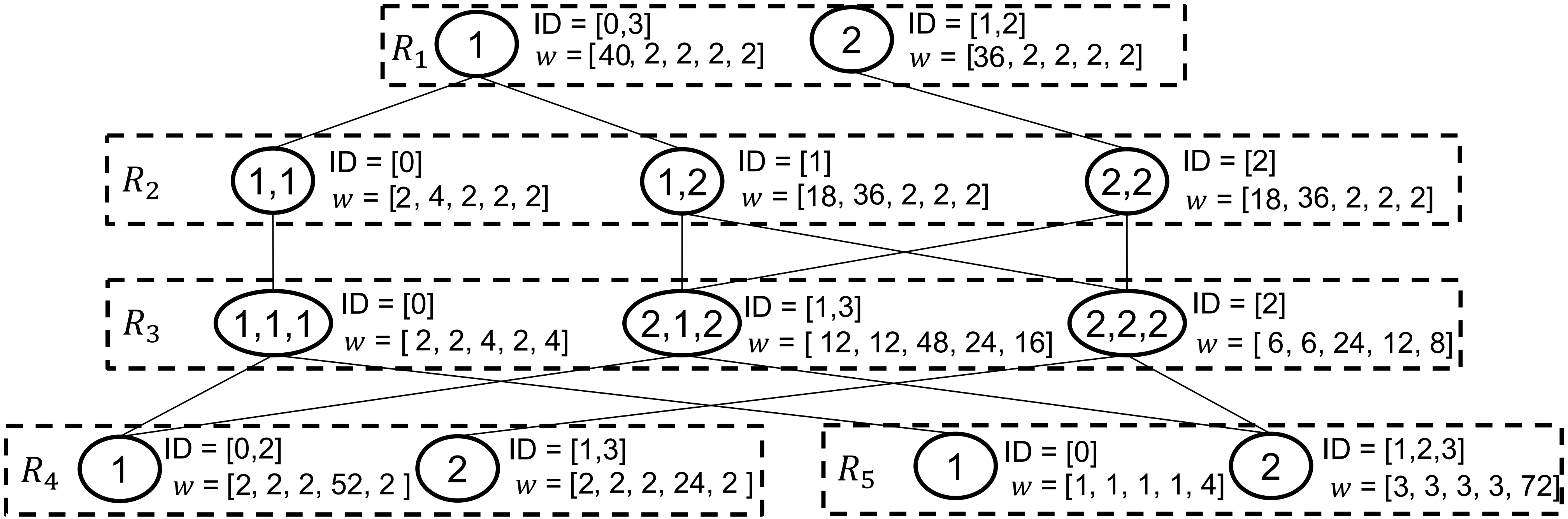
$$w_i(t_j) = \left| \bowtie \left(\mathbb{R}(j) \backslash \mathrm{R}_j\right) \bowtie \{t_j\} \right|, \qquad w_i(v_j) = \sum_{t_j \in \mathbb{T}(v_j)} w(t_j)$$

    where $\mathbb{R}(j)$ is the set of tables in the subtree at R_j and $\mathbb{T}(v_j)$ is the set of tuples that v_j represent.
  - Intuitively, it is the cardinality of the sub-join of the sub-tree at R_j that involves t_j or v_j.



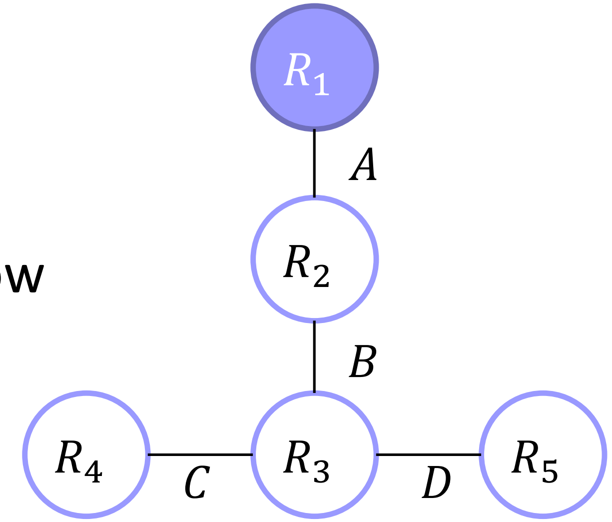
Weighted join graph

- For example, the weights w.r.t. R_1 are
  - $w_1(t_1) = |\{t_1\} \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5|$
  - $w_1(t_2) = |\{t_2\} \bowtie R_3 \bowtie R_4 \bowtie R_5|, w_1(t_3) = |\{t_3\} \bowtie R_4 \bowtie R_5|, w_1(t_4) = w_1(t_5) = 1$
  - $w_2(t_2) = |R_1 \bowtie \{t_2\} \bowtie R_3 \bowtie R_4 \bowtie R_5|$
  - $w_3(t_2) = w_4(t_2) = w_5(t_2) = |R_1 \bowtie \{t_2\}|$



Drawing a single random join sample

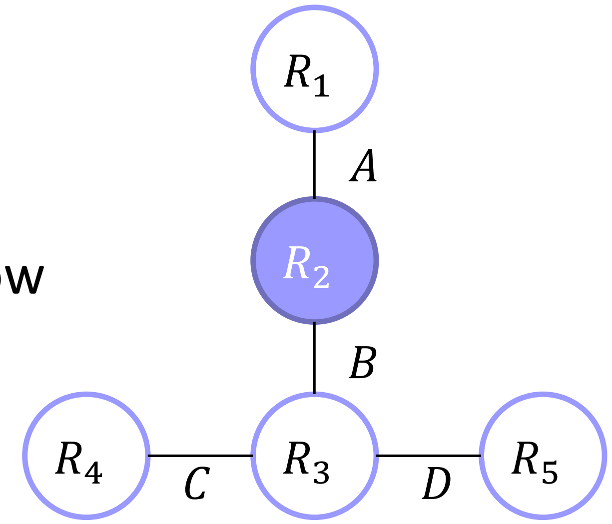
- How to draw random sample from a join?
  - Fix a join order by choosing any relation R_i as the query tree root
    - Let's say we choose R_1
    - For simplicity, omit the subscript i in the weight functions for now



  - Start from the root j $= 1$,
    - Step 1: randomly draw $t_j \in R_j$ with $p \propto w(t_j) = \left| \bowtie \left(\mathbb{R}(j) \backslash \mathrm{R}_j \right) \bowtie \{t_j\} \right|$

Drawing a single random join sample

- How to draw random sample from a join?
  - Fix a join order by choosing any relation R_i as the query tree root
    - Let's say we choose R_1
    - For simplicity, omit the subscript i in the weight functions for now



  - Start from the root $j = 1$,
    - Step 1: randomly draw $t_j \in R_j$ with $p \propto w(t_j) = \left| \bowtie \left(\mathbb{R}(j) \backslash \mathrm{R}_j \right) \bowtie \{t_j\} \right|$
    - Step 2: for each immediate child R_k, recursively apply step 1 and 2, except that
      - Substitute R_j with $R_k[t_j]$, where $R_k[t_j]$ includes all tuples of R_k that join t_j
  - Or, can it be implemented with just one random number?

Problem Formulation

Given a pre-specified SPJ query in the following form,

```
SELECT *
FROM R1, R2, …, Rn
WHERE <join-preds>
  AND <filter-preds>;
```

where a `<join-pred>` is in the form of,

- `Ri.A op Rj.B`
- `|Ri.A - Rj.B| < d`

(op is one of <, <=, =, >, >=; d is a constant)

maintain a readily available join synopsis (random sample) in a database with any insertions or deletions of tuples, for a user-specified synopsis type (fixed-size w/ replacement, fixed-size w/o replacement or Bernoulli).

- Baseline: SJ (Symmetric index/hash Join)
  - builds conventional tree or hash indexes on all the join columns
    - storage cost is $O(nN)$, where N is the size of the largest table.
  - incrementally maintains samples over a scan of the *full* join results upon insertion
    - insertion cost is at least linear to the join size (costly!)
  - rescans join upon deletion to replenish missing samples upon deletion (very costly!)

From random sampling to reservoir sampling (cont'd)

- Issue 1:
  Two batches of join results involving t_i and t_j in *different* tables have to be enumerated with *different* query tree roots R_i and R_j.

- Solution: maintain all the weights w.r.t. all the possible query tree roots
  - For a query with n tables, there are up to $2n - 2$ distinct weight functions and $2n - 2$ indexes.
  - Total storage overhead is linear:
    - Also bounded by $O(nN)$, where N is the size of the largest table
    - An additional 1 / 2 of indexing overhead for trees in practice
    - Further reduced by consolidating tuples with the same join attributes into vertices

From random sampling to reservoir sampling (cont'd)

- Issue 2:
  - Still need to draw a random number for each join result
  - Though unselected ones are never retrieved

- Solution:
  - Generate skip numbers
    - The classic Vitter's algorithm for fixed-size synopsis w/ replacement
    - Maintain m independent reservoirs for fixed-size synopsis w/o replacement
    - Use the Walker's alias algorithm to draw skip numbers for Bernoulli synopsis

From random sampling to reservoir sampling (cont'd)

- Issue 3:
  - Deletion in fixed-size sampling w/ or w/o replacement can result in insufficient number of samples

- Solution:
  - Redraw the samples using the weighted graph index using any query tree root
  - Need to deduplicate re-drawn samples for the case w/o replacement

From random sampling to reservoir sampling

- Recall that reservoir sampling
  - can maintain a fixed-size sample w/o replacement over a stream of items
  - deletion can lead to insufficient sample size - we'll deal with that later

- Here, the items are the join results.
- The 2nd algorithm for drawing a random sample
  - defines a one-to-one mapping from an index number to a join result.
  - For an inserted tuple $t_i \in R_i$, let R_i be the query tree root.
    - The batch of the new join results map from a consecutive range of

$$\sum_{t_i' < t_i} w(t_i') \leq l < \sum_{t_i' \leq t_i} w(t_i')$$

    - We can enumerate the stream by looping over the index numbers.
    - Apply RS on a view of data stream by concatenating these batches.