Efficient Join Synopsis Maintenance for Data Warehouse

Zhuoyue Zhao
University of Utah
zyzhao@cs.utah.edu

Feifei Li
University of Utah
lifeifei@cs.utah.edu

Yuxi Liu*
University of Utah
louisja1@cs.utah.edu

ABSTRACT

Various sources such as daily business operations and sensors from different IoT applications constantly generate a lot of data. They are often loaded into a data warehouse system to perform complex analysis over. It, however, can be extremely costly if the query involves joins, especially many-to-many joins over multiple large tables. A join synopsis, i.e., a small uniform random sample over the join result, often suffices as a representative alternative to the full join result for many applications such as histogram construction, model training and etc. Towards that end, we propose a novel algorithm SJoin that can maintain a join synopsis over a pre-specified general θ-join query in a dynamic database with continuous inflows of updates. Central to SJoin is maintaining a weighted join graph index, which assists to efficiently replace join results in the synopsis upon update. We conduct extensive experiments using TPC-DS and a simulated road sensor data over several complex join queries and they demonstrate the clear advantage of SJoin over the best available baseline.

CCS CONCEPTS

• **Information systems → Join algorithms**; *Data warehouses*; • **Theory of computation → Sketching and sampling**.

KEYWORDS

join synopsis; random sampling

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389717

*The author was also affiliated with Shanghai Jiao Tong University. Work was done during an internship at University of Utah.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389717

1 INTRODUCTION

A large volume of data is constantly generated from various sources such as daily business operations, sensors from different IoT applications, and etc. It is common to monitor and perform complex analysis over these frequently updated data. To explore the data, the data are often loaded into a full-fledged data warehouse system where one can run various complex analytical queries. Although the state-of-the-art data warehouse systems that produce exact answers are optimized to achieve low response time, it can still be quite slow, especially for a many-to-many join over multiple large tables. Even worse, the queries may need to be frequently re-run over the updated data to get the up-to-date results.

```
SELECT *
FROM store_sales ss,store_returns sr,catalog_sales cs
WHERE ss.item_sk = sr.item_sk
  AND ss.ticket_number = sr.ticket_number
  AND sr.customer_sk = cs.customer_sk
  AND ss.sold_date_sk <= cs.sold_date_sk
```

Figure 1: Q1: Linking store returns with subsequent catalog purchases for correlation analysis

To give a concrete example (Figure 1), suppose we have a business sales database with 3 tables. Both `store_sales` and `store_returns` have (`item_sk`,`ticket_number`) as the primary key and `customer_sk` is not a key in any table. An analyst may want to monitor and analyze a series of events where a customer returns some store purchase and subsequently places catalog orders, to study how they are correlated for marketing purposes. To do that, the analyst might issue query Q1 to identify the events and perform analysis over the join result. Note that, `store_returns` ⋈ `catalog_sales` is many-to-many because the join attribute `customer_sk` is not a key, which raises several issues:

- The join query can be prohibitively expensive to compute regardless of the algorithm used, because the result is simply too large. For instance in Q1, suppose each customer places d catalog orders after a store return on average, the join size would be d times of the size of `store_returns`, which can easily be orders of magnitude larger than any of the base tables.

- It can be too expensive to perform analysis on the join results because the input is too large. An example is to build an equi-depth histogram on the number of days elapsed before the second purchase happen after the first one. Or the analyst may be tasked to build a model to predict the subsequent catalog purchase after the return based on the join result. All of these tasks may take a very long time on the large join result.
- Incremental maintenance of the join result can also be prohibitively expensive. To maintain an exact join result, one has to at least compute the delta join result associated with any base table update, which can be quite large in a many-to-many join. After that, the analysis on the join result needs to be incrementally updated with the delta join result, or even worse, it needs to be run again in full if incremental update is unavailable for the analysis. Both of that contributes to high latency in handling updates.

The crux of the problem is the high cost of exactly computing a huge join. To avoid that, we can instead maintain a join synopsis, i.e., a small uniform random sample of the join results, for each pre-specified join query. It can serve as a representative substitute to the full join results in many tasks without the loss of too much accuracy. For example, an $\frac{fN}{k}$-deviant approximation of an equi-depth k-histogram over N items can be obtained from a random sample of size $O(\frac{k \log N}{f^2})$ with high probability. Another important use case is machine learning. There have been many studies showing, both theoretically (e.g.,[27]) and in practice (e.g.,[12, 19]), that it is feasible to use a small sample in lieu of the full data to train a model with similar errors.

Our approach is also akin to incremental view maintenance over pre-specified join queries but, instead of maintaining all the join results which are too expensive to compute, store and process, we maintain random samples of them. The key challenge is to efficiently maintain it on the fly in face of updates with minimal overhead. While there have been works related to join synopsis maintenance for foreign-key joins [1] in a dynamically updated database and random sampling over join queries [5, 34] in a static database, it is, to our best knowledge, still unclear how to maintain join synopses for general join queries in a dynamically updated database.

Our contribution. To that end, we design a novel algorithm SJoin (Synopsis Join) for maintaining join synopses on general θ-join queries in a dynamically updated database. In a nut shell, SJoin maintains a join synopsis for each pre-specified query as well as several weighted join graph indexes. The indexes encode weights, i.e., the number of join results of a subjoin starting from a tuple, which allows us to efficiently sample from the delta join results and the historical join results to replace and/or replenish join results in the synopsis without computing them in full. For instance, if we insert a new tuple into `store_returns` and it matches d tuples in `catalog_sales` in Q1, SJoin only needs to fetch a small fraction of the d join results instead of enumerating them in full. The larger the dataset is, the more CPU time SJoin can save.

n	Number of range tables in the query
N	The size of the largest range table in the query
R_i	The i^{th} range table
$R_i.A_p$	The p^{th} attribute of R_i
$\mathcal{J}$	The set of join results of the query
S	The join synopsis
G_Q	The unrooted query tree
$G_Q(R_i)$	The query tree with R_i as the root
$\delta(R_i)$	The set of range tables that join with R_i by some join predicates, i.e., neighbors of R_i in G_Q
G	The join graph
t_i, t_i', t_i''	Tuples in R_i
v_i, v_i', v_i''	Vertices of R_i in G
$w_i(v_j)$	The i^{th} weight of v_j
$W_j(v_i)$	Sum of the i^{th} weights of v_i's joining vertices in R_j

Table 1: Notations used in the paper

The rest of this paper is organized as follows:

- We define the problem of join synopsis maintenance in Section 2.
- We review the most related works in Section 3 and other related works in Section 8.
- We present the weighted join graph, a crucial index central to the SJoin algorithm, in Section 4.
- We describe the operation of SJoin in Section 5 and give a key optimization in Section 6. We also compare SJoin with the best available baseline in Section 6.
- We perform extensive experimental evaluations in Section 7 using two datasets generated by the TPC-DS data generator (a typical data warehouse setup) and the LinearRoad data generator (a typical IoT application) [2, 13, 31], which clearly demonstrates the efficiency and scalability of our approach.
- We conclude our work and discuss future directions in Section 9.

2 PROBLEM FORMULATION

Table 1 lists the major notations used throughout the paper. In this paper, we consider an SPJ query in the following form:

```
SELECT *
FROM R1, R2, ..., Rn
WHERE <join-predicates>
 [AND <filter-predicates>]
```

In the query, $R_1, R_2, \ldots, R_n$ are n range tables, which can be base tables, subquery results or views in the database. The predicates specified in the `WHERE` clause are divided into join predicates and filter predicates. A predicate between $R_i.A_p$

and $R_j.A_q$ (where A_p and A_q are two attributes) is considered as a join predicate if it is in one of the following forms:

- $R_i.A_p$ op $cR_j.A_q + d$
- $|R_i.A_p - cR_j.A_q|$ lt d,

where op is one of <, <=, >, >=, =; lt is < or <=; and c, d are constants. The definition is broad enough to include common cases such as equi-join, band-join and so on, while it is also confined to those that can be expressed as an open or close range of one attribute in terms of the other, so that we can utilize the common tree indexes to perform index joins.

Suppose $\mathcal{J}$ is the set of join results of query Q given the data in the tables. A join synopsis S is a subset of $\mathcal{J}$ that contains a uniform sample of the join results. There are three different types of join synopses:

- Bernoulli join synopsis: each join result in $\mathcal{J}$ is independently put into S with a fixed probability p.
- Fixed-size join synopsis w/o replacement: a fixed number m of distinct join results are selected into S from $\mathcal{J}$ with equal probabilities.
- Fixed-size join synopsis w/ replacement: a fixed number of m (potentially duplicate) join results are selected into S from $\mathcal{J}$ with equal probabilities.

Problem formulation. Given a database and a user-specified n-table join query Q, we want to efficiently maintain a join synopsis S of any of the three types upon any sequence of insertions and deletions to any of the n tables. The join synopsis should be ready to be returned at any time within an $O(1)$ response time, regardless of the sizes of the range tables and how many insertions and deletions have happened.

3 BACKGROUND

Join synopses are useful for saving query time by both avoiding the full join computation and reducing the input size for the subsequent operators in the query plan (e.g., aggregation, UDF). Contrary to samples produced by join sampling algorithms that are designed for aggregations [11, 14, 15], it is a uniform and independent sample of the join results and can be used in many other applications with certain accuracy guarantees (e.g., using uniform samples for approximate equi-depth histogram construction [4]) or without violating assumptions (e.g., training set in machine learning which is often assumed to be an i.i.d. sample of the population).

On the other hand, construction and maintenance of join synopses is known to be a hard problem in the literature because simply random sampling the range tables can only yield very sparse results in a typical case. For example, taking a p fraction of two tables to be joined only yields a p^2 fraction of all the join results. The more table participates in a multi-way join, the smaller fraction the final results will be. The most related work is due to Acharya et al. [1] who studied the maintenance of join synopses on a foreign-key join $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, where for any $i \geq 2$, R_i joins on a primary key with a foreign key in R_j for some $j < i$. Their observation is that the tuples in R_1 has a one-to-one mapping to the join results and thus maintenance of a join synopsis boils down to maintaining a uniform sample of R_1 and indexes over the join columns. It, however, does not work when there is some join not on foreign key and primary key columns. In addition, the work only considered the type of Bernoulli synopsis where each join result is included with a fixed probability.

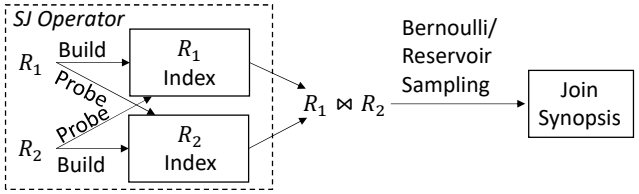


Figure 2: Operation of Symmetric Join

Another related topic is getting uniform and independent samples from joins with replacement. Chaudhuri et al. studied the problem of drawing random samples over two-table joins in [5] and Zhao et al. extended it to multi-way natural joins (aka equi-joins) in a recent work [34]. The key idea is to fix a join order and compute the weights (i.e. the number of join results of a subjoin) for each tuple in the database using a dynamic programming algorithm. Then a uniform sample *with replacement* can be obtained by sampling tuples in the fixed join order proportional to the computed weights. It, however, does not work for the purpose of join synopsis maintenance because computing the weights involves scanning all the range tables in full and thus is still quite time-consuming despite being cheaper than a full join. Moreover, it does not support general join predicates such as band joins and inequality joins, which requires careful design of indexing structures to make it efficient.

Baseline Approaches. Given the limitations of related works in the literature, we find that the only reasonable baseline for general joins is to use the Symmetric Join (SJ) to incrementally produce join results and update the join synopsis. SJ was originally proposed in the context of streaming joins [18, 32]. Figure 2 shows its operation on a two-table join. The dashed box is the SJ operator, which can produce the new join results upon insertion. It builds indexes on the fly when tuples are inserted or deleted. When a new tuple is inserted, we evaluate the subjoin in full for the new join results by probing the indexes of other tables starting from the new tuple. To apply it to a join with more than 2 tables, we can build a tree of SJ operators and recursively apply the SJ operators. When producing the new join results, we select some of them with a fixed probability for a Bernoulli synopsis, or substitute some of them in a fixed-size synopsis using the classic Reservoir Sampling algorithm. To handle deletion of a tuple, we need to remove its associated join results in the

```
interface View {
    UINT8 length();
    TID[] get(UINT8 index);
}
```

Figure 3: Iterator interface of the non-materialized delta join view

synopsis. In the case of a fixed-size synopsis, we further need to re-compute the full join to replace the missing samples. Thus , in all cases, the complexity of insertion or deletion is at least linear to the subjoin or the full join size regardless of how the join and the indexes are optimized. That makes updates very expensive, especially for many-to-many joins.

4 WEIGHTED JOIN GRAPH INDEX

Before we present the SJoin algorithm, we introduce the weighted join graph, an index central to the operation of the SJoin algorithm. The index is crucial for efficient synopsis maintenance because it serves two purposes:

- it can create a *temporary non-materialized delta join view* over the subjoin results associated with the new tuple upon insertion of a tuple into any range table. The view provides an array-like random access interface (Figure 3). Over that we can apply a variant of reservoir sampling for all three types of synopsis.
- it can be used to re-draw uniform and independent samples from the entire join results to replenish a fixed-size synopsis, when some join samples have to be removed from the synopsis because their comprising tuples are deleted from the range tables.

In a nut shell, a weighted join graph encodes all the weights needed to extract uniform sampling over a subjoin starting with an arbitrary tuple in any table. In this section, we will show how the weighted join graph is constructed and efficiently maintained, and how to perform these two tasks.

Throughout this section and the section after, we will use a 5-table acyclic join query as a running example (Figure 4). In the query, R_2, R_4 and R_5 all join with R_3 on different attributes while R_1 joins with R_2, and there are no filter predicates.

4.1 Unrooted query tree

In a conventional database, the query plan for join is often organized as a rooted tree, which implicitly defines a join order. In contrast, we build an unrooted query tree given a pre-specified join query in SJoin. It is unrooted for a reason: we might consider subjoins starting from any tables as new tuples can be inserted into any range tables. The query tree is constructed as follows. Each range table in the query is represented as a vertex in the query tree and there is an edge between two vertices if there is a join predicate between the two corresponding range tables. If the resulting graph is acyclic, which can be easily determined by a simple traversal of the graph, we already have an unrooted query tree,

```
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
```

Figure 4: A 5-way equi-join query example

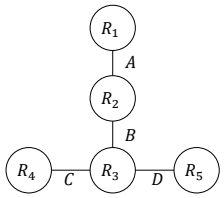


Figure 5: Query Tree G_Q for the running example

denoted as G_Q. The query is called an acyclic query in this case. If a cycle is found during the traversal, we arbitrarily remove an edge on the cycle and treat the corresponding join predicate as a filter predicate. Thus we will eventually end up with an unrooted query tree G_Q for cyclic queries as well. For instance, Figure 5 shows the resulting query tree G_Q for the running example. Since the query is already acyclic, we do not have to remove any edges to make it acyclic.

4.2 Weighted join graph

A join graph G is built on a database instance. Figure 6 shows the join graph of the running example for the instance shown at its top. For simplicity, only the join attributes are shown and each row has a unique row ID. The vertices are the distinct join attributes projected from the tuples in the tables. An edge between two vertices means that the corresponding tuples satisfy the join conditions. For example, the vertex with join attributes (2,1,2) in R_3 corresponds to tuples 1, 3 in R_3. Its neighbors in R_2, R_4, R_5 are those that satisfy the corresponding join predicates. In the remainder of this paper, vertices (resp. tuples) in R_j are denoted as $v_j, v'_j, v''_j, \ldots$ (resp. $t_j, t'_j, t''_j, \ldots$), where we use the subscript to imply the range table they belong to and the prime signs to differentiate between different vertices (resp. tuples) from the same table.

For an n-table query on a given database instance, each join result can be mapped to exactly one subgraph of G that is homomorphic to G_Q and covers exactly one vertex in each range table. On the other hand, one such subgraph may correspond to multiple join results because of tuples with duplicate join attribute values. The vector w represents a vector of weights which will be defined shortly. We denote a join result as a 5-tuple of the row IDs of its comprising tuples in the range tables. Hence, join result $(3, 1, 3, 0, 2)$ corresponds to the subgraph highlighted in orange outlines. In the mean time, there are 23 other join results that the outlined subgraph corresponds to.

Let $G_Q(R_i)$ be a rooted query tree by setting R_i as the root of G_Q. It actually corresponds to a query plan of Q that starts from a tuple in R_i, denoted as Q_i. Each subtree at root R_j in $G_Q(R_i)$ corresponds to a subjoin of Q, denoted as $Q_i(R_j)$. Hence, for each table R_j, there are up to n possible different subjoins that starts with R_j. With that, we define the set of weights of vertices as in Definition 4.1.

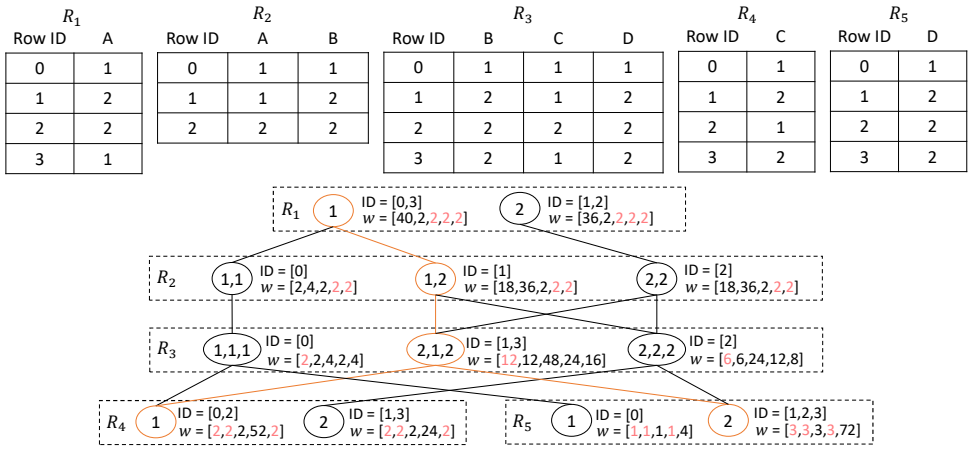


Figure 6: Join Graph of the running example on the database instance shown at the top. Non-join attributes in the tables are omitted in the figure. Each vertex has a row ID list ID and its 5 weights. The weights in red are always the same as one of the the others. The subgraph in orange corresponds to 24 join results.

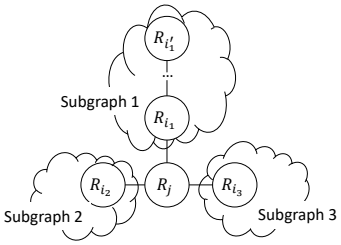


Figure 7: A stream R_j only has $d+1$ unique weight functions if its degree is d in G_Q.

Definition 4.1. Let A_j be the set of join attributes in R_j and v_j be a vertex in R_j. For a subtree in G_Q that corresponds to subjoin $Q_i(R_j)$, let $\mathbb{R}_i(R_j)$ be the set of tables in the subtree. Then i^{th} weight of v_j is defined as

$$w_i(v_j) = |\bowtie(\mathbb{R}_i(R_j) - \{R_j\}) \bowtie \{t_j \in R_j | \pi_{A_j} t_j = v_j\}|,$$

i.e., the number of results of all tuples that have the same join attribute values as v_j in R_j joining the other tables in $Q_i(R_j)$. For instance, let v_3 be the vertex (2,1,2) and t_3, t'_3 be tuples 1,3 in R_3. Then $w_1(v_3) = |\{t_3, t'_3\} \bowtie R_4 \bowtie R_5| = 12$.

The weights of each vertex in the running example are shown as a vector with a length of 5 to the right of it. Note that, for a vertex v_j, there might be weights that are always the same. For example, $w_1(v_3)$ is always the same as $w_2(v_3)$ for any $v_3 \in R_3$. This happens when they correspond to the number of join results of the same subjoin and we want to avoid computing and storing them for more than once. Fortunately, it can be easily determined with Theorem 4.2.

Theorem 4.2. *$w_i(v_j) \equiv w_{i'}(v_j)$ iff $R_{i'}$ is not in the subtree of $G_Q(R_i)$ with the subtree root being R_j.*

For example, we can determine that $w_1(v_3) \equiv w_2(v_3)$ from the fact that the subtree rooted at R_3 in $G_Q(R_1)$ only includes tables R_3, R_4, R_5, not R_2. In contrast, $w_1(v_3) \not\equiv w_3(v_3)$ because R_3 is in that subtree.

Corollary 4.3. *There are only $d+1$ unique weight functions for the vertices in a table R_j if its degree in G_Q is d.*

Corollary 4.4. *There are $3n-2$ unique weight functions across all vertices.*

A further observation is Corollary 4.3. This can be explained with the general example shown in Figure 7, where R_j has $d = 3$ neighbors in G_Q. The entire query tree can be partitioned into 4 connected components, including three that contain the neighbors $R_{i_1}, R_{i_2}, R_{i_3}$ respectively, and R_j itself. Take subgraph 1 as an example. Any $R_{i'_1}$ in the subgraph is not in the subtree of $G_Q(R_{i_1})$ rooted at R_j. By Theorem 4.2, $w_{i_1}(v_j) \equiv w_{i'_1}(v_j)$. Similar arguments also apply to subgraphs 2 and 3. The last unique weight is $w_j(v_j)$, which is the size of the entire join and thus is different from any other ones. Corollary 4.4 follows since the total degree of G_Q is $2n-2$.

4.3 Index implementation

Instead of using an explicit graph representation like adjacency list, we build multiple indexes over the vertices to implicitly represent the join graph and encode the weights. For each table R_i, we build a hash index H_i over the vertices to facilitate fast look ups with join attribute values. It will be looked up when we need to map a tuple to its corresponding vertex during insertion or deletion. We further build one aggregate tree index (which are AVL trees in our in-memory implementation) over each join attribute A of vertices in each table R_i, denoted as $I_i(A)$ and there are $2n-2$ of them. The difference between an aggregate tree index and an ordinary tree index is the former can maintain subtree aggregates of some numeric values in the vertices, which can be used to find the smallest vertex (w.r.t. the join attribute the tree is built on) whose prefix sum of the numeric value is greater than some threshold (similar to `std::lower_bound()` in C++) and compute subtree sums in any interval. In our case, we store all the unique weights in each vertex and maintain subtree aggregates of them. Let the set of R_i's neighbors in G_Q be $\delta(R_i)$. For a vertex v_i in R_i, $w_j(v_i)$ and its subtree aggregates are maintained in the $I_i(A)$ if $R_j \in \delta(R_i)$ and their join predicate is on column $R_i.A$. A special case is $w_i(v_i)$, which along with its subtree aggregates are maintained in an arbitrarily chosen tree index of R_i, denoted as $I_i(A_0)$. In addition, we also cache the total weights of the joining vertices in R_j with respect to $G_Q(R_i)$: $W_j(v_i) = \sum_{v_j \in R_j : v_j \bowtie v_i} w_i(v_j)$, as they may often be used in computation of weights but seldom updated.

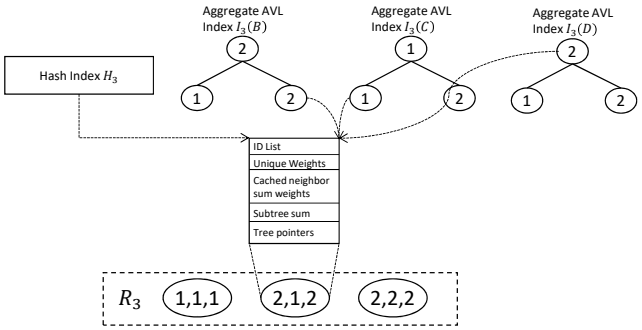


Figure 8: Indexes over R_3 in the running example

For example, we build 3 aggregate tree indexes over columns B, C, D of R_3 respectively, as shown in Figure 8. We maintain subtree aggregates of $w_2(v_3)$, $w_3(v_3)$ in $I_3(B)$; $w_4(v_3)$ in $I_3(C)$; and $w_5(v_3)$ in $I_3(D)$. One vertex is a single data structure shared by all the relevant indexes with tree pointers embedded into it. All fields in the vertex (tree pointers, weights and etc.) are accessed with an offset from the beginning from the structure, with the exception of the variable length ID list, which is implemented as a structure header with two pointers to the beginning and the end of the list. All the offsets can be determined by the query planner based on the unrooted query tree in initialization so there will be no overhead in the runtime. The benefit of this design is that we do not need to search the entire AVL index for the corresponding tree nodes when we find a vertex through the hash index or another AVL index, which is very common in the maintenance and operation of the weighted join graph index.

4.4 Index maintenance

We explain how the weighted join graph index is updated on each tuple insertion in this subsection. Deletion can be handled by reversing the operations in the insertion algorithm and thus is omitted for brevity.

Denote the set of R_i's neighbors in G_Q as $\delta(R_i)$ and the join attribute of R_i in the join predicate between R_i and R_j as $A_{i,j}$. Algorithm 1 shows the process of inserting a new tuple t_i to the weighted join graph index. On line 13, the algorithm first tries to find an existing vertex v_i that corresponds to the tuple or create one if it does not exist using the hash index H_i. In the case it is newly created, we populate the cache of the total weights of joining vertices in neighbor tables $W_j(v_i)$. On line 16 and 19, we compute all the weights for v_i using the cached total weights with Equation 1:

$$w_j(v_i) = v_i.ID.\texttt{length}() \times \prod_{R_j:R_j \in \delta(R_i) \wedge R_j \neq R_i} W_j(v_i). \quad (1)$$

The equation holds for $i = j$ because the weight is the total number of join results of v_i which is the size of the Cartesian product among the tuple ID list and the subjoin results for each child. Similarly, it holds for $i \neq j$ because in that case, the weight is the size of the Cartesian product among the tuple ID list and the subjoin results for each child except

Algorithm 1: Insert a new tuple to the weighted join graph index

```
   Input: An inserted tuple t_i in R_i
1  def updateNeighbor(i, j, u)
       //R_i: parent table
       //R_j: child table to be updated
       //u: ordered list of key-Δw pairs
2      u' ← an array of empty ordered maps ;
3      foreach ( (key, Δw) ∈ u )
4          foreach ( v_j that join with the key )
5              W_i(v_j)+ = Δw;
6              recompute w_j(v_j) ;
7              foreach ( R_j' ∈ δ(R_j) − {R_i} )
8                  w' ← w_j'(v_j) ;
9                  recompute w_j'(v_j) ;
10                 u'[j'][v_j.A_j,j']+ = w_j'(v_j) − w' ;
11     foreach ( R_j' ∈ δ(R_j) − {R_i} )
12         updateNeighbor(j, j', u'[j'])
13 v_i ← find or insert the corresponding vertex in H_i;
14 if v_i is a new vertex then
15     W_j(v_i) ← Σ_{v_j ∈ R_j : v_j ⋈ v_i} w_i(v_j) for all R_j ∈ δ(R_i) ;
16 compute w_i(v_i);
17 foreach ( R_j ∈ δ(R_i) )
18     w' ← w_j(v_i) ; //0 if it's a new vertex
19     compute w_j(v_i);
20     updateNeighbor(i, j, {(v_i.A_i,j, w_j(v_i) − w')})
```

R_j. Whenever a weight $w_j(v_i)$ is updated, v_i is inserted into $I_i(A_{i,j})$ ($I_i(A_0)$ in case $i = j$) if not already existent in the tree index, and subtree aggregates are re-computed up to the index root. Finally, the process recurses on reachable vertices in the join graph with the `updateNeighbor` function on lines 1-12. To do that, we create an ordered map u where the key is the join attribute value of the parent and the value is the total delta weight of that join attribute for each table. In the update of R_j from a parent table R_i, all weights of v_j except $w_i(v_j)$ for all vertices v_j that join some keys in u need to be updated. Note that the nested for loops lines 3-4 are essentially a join between the sorted list u and the set of vertices in R_i. For equi-join, an index nested loop suffice. It, however, can incur $O(N^2)$ cost in the worst case for band/range joins. Thus, in the latter case, we perform the join on lines 3-4 using the merge process in the sort-merge join. Because the delta weight can be accumulated on the fly, we only need to update one vertex once, which means a linear time complexity for each execution of the `updateNeighbor` function.

Theorem 4.5. *Let $h(v_i)$ be the number of vertices reachable from the corresponding vertex v_i of a tuple t_i in the weighted join graph. It takes $O(h(v_i) \log N)$ time to maintain the weighted join graph upon inserting/deleting t_i, where N is the size of the largest range table given the database instance.*

Algorithm 2: Mapping a join number to a join result

Input: A join number l, the query tree root R_i
Output: The corresponding join result $t = (t_1, t_2, \ldots, t_n)$

```
1  def mapJoinNumber(j, v_j, l)
       //R_j: table being partitioned
       //v_j: mapped vertex from R_j
       //l: remaining join number
2      w' ← w_i(v_j)/v_j.ID.length() ;
3      t[j] = v_j.ID[⌊l/w'⌋] ;
4      l ← l mod w' ;
5      foreach ( R_j's child R_k in G_Q(R_i) )
6          W ← Σ_{v_k : v_k ⋈ v_j} w_i(v_k) ;
7          l' ← l mod W;
8          l ← ⌊l/W⌋ ;
9          v_k ← min{v_k | Σ_{v'_k : v'_k ⋈ v_j ∧ v'_k < v_k} w_i(v'_k)} ;
10         mapJoinNumber (k, v_k,
             l' − Σ_{v'_k : v'_k ⋈ v_j ∧ v'_k < v_k} w_i(v'_k));
11 v_i ← min{v_i | l < Σ_{v'_i < v_i} w_i(v'_i)} ;
12 mapJoinNumber(i, v_i, l − Σ_{v'_i < v_i} w_i(v'_i))
```

With that, inserting/deleting a tuple in the weighted join graph index only incurs a linear cost in terms of the number of reachable vertices from v_i in the join graph, a cost that is always smaller than that of the index nested loop in SJ starting from the inserted/deleted tuple. To show that, we count how many index accesses there are. Consider each vertex v_j reachable from v_i, which corresponds to multiple tuples in R_j and each of which will be accessed via index in SJ. In other words, SJ performs an additional index access for every partial join result starting from t_i, which can be orders of magnitudes larger than $h(v_i)$. For instance, suppose we are doing the following join query $R_1 \bowtie_{|R_1.A-R_2.A|\leq d} R_2 \bowtie_{|R_2.A-R_3.A|\leq d} R_3$, and there are k tuples on average with the same value of A in both R_2 and R_3. Starting from a tuple t_1, SJ performs $1 + (2d+1)k + ((2d+1)k)^2$ index accesses, i.e., a $O(d^2k^2 \log N)$ cost. In contrast, the cost of weighted join graph maintenance in SJoin is only $O(d \log N)$.

4.5 Random access to join results

The main operation that the weighted join graph index provides is random access to the join results via join numbers. Let $J = |\mathcal{J}|$, i.e., the total number of join results in the database instance. A *join number* is an integer in $[0, J-1]$ that is mapped to exactly one join result, which works just like an array index in a typical programming languages. In a nut shell, we define the mapping from a join number to a join result by recursively partitioning the domain of the join numbers proportional to the weights in the join graph, in the order of preorder traversal of $G_Q(R_i)$ for some R_i. Hence, there can be n different mappings from the join numbers to the join results for an n-table join, each with respective to a different query tree root R_i.

For the mapping with respect to $G_Q(R_i)$, we totally order all vertices in a table R_j other than R_i, in the increasing order of its join attribute value in its join predicate to its parent and arbitrarily break tie when there are equal values. The vertices of the tree root R_i are instead totally ordered by the first join attribute value (which is actually an arbitrary choice) and ties are broken arbitrarily. For example, R_2 in the running example is ordered by the attribute value of column A with respect to $G_Q(R_1)$, while R_1 is ordered by the attribute value of its first column that happens to be A as well.

The mapping with respect to $G_Q(R_i)$ is shown as a function in Algorithm 2, which maps a join number $l \in [0, J-1]$ to the corresponding join result. We will explain the recursive process using the running example of $l = 30$ with respective to $G_Q(R_1)$. The algorithm recursively calls `mapJoinNumber()` to partition the domain of the join numbers, identifies the subdomain that the join number is in and remaps the subdomain to start from 0. On each table, it involves 3 steps:

(1) **(Intra-table partition)** In this step, the domain of the join number is partitioned into consecutive subdomains that are proportional to $w_i(v_k)$, in the total order of v_k's that join the previous selection of vertex v_j (all vertices in the case of root). Then we find a v_k that corresponds to the subdomain that the remaining join number l is in and recursively call the function on v_k with an adjusted remaining join number. In the running example, the domain is initially $[0, 75]$. It's divided into two subdomains $[0, 39]$ and $[40, 75]$ corresponding to vertex (1) and (2) in R_1 (lines 11-12). Since $l = 30$ falls into $[0, 39]$, `mapJoinNumber` is called on vertex (1) with $l' = 30$. Note that the lower bound of the subdomain, which happens to be 0, is subtracted from l to obtain the new remaining join number l'.

(2) **(Intra-vertex partition)** In the second step, the domain of the join number is further partitioned into subdomains of equal lengths for tuples listed in the ID list. Continuing with the running example after the first step, we have two subdomains $[0, 19]$ and $[20, 39]$ for tuple 0 and 3 in R_1 respectively (lines 2). Since $30 \in [20, 39]$, tuple 3 is selected as part of the join result (line 3), and the domain of l is readjusted to $[0, 19]$ by modulo that by the length w' of the subdomains, yielding the remaining join number of $l = 10$.

(3) **(Inter-table partition)** The final step projects the domain of the join number into several subdomains, each with a length of the total weight of joining vertices in one of the child table of R_j in $G_Q(R_i)$. Suppose the child tables are $R_{k_1}, R_{k_2}, \ldots, R_{k_{n_j}}$ from left to right, and the total weights of joining vertices in each of them are $W_1, W_2, \ldots W_{n_j}$ respectively. Then the remaining join

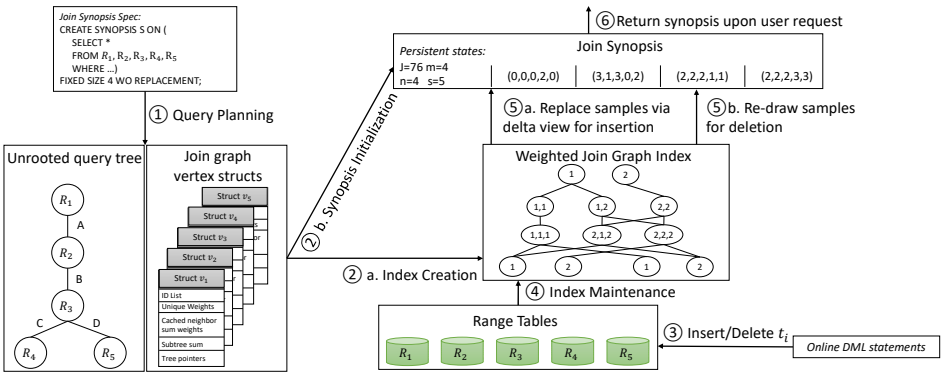


Figure 9: Operation of the SJoin algorithm

number is decomposed as n_j join numbers in the subdomains, denoted as $l_1, l_2, \ldots l_{n_j}$, with the following mapping: $l = \sum_{i=1}^{n_j} l_j \prod_{i'=1}^{i-1} W_{i'}$. Then the process continues on each join number in the subdomains from step 1 intra-table partition. In the running example, the join number does not change after step 3 applies to vertex (1) in R_1 because it only has one child R_2. Fast forward to the moment when we are at vertex (2,1,2) in R_3 with the remaining join number $l = 2$. Then we map it to two join numbers 0 and 1 for R_4 and R_5 respectively (lines 6-8).

It can be shown that the time complexity of Algorithm 2 is $O(n \log N)$ because line 6 takes $O(1)$ time by simply reading the cached value $W_k(v_j)$; line 9, 11 and the remaining join number computation on line 10, 12 can be implemented with the aggregate tree indexes in $O(\log N)$ time.

Recall the two tasks listed at the beginning of this section. They can be implemented with Algorithm 2. For the non-materialized delta join view, we find that the join numbers of all the new join results involving t_i are in a consecutive subdomain with respect to $G_Q(R_i)$, upon insertion a new tuple t_i in table R_i. Let v_i be the corresponding vertex of t_i, $U = \sum_{v'_i <= v_i} w_i(v'_i)$ and $w' = w_i(v_i)/v_j.ID.length()$. Then the subdomain is $[U - w', U - 1]$. For the view V of t_i, $V.\texttt{length}() = w'$ and $V.\texttt{get}(index)$ can simply invoke Algorithm 2 with $l = U - w' + index$ and R_i. Re-drawing uniform sample from all join results boils down to drawing a random join number in $l = [0, J - 1]$ and invoking Algorithm 2 with l and R_1 (the choice of the query tree root can be arbitrary).

5 SJOIN ALGORITHM

In this section, we give an overview of the design of the SJoin algorithm, and then provide details of how the join synopsis is maintained with insertion and deletion of tuples in SJoin.

5.1 Algorithm overview

Figure 9 shows the operation of the SJoin algorithm. At database creation, the query planner will build the unrooted query tree for the join and automatically compute the offsets of the vertex data structures in the weighted join graph index based on the pre-specified query. It then creates an empty weighted join graph. Given the pre-specified synopsis type and parameters, the query planner initializes the join synopsis and some persistent states for synopsis maintenance.

The range tables are stored as ordinary heap files in the database. The only special requirement is that the system can associate a unique tuple identifier (or TID) to each tuple across insertion and deletion. In some systems, such identifiers are already available. If TID is not available, we attach an additional monotonically increasing row ID column to each table, like what we've shown in the running example in Figure 6. A tuple can be identified by the pair of its (range) table ID and its row ID, denoted as $t_{(\texttt{table ID, index})}$. For example, $t_{(3,2)}$ denote the tuple with index 2 in R_3. In the remainder of this paper, we assume the latter is the representation of a TID. Note that the same table may appear multiple times as different range tables in the FROM clause. They can be stored in the same heap file but they must be identified as different range tables in the join. This arrangement allows us to represent a join result uniquely as a tuple of the row IDs of the comprising tuples in the range tables. For instance, join result $(1,2,0,3,2)$ is comprised of tuples $t_{(1,1)}$, $t_{(2,2)}$, $t_{(3,0)}$, $t_{(4,3)}$ and $t_{(5,2)}$.

The weighted join graph is updated whenever a range table is updated. As described in Section 4, the weighted join graph can provide two functionalities: providing a non-materialized delta join view with random access upon insertion of a tuple and re-drawing samples from all join results.

The join synopsis is updated whenever there are new/deleted join results associated with an insertion/deletion. For inserted tuples, the synopsis is updated by applying a variant of reservoir sampling to the delta join view. For deleted tuples, the persistent state of the reservoir sampling is adjusted to reflect a reduced size of join results in the database instance. A fixed-size synopsis is further replenished with re-drawn samples if its size falls below the specified size. The join synopsis is always ready to be returned at any time, upon a user request. Here, we assume the system fully serialize

all updates and synopsis requests, which can be done using simple concurrency control schemes such as locking.

Filter Handling. There could be additional filter predicates unaccounted for the join tree. The most common ones are single-table filters such as A op c for some constant. They can be applied as a pre-filter prior to insertion of tuples into the range tables. The range tables are then replaced by an indirection table where each row stores an TID to a tuple in the range table that satisfies the filter. A lesser common scenario are multi-table filters, which could be due to cyclic join queries or user-defined predicates. These are applied on top of the join synopsis upon user request. It may, however, reduce the number of valid samples in the synopsis and make it lower than specified in fixed-size sampling. To ensure the resulting synopsis size is larger than desired with high probability, we can estimate the selectivity of the multi-table filter predicates f using existing system statistics and enlarge the synopsis size by a factor of $O(1/f)$ on the fly.

5.2 Tuple insertion

Algorithm 3: Join synopsis maintenance with a view

Input: Synopsis size m or sampling rate p, Join result view V

```
1  Persistent state:
2  S ← initialize-reservoir() ;      //join synopsis
3  n ← 0 ;          //# of valid join result samples
4  J ← 0 ;        //# of join results skipped and/or
     selected
5  s ← 0 ;                    //remaining skip number
6  Pseudocode:
7  i ← 0 ;
8  while i < V.length() do
9      if i + s >= V.length() then
10         J ← J + V.length() − i ;
11         s ← s − (V.length() − i) ;
12         break;
13     i ← i + s ;
14     J ← J + 1 ;
15     t ← V.get(i);
16     Update S with t ;
17     s ← drawSkipNumber(m, n, J) ;
```

An insertion of a tuple involves inserting it into the range table and then updating the weighted join graph using Algorithm 1. Then, the weighted join graph returns a *temporary non-materialized delta join view* V, which provides random access to all the new join results associated with the inserted tuple. With V, we selectively replace some of the join results in the join synopsis with join results in V using Algorithm 3, a general framework inspired by reservoir sampling. The goal is to ensure the join results in the synopsis are a random sample of the specified type (fixed-size synopsis w/ or w/o replacement or Bernoulli synopsis), drawn from all the join results in the database. Note that the naive way to achieve that is to use the vanilla Reservoir sampling for a fixed-size synopsis w/ or w/o replacement, or to independently select each new one with a fixed probability (i.e. coin flipping) for a Bernoulli synopsis. However, the naive way requires a linear scan over all the new join results.

In contrast, Algorithm 3 makes the same random selections as if we had run the naive algorithms, but avoids the scan of the unselected ones by generating skip numbers - the number of join results not selected in a row until the next one is selected. More specifically, we maintain four persistent states across all invocation of the algorithm: S, the join synopsis; n, the number of valid join results in the synopsis; J, the total number of join results in the current database instance; and s, the remaining skip number for the next run of the algorithm. The algorithm is called on every view V. In the main loop, we skip s of the join results and then access the next one to append to or replace something in the synopsis S. After that a new value is drawn for the skip number s. The loop ends when all join results in the view V are either skipped or selected. Hence, the key to implementing the three types of synopsis is to find the correct distribution of the skip numbers and find a way to generate them in constant time, which are shown below.

Fixed-size synopsis w/o replacement. For this type of synopsis, we adopt Vitter's algorithm [29] to apply it on the non-materialized join result views. In a naive reservoir sampling for fixed-size synopsis w/o replacement, the first m items are always put into the synopsis, and the subsequent ones are selected to replace a randomly selected join result in the synopsis with probability $\frac{m}{J+1}$ given there are J join results that have been enumerated. Hence, when $n < m$, the skip number is always 0, because the next join result must be selected according to reservoir sampling. When $n = m$, the skip number follows a distribution with its probability mass function (PMF) as

$$f(s) = \frac{z}{m+s+1} \prod_{i=1}^{s} \frac{m+i-z}{m+i}.$$

Vitter's algorithm provides a rejection sampling algorithm for quickly draw s from the distribution in constant time in expectation. With a selected join result, it is appended to the join synopsis if $n < m$, or substitute a randomly selected join result in the synopsis if $n = m$.

Fixed-size synopsis w/ replacement. An m-size synopsis with replacement can be conceptually maintained as m independent 1-size synopsis without replacement. Instead, we can reuse the above framework as follows. The synopsis S is initialized as an array of NULL pointers. In each loop, imagine we have m runs of the algorithm that have made the same progress in terms of J. For the i^{th} reservoir, suppose we have the remaining skip number as s_i. Then we can compute

a maximum value $N_i = J + s_i$, which points to the last join result to skip immediately before the next to be chosen. The skip number s in Algorithm 3 can be simply computed as $s = \min_i\{N_i\} - J$. Note that N_i does not change until some join result in the i^{th} reservoir is replaced, despite being defined with J and s_i, because s_i is decreased by 1 whenever J is increased by 1. To implement `drawSkipNumber()` on line 17, we can maintain a min-heap over the N_i values. Every call to it returns the smallest N_i in the heap. The new value of N_i is computed by adding J to a newly generated skip number s_i. The update of S on line 16 can also be assisted by the min-heap. It replaces the join result in all reservoirs whose N_i equals the smallest value in the min heap.

Bernoulli synopsis. In Bernoulli synopsis, every join tuple is independently selected with a fixed probability p. Therefore, the skip number s follows a geometric distribution:

$$f(s) = (1-p)^s p.$$

To draw a random skip number in constant time in expectation, we build an alias structure [30] for the following distribution on the sub-domain $[0, \lceil 1/p \rceil + 1]$:

$$g(s) = \begin{cases} f(s) & (s <= \lceil 1/p \rceil) \\ 1 - \sum_{i=1}^{\lceil 1/p \rceil} f(s) & (s = \lceil 1/p \rceil + 1) \end{cases}$$

With the alias structure, we can draw a random number in the sub-domain from $g(s)$ in constant time. We repeatedly draw a random number s from $g(s)$ until $s \leq \lceil 1/p \rceil$. Suppose we have drawn s for k times and last draw of s ends up with the value s_k . It can be shown that $\lceil 1/p \rceil (k-1) + s_k$ follows the geometric distribution $f(s)$ and the process terminates in $O(1)$ steps in expectation.

5.3 Tuple deletion

Different from the tuple insertion case, the join graph and the join synopsis need to be updated before the tuple is deleted from the range table. A deleted tuple t_i is first identified by its TID, usually from an index lookup or from a table scan. The weighted join graph is then updated accordingly to purge the tuple from it. Since the total number of join results in the database has been decreased, the J value in Algorithm 3 is also decreased by the same amount in order for the reservoir sampling continue to work. Let v_i be the corresponding vertex in the weighted join graph index of t_i. The amount to decrease from J can be found by looking up $w_i(v_i)/v_i$.`ID.length()` in $O(1)$ time.

Furthermore, any join result in the synopsis that includes the deleted tuple also needs to be purged. To efficiently support that, we maintain a small hash table from TID to the active join result samples in the synopsis. Once some join results are purged from the synopsis, they effectively reduce n, the number of valid samples in the join synopsis. For a Bernoulli synopsis with an inclusion probability of p, the remaining join results in the synopsis still have the same inclusion probability, and thus the synopsis remains valid. On the other hand, for a fixed-size synopsis w/ or w/o replacement of size m, that can lead to an insufficient number of join results even if we still have $J >= m$. In that case, we can use the weighted join graph to re-draw random join results to replenish it as described in Section 4.5. Since the re-drawn join results are uniform and independent, putting them into the join synopsis makes it remain a uniform and independent fixed-size synopsis of size m.

An additional consideration for a fixed-size synopsis w/o replacement is that we need to reject any re-drawn join result samples that are duplicates of those in the synopsis and retry. The rejection can be high if J is close to m. To avoid that, we optimize the process by running Algorithm 3 on the view V over all the join results to recreate a new synopsis when $m \geq 1/2J$. As a result, we can bound the expected number of accesses to join results by $2m$: when $m < 1/2J$, the rejection rate is < 0.5. So it accesses 2 times of the number of missing samples in expectation, which is at most $2m$. When $m \geq 1/2J$, the number of accesses is J, which is $<= 2m$ by assumption. Note that the view V can provide random access to all join results in a similar way to a delta join view, by getting a join result via a join number in range $[0, J-1]$ with respect to $G_Q(st[i])$ for some arbitrary R_i and thus has the same of `length()` and `get()` operations as the delta join views.

5.4 Proof of correctness

Theorem 5.1. *For any join query Q, SJoin can maintain a join synopsis of any of the three types after an arbitrary sequence of insertions and deletions.*

Due to the interest of space, we only show a proof sketch of Theorem 5.1. It suffices to show that SJoin correctly maintains a synopsis of the specified type after each insertion or deletion. For each insertion, we have a view V over a new set of join results associated with the insertion and we run Algorithm 3 with the appropriate skip number distribution on V. We have shown in Section 5.2 that it makes the same random selection as the vanilla reservoir sampling for a fixed-size synopsis or the naive coin flipping algorithm for a Bernoulli synopsis. Hence, the synopsis maintains a random sample of the specified type on all the join results after the insertion. For each deletion, we have shown in Section 5.3 that the synopsis remains a random sample of the specified type after purging the deleted join results and, for a fixed-size synopsis, possibly re-drawing random join results from the remaining ones in the database using Algorithm 2.

6 ANALYSIS AND OPTIMIZATION

Now, we present a theoretical comparison on the time and storage cost of SJoin and the baseline approach and provide an important optimization in SJoin that can greatly improve its performance. Here, we assume the user limits the synopsis size to a reasonably small number, meaning m or p are

small for fixed-size or Bernoulli synopsis. Then the synopsis maintenance in *SJoin* only accesses a small number of join results (in the order of $O(m \log J)$ for fixed-size synopsis or $O(pJ)$ for Bernoulli synopsis) and thus their cost are relatively small compared to weighted join graph updates.

Insertion Cost For an insertion, the major advantage of the SJoin algorithm is that the number of vertices it needs to visit can be significantly smaller than the number of tuples it would need to visit in SJ. To give some intuition, let's consider a subjoin $t_1 \bowtie R_2 \bowtie R_3$, where t_1 joins d_1 tuples in R_2 and each of them joins d_2 tuples in R_3. Then the number of tuples visited by SJ is $d_1 d_2$. Suppose that the every m_1, m_2 tuples among the joining tuples in R_2, R_3 correspond to one vertex in the join graph, respectively. Then the number of visited vertices in SJoin is roughly $\frac{d_1 d_2}{m_1 m_2}$. In a many-to-many join, m_1, m_2 tends to be very large. For instance, a sales return record in the motivating example can match a number of catalog sales records placed by the same customer. In this case, SJoin will outperform the baseline by a significant margin. SJoin, however, may impose more overhead in the case of foreign-key join because SJoin will access the same number of vertices as the number of tuples accessed in SJ.

Foreign-key subjoin optimization. To solve that issue, we remove foreign key joins from the unrooted query tree. For each foreign key subjoin $R_i \bowtie_A R_j$, where the join key A is a primary key in R_i and a foreign key in R_j, we replace R_i, R_j with a new table $R_{j'}$ whose schema is the union of R_i's and R_j's during the query planning phase. The new table is essentially the join results of $R_i \bowtie_A R_j$. We apply this process iteratively until there is no foreign key subjoin in the query tree. During runtime, we maintain a hash table over column A of R_i and insertion of R_i only updates the hash table without triggering weighted join graph and synopsis maintenance. This is Okay because there would be no joining tuples in R_j and thus no new join results for the entire join Q, due to the foreign key constraint. Upon an insertion of a tuple $t_j \in R_j$, we look up the joining tuple in R_i using the hash table and appends it to t_j. The resulting tuple is then inserted into R'_j and it will trigger either another foreign key lookup (if it was originally in another foreign key subjoin) or maintenance of the weighted join graph and the join synopsis (if it is in the final query tree). A deletion to R_j triggers the deletion of the tuple from $R_{j'}$ and the rest are handled as usual in SJoin. Note that a deletion R_i does not trigger anything in the system other than the update of the hash table because no tuple in $R_{j'}$ can consist of it due to the foreign key constraints.

Deletion Cost Now we move on to the comparison of SJoin and SJ in terms of deletion. Both algorithms do not have much to do in Bernoulli sampling other than updating the indexes and purging associated join results from the join synopsis. It is, however, not the case for fixed-size sampling because we may need to replenish purged samples in the synopsis. We have shown that we can utilize the weighted join graph to efficiently re-draw a random sample from the full join results in $O(n \log N)$ time where n is the number of tables in the query, so the total cost is usually almost linear to the number of join results to accessed (with a log factor). On the other hand, SJ provides no capability of re-drawing random join samples so it has to rebuild the synopsis by re-computing the full join results, which is a lot more than what we have to access in SJoin. As a result, SJoin can deal with deletion much faster than the baseline.

Storage Cost Consider a pre-specified query with n tables where the largest table has N tuples. The storage cost consists of the common $O(nN)$ cost for storing the base tuples and the different indexing costs for SJoin and SJ. SJoin builds n hash tables and $2n - 2$ aggregate trees over the vertices, which translates to a storage overhead of $O(nN)$. Note that the number of vertices in any table cannot exceed N and there are only $3n - 2$ unique weights. As a result, the overhead of weights is bounded by $O(nN)$ and thus the total storage cost of SJoin is $O(nN)$. On the other hand, SJ only builds $2n - 2$ ordinary indexes, which also takes up $O(nN)$ space. Hence the storage cost of SJ is asymptotically the same as SJoin. As we will show in the experiments, SJoin's actual memory usage in practice is on par with SJ despite having a larger constant factor for storing the weights and the additional n hash tables, thanks to the savings from consolidating the tuples with the same join attribute values into a single vertex.

7 EXPERIMENTS

In this section, we present a comprehensive study on the performance of SJoin compared to the baseline approach SJ as described in Section 3. We implemented a minimal in-memory single-threaded query engine in C++ as a common platform for comparison among SJoin, the SJoin with the foreign-key join optimization (denoted as SJoin-opt), and the baseline approach SJ. We intend to open-source the implementation for easy reproduction of the results shown here. Our test environment is equipped with an Intel Core i7-3820 CPU, 64GB of memory and a magnetic hard disk.

7.1 Experimental setup

We evaluate the algorithms using several general equi-joins on data from the TPC-DS data generator [25] (a typical data warehouse setup) and a band join on the Linear Road data generator [2] (a typical IoT application). We measure the *instant* throughput at different checkpoints during the process of continuously inserting or deleting tuples from the tables. The instant throughput is approximated with the average number of insertions or deletions performed per second in a 5-second window around the time of measurement.

```
(QX) SELECT *
FROM store_sales, store_returns, catalog_sales,
     date_dim d1, date_dim d2
WHERE ss_item_sk = sr_item_sk
  AND ss_ticket_number = sr_ticket_number
  AND sr_customer_sk = cs_bill_customer_sk
  AND d1.d_date_sk = ss_sold_date_sk
  AND d2.d_date_sk = cs_sold_date_sk;
(QY) SELECT *
FROM store_sales, customer c1, household_demographics d1,
     customer c2, household_demographics d2
WHERE ss_customer_sk = c1.c_customer_sk
  AND c1.c_current_hdemo_sk = d1.hd_demo_sk
  AND d1.hd_income_band_sk = d2.hd_income_band_sk
  AND d2.hd_demo_sk = c2.c_current_hdemo_sk;
(QZ) SELECT *
FROM store_sales, customer c1, household_demographics d1,
    item i1, customer c2, household_demographics d2, item i2
WHERE ss_customer_sk = c1.c_customer_sk
  AND c1.c_current_hdemo_sk = d1.hd_demo_sk
  AND d1.hd_income_band_sk = d2.hd_income_band_sk
  AND d2.hd_demo_sk = c2.c_current_hdemo_sk
  AND ss_item_sk = i1.i_item_sk
  AND i1.i_category_id = i2.i_category_id;
(QB) SELECT *
FROM lane1, lane2, lane3
WHERE |lane1.pos - lane2.pos| <= d
  AND |lane2.pos - lane3.pos| <= d
```

Figure 10: Queries used in the experiments. Tables that have online insertion or deletion are highlighted in bold face. QX, QY, QZ are run on the TPC-DS dataset while QB is run on the Linear Road dataset. The d in QB is a query parameter varying in our experiments.

For the TPC-DS dataset, we use a scale factor of 10. We do not use a larger dataset because the data distribution remains the same in TPC-DS regardless of the size and the performance curve stabilizes after inserting a handful of tuples. Some of the smaller dimension tables such as `date_dim, household_demographics and etc.`, are preloaded while larger ones are updated on the fly. The fact tables `store_sales`, `store_returns`, `catalog_sales`, `customer`, `item` have about 29M, 2.9M, 14M, 500K, 102K tuples to be inserted. To test deletion performance, we delete about 20% of the oldest tuples while new ones are being inserted. If a table appears twice in a query, it might be duplicated for ease of implementation but that does not have impact on our findings here. The linear road benchmark, which is originally a streaming benchmark, consists of position data along with other information sent every 30 seconds from each vehicle on a simulated highway. We load into 3 tables the tuples from 3 specific car lanes in timestamp order and delete any tuple that is more than 60 seconds older than the newest tuple in the system. By default, we build a fixed-size synopsis without replacement of size 10,000 for all experiments unless otherwise specified. We also simulate the request for the join synopsis by reporting run-time statistics of the join synopsis after every 50,000 insertions/deletions are processed. Finally, Figure 10 lists the joins used in our experiments, each of which has at least one many-to-many subjoin.

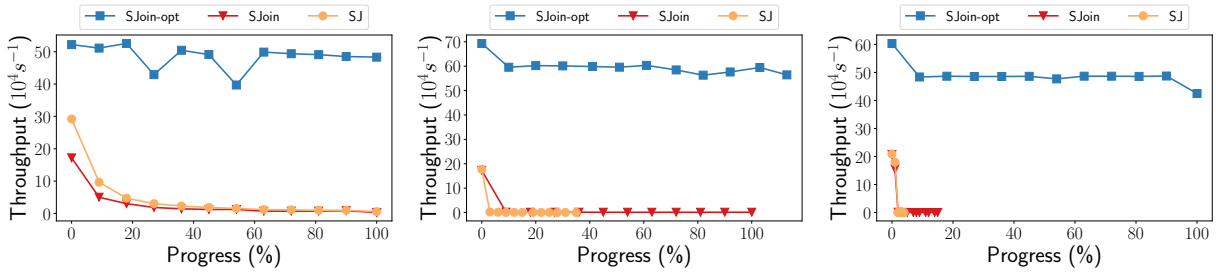
7.2 Insertion throughput

In our first batch of experiments, we evaluate the insertion throughput by maintaining the default fixed-size join synopsis w/o replacement of size 10,000 for QX, QY, QZ on the TPC-DS dataset with insertions only. Figure 11 shows the instant throughput plotted against the loading progress (i.e., the percentage of insertions processed). Some of the lines are incomplete (SJ in QY and SJoin, SJ in QZ) because they failed to finish processing all insertions in 6 hours.

As expected, the general trend for all the algorithms is that the throughput drops after an initial phase where join results are sparse and are easy to compute and then stabilizes once we have loaded a small fraction of the tuples into the database. We can observe that the optimized SJoin (SJoin-opt) achieves about 167, 1400 and 8036 times higher insertion throughput than the baseline SJ for QX, QY and QZ respectively, which shows SJoin-opt can maintain high insertion throughput for join synopses over complex multi-way joins.

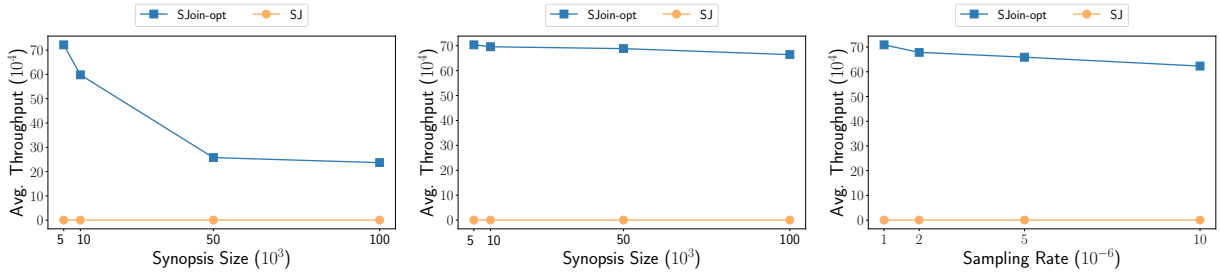
What's interesting is the crucial role of the foreign-key subjoin optimization in improving the performance, without which SJoin only outperforms SJ by 3x and 4x in QY and QZ, and even has a throughput drop of about 40% compared to SJ in the case of QX. That is due to the cost of looking up an excessive number of vertices and the additional overhead of updating weights in the weighted join graph. It simply outweighs the savings in avoiding enumerating join results.

Take QX as an example, and there is a foreign key subjoin between `store_returns` and `store_sales`. That means every vertex in `store_sales` maps to exactly one tuple because the join key `(ss_item_sk, ss_ticket_number)` is a primary key. It happens that the corresponding foreign key `(sr_item_sk, sr_ticket_number)` is also a primary key in `store_returns` and thus it also has one tuple per vertex. Now, consider an insertion into `catalog_sales` which join with d tuples in `store_returns`. We will update d vertices in each of the first two tables, and up to d vertices in both of the two date_dim tables. That translates to about $2d$ to $4d$ index lookups and additional weight updates which also require additional index updates for subtree aggregates. In contrast, SJ only performs the same amount of index lookups and no other additional operations. It does not happen for SJoin-opt because the primary keys are no longer included in the vertices and one vertex can correspond to many tuples. Thus, the total number of updated vertices is significantly smaller than the number of join results. Hence, we achieve 167 higher insertion throughput despite having more computation to do on each vertex in SJoin than on each tuple in the



(a) QX (b) QY (c) QZ

Figure 11: Maintain join synopses for QX, QY, QZ with insertions only



(a) Fixed-size synopsis w/o replacement **(b) Fixed-size synopsis w/ replacement** **(c) Bernoulli synopsis**

Figure 12: Maintain different types of synopses with varying parameters

baseline SJ. For that reason, we do not report the numbers of the unoptimized SJoin in the remainder of the experiments.

7.3 Deletion throughput

To evaluate the deletion performance, we run QY on the TPC-DS dataset with 20% of the tuples being deleted while the tuples are inserted. Specifically, we delete the oldest 600 and 100 tuples from `store_sales`, `customer` `c2` tables after every 3000 and 500 tuples are inserted into them. Figure 13 shows the experimental results of SJoin-opt and the baseline SJ. We discover that SJoin-opt maintains about a third of the throughput compared to an insertion only workload, because of the additional bookkeeping and cost required, which includes maintaining a hash set of distinct join samples in the synopsis for rejecting duplicate samples re-drawn after deletion of join samples in the synopsis, maintaining a hash map that maps from TID to join samples in the synopsis that assists identifying those deleted ones and so on. The performance gap between SJoin-opt and SJ becomes larger because the latter has to rebuild the join synopsis by re-computing the full join every time we delete something from a table. As a result, SJ is only able to process about 5% of all the inputs in 6 hours while SJoin-opt finished processing all the 35.6 million inputs in about 3 minutes.

7.4 Different types of synopsis

We further test how SJoin-opt handles the maintenance of different types of join synopsis with varying parameters. We run QY with insertion only as in Section 7.2, but with 3 types of join synopsis and 4 parameters for each type. We plot the overall average throughput against the synopsis size (for the fixed-size ones) or the sampling rate (for the Bernoulli one) in Figure 12. We find that in either case, SJoin-opt can consistently maintain a high throughput compared to SJ, regardless of the type of synopsis to be maintained.

7.5 Varying join fanout

Join fanout of a join $R_1 \bowtie R_2$ is the average number of joining tuples in R_2 with a tuple in R_1. The higher the join fanout is, the more expensive the join is because there will be more join results. In this experiment, we run the band join QB on the Linear Road dataset with varying d values. By setting a higher d value, the join fanout between adjacent two lanes becomes higher. Figure 14 shows how the join fanout affects the throughput of SJoin compared to SJ. In this case, SJoin-opt scales linearly with an additional log factor in terms of d, because the number of vertices updated upon each insertion and/or deletion is linear to d. On the other hand, SJ's throughput drops to almost 0 due to 2 reasons: 1) each insertion is associated with $O(d^2)$ new join results; 2) each deletion triggers a full re-computation of join results. Clearly, SJoin-opt can maintain a high throughput even if the join fanout gets larger when the data distribution changes, while the baseline SJ cannot handle that.

7.6 Storage overhead

A question one might ask is how much storage does SJoin-opt consume in practice, compared to SJ that only maintains conventional tree indexes. Since our implementation is in-memory, we report the peak memory usage in Table 2. For

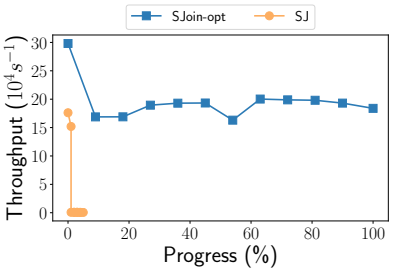


Figure 13: Maintain a join synopsis for QY with insertions and deletions

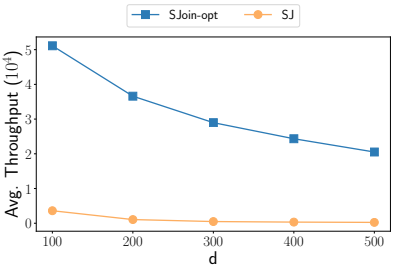


Figure 14: Maintain a join synopsis for QB with varying join fanout

	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 2: Peak memory usage

each query, it includes the total space of the range tables and the indexes. As indicated in the table, the memory usage of SJoin-opt is on par with SJ. It can even be smaller in certain cases despite having additional weights stored in the indexes. That is because the tuples with the same join attribute values are combined into a single vertex that only stores one set of tree pointers for an index in SJoin-opt. In contrast, they are in different tree nodes in SJ and thus consume more space for pointers. Overall, we find that the the memory usage of SJoin-opt is within about ±25% of that of SJ.

8 RELATED WORK

We have reviewed the most relevant studies on join synopsis maintenance and join sampling in Section 3. In short, the only prior work on join synopsis maintenance was due to Acharya et al. [1] on the foreign-key join case. To the best of our knowledge, there is no existing work on the general θ−join. Join sampling on a static database for general join queries was a better-studied topic in the literature [5, 34]. They, however, do not work for join synopsis maintenance on a dynamically updated database without a lot of re-computation.

There have been notable interests in the problem of approximate query processing over join queries. A common approach is using samples to estimate aggregations [11, 14, 15]. They often relax the uniformity or the independence properties of samples to make them easier to compute. The samples are sufficient for aggregation purpose, but are not suitable for general tasks. Sketches [3, 6, 9, 16, 20, 21] are also commonly used for approximate frequency estimation and join size estimation, and [8] uses sketches to answer SUM and COUNT queries over acyclic joins. But they only work for the specific tasks they are designed for. On the other hand, a join synopsis consists of a random sample of the join results and thus can be used for general tasks with provable guarantees. That said, since we focus on the efficiency of join synopsis maintenance instead of investigating how they can be used for specific tasks, an experiment comparing the quality of approximation with a join synopsis and sketches on a specific task is not included in the paper due to the space constraint.

Another related area of study is join over stream data [10, 17, 26, 28, 32], where streams are tables that have tuples being inserted and deleted all the time in the sequence of timestamps. Many have studied how to minimize certain error measures resulted from dropping tuples from memory prematurely due to memory constraints. A popular measure is to maximize the join output [7, 22, 24, 33] but the results may not be statistically meaningful due to bias in the join. In comparison, a join synopsis is generally a representative substitute for the full join. There have also been works on the topic of random sampling over joins of stream data [22, 23] but they are tied to specific application (e.g., aggregation) and/or have restrictive assumptions on the data distribution.

9 CONCLUSION

To summarize, we present a novel algorithm SJoin for efficient join synopsis maintenance over general join queries in a dynamically updated data warehouse, which is useful for monitoring and analytical tasks over recurrent join query results. Our experiments shows SJoin can maintain a high throughput with different queries and different types of synopses and outperforms the best available baseline SJ by orders of magnitudes. That being said, an interesting and challenging future direction is to investigate how to implement it in a distributed or parallel database to allow concurrent updates and queries, which may require careful redesign of the index structure for concurrent reads and writes as well as a finer-grained concurrency control scheme that guarantees consistency of the weights with low overhead.

10 ACKNOWLEDGMENT

Feifei Li, Zhuoyue Zhao and Yuxi Liu are supported by NSF grant 1619287, 1801446, 1816149. The authors thank Dr Bin Wu of the database and storage lab at Alibaba Cloud for the feedbacks during the preparation of the paper. The authors also greatly appreciate the valuable feedbacks from the anonymous SIGMOD reviewers.

REFERENCES

[1] 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)*. ACM, New York, NY, USA, 275–286. https://doi.org/10.1145/304182.304207

[2] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *VLDB*.

[3] Moses Charikar, K. Chen, and Michael Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Proceedings International Colloquium on Automata, Languages and Programming*.

[4] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1998. Random sampling for histogram construction: how much is enough?. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[6] G. Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55 (2005), 58–75.

[7] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate Join Processing over Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD âĂŹ03)*. Association for Computing Machinery, New York, NY, USA, 40âĂŞ51. https://doi.org/10.1145/872757.872765

[8] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2002. Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD âĂŹ02)*. Association for Computing Machinery, New York, NY, USA, 61âĂŞ72. https://doi.org/10.1145/564691.564699

[9] Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. 2004. Processing Data-Stream Join Aggregates Using Skimmed Sketches. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari (Eds.), Vol. 2992. Springer, 569–586. https://doi.org/10.1007/978-3-540-24741-8_33

[10] Lukasz Golab and M. Tamer Özsu. 2005. Update-pattern-aware Modeling and Processing of Continuous Queries. In *SIGMOD*. 658–669.

[11] P. J. Haas and J. M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*. 287–298.

[12] Silu Huang, Chi Wang, Bolin Ding, and Surajit Chaudhuri. 2019. Efficient Identification of Approximate Best Configuration of Training in Large Datasets. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. 3862–3869. https://doi.org/10.1609/aaai.v33i01.33013862

[13] IBMStreams. 2016. IBM Streams Linear Road Benchmark. https://github.com/IBMStreams/benchmarks/tree/master/StreamsLinearRoadBenchmark

[14] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 631–646. https://doi.org/10.1145/2882903.2882940

[15] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*.

[16] J. Misra and D. Gries. 1982. Finding repeated elements. *Sc. Comp. Prog.* 2 (1982), 143–152.

[17] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. 2004. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, Washington, DC, USA, 251–. http://dl.acm.org/citation.cfm?id=977401.978115

[18] AJoin nnita N. Wilschut and Peter M. G. Apers. 1991. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, USA, December 4-6, 1991*. 68–77. https://doi.org/10.1109/PDIS.1991.183069

[19] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD âĂŹ19)*. Association for Computing Machinery, New York, NY, USA, 1135âĂŞ1152. https://doi.org/10.1145/3299869.3300077

[20] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, 1449âĂŞ1463. https://doi.org/10.1145/2882903.2882948

[21] Florin Rusu and Alin Dobra. 2008. Sketches for Size of Join Estimation. *ACM Trans. Database Syst.* 33, 3, Article 15 (Sept. 2008), 46 pages. https://doi.org/10.1145/1386118.1386121

[22] Utkarsh Srivastava and Jennifer Widom. 2004. Memory-limited Execution of Windowed Stream Joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 324–335. http://dl.acm.org/citation.cfm?id=1316689.1316719

[23] Yufei Tao, Xiang Lian, Dimitris Papadias, and Marios Hadjieleftheriou. 2007. Random Sampling for Continuous Streams with Arbitrary Updates. *IEEE TKDE* 19, 1 (2007), 96–110.

[24] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. 2005. RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. ACM, New York, NY, USA, 371–382. https://doi.org/10.1145/1066157.1066200

[25] Transaction Processing Performance Council. 2019. TPC Benchmark DS. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.11.0.pdf

[26] Tolga Urhan and Michael J. Franklin. 2001. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *VLDB*. 501–510.

[27] V. N. Vapnik and A. Y. Chervonenkis. 1971. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications* 16 (1971), 264–280.

[28] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. 2003. Maximizing the Output Rate of Multi-way Join Queries over Streaming Information Sources. In *VLDB*.

[29] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985).

[30] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256.

[31] Walmart Labs. 2016. Walmart Labs Streams Linear Road Benchmark. https://github.com/walmartlabs/LinearGenerator

[32] Junyi Xie and Jun Yang. 2007. A Survey of Join Processing in Data Streams. In *Data Streams - Models and Algorithms*. 209–236.

[33] Junyi Xie, Jun Yang, and Yuguo Chen. 2005. On Joining and Caching Stochastic Streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD âĂŹ05)*. Association for Computing Machinery, New York, NY, USA, 359âĂŞ370. https://doi.org/10.1145/1066157.1066199

[34] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*. 1525–1539.