Memory-Limited Execution of Windowed Stream Joins*

Utkarsh Srivastava Jennifer Widom

Stanford University
{usriv,widom}@db.stanford.edu

Abstract

We address the problem of computing approximate answers to continuous sliding-window joins over data streams when the available memory may be insufficient to keep the entire join state. One approximation scenario is to provide a *maximum subset* of the result, with the objective of losing as few result tuples as possible. An alternative scenario is to provide a *random sample* of the join result, e.g., if the output of the join is being aggregated. We show formally that neither approximation can be addressed effectively for a sliding-window join of arbitrary input streams. Previous work has addressed only the maximum-subset problem, and has implicitly used a *frequency-based model* of stream arrival. We address the sampling problem for this model. More importantly, we point out a broad class of applications for which an *age-based* model of stream arrival is more appropriate, and we address both approximation scenarios under this new model. Finally, for the case of multiple joins being executed with an overall memory constraint, we provide an algorithm for memory allocation across the joins that optimizes a combined measure of approximation in all scenarios considered. All of our algorithms are implemented and experimental results demonstrate their effectiveness.

1 Introduction

Data stream systems [14, 18, 22] face the challenge that immediate online results often are required, but sufficient memory may not be available for the run-time state required by a workload of numerous queries over high-volume data streams [7, 13]. There are two basic solutions: provide *approximate* instead of accurate query results using memory exclusively to ensure high performance [2, 7, 9], or provide accurate results by using disk with the risk of failing to keep up with the input rate [7, 19]. In this paper, we address the problem of memory-limited execution of *sliding-window joins* [2] in data stream systems, focusing on providing approximate results.

Consider a continuous sliding-window join between two streams S_1 and S_2, denoted as $S_1[W_1] \bowtie_\theta S_2[W_2]$. Windows W_1 and W_2 consist of the most recent tuples on their respective streams, and may be tuple-based (e.g., the last 1000 tuples), or time-based (e.g., tuples arriving in the last 10 minutes). The output of the join contains every pair of tuples from streams S_1 and S_2 that satisfy the join predicate θ and are simultaneously present in their respective windows. In general, to perform the join accurately, the entire contents of both windows must be maintained at all times. If we have many such joins with large windows over high-volume data streams, memory may be insufficient for maintaining all windows in their entirety. If the data stream application has stringent performance requirements (to preclude the use of disk), but can tolerate an approximate join result, there are two interesting types of approximation:

1. **"Max-Subset" Results:** If the application benefits from having a maximum subset of the result, we can selectively drop tuples (sometimes referred to as *load shedding* [7, 17]) with the objective of maximizing the size of the join result produced.

2. **Sampled Results:** A random sample of the join result may often be preferable to a larger sized but arbitrary subset of the result. For example, if the join result is being aggregated, the sample can be used to provide a consistent and unbiased estimate of the true aggregate.

Previous work on memory-limited join execution [7, 13] has considered only max-subset results, and has implicitly assumed a *frequency-based* model of stream arrival. In this model, each join-attribute value has a roughly fixed frequency of occurrence on each stream. These frequencies (either known or inferred through monitoring) are used to make load-shedding decisions, i.e., which tuples to drop and which to retain, in order to maximize the size of the join result produced. However, no justification has been provided as to why this (or any other) model is required for addressing the max-subset approximation problem. Our first contribution is to show formally that if a sliding-window join over arbitrary streams is to be executed without enough memory for retaining the entire windows, neither of the above types of approximations can be carried out effectively: For the max-subset problem, any online algorithm can return an arbitrarily small subset as compared to the optimal (offline) algorithm [7], and for the sampling problem, no algorithm can guarantee a nonzero uniform random

*This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799 and by a Sequoia Capital Stanford Graduate Fellowship.

Model	Max-Subset	Random Sample
Age-Based	Section 3	Section 4
Frequency-Based	Addressed in [7]	

Figure 1: Problem space

sample of the join result. Thus, we must have some model of stream arrival to make any headway on the problem.

There are many applications for which the frequency-based model considered in previous work is inappropriate. (One obvious case is a foreign-key join, where on one stream each value occurs at most once.) For these applications, we define an *age-based* model that is often appropriate and enables much better load-shedding decisions. In the age-based model, the expected join multiplicity of a tuple depends on the time since its arrival rather than on its join-attribute value. Examples will be given in Section 2.2.

Given the two types of approximation and the two models, we have the problem space shown in Figure 1. The max-subset problem has been addressed in [7], but only for the frequency-based model. To the best of our knowledge, the sampling problem, i.e., the problem of extracting a random sample of the join result with limited memory, has not been addressed in previous work. Our contribution is to address the max-subset problem for the age-based model, and the sampling problem for both models.

Our discussion so far assumes a single two-way sliding-window join. In reality, we expect to be executing many queries simultaneously in the system. Thus, there is an added dimension to all of the above problems: memory allocation among multiple joins. The total available memory should be allocated to the different joins such that a combined approximation measure is optimized. We provide an optimal memory allocation scheme that minimizes the maximum approximation error in any join. Our technique also extends to the weighted case, i.e., when different joins have different relative importance.

1.1 Related Work

There has been considerable work recently on data stream processing; see [11] for a survey. We discuss only the body of work related to answering queries approximately when available memory is insufficient. This work can be broadly classified into two categories. One category consists of load-shedding strategies for max-subset approximation of joins. Random load-shedding is the simplest, and has been considered in [13]. [7] primarily considers the offline load-shedding problem (one in which all future tuple arrivals are known), and provides some heuristics for the online case that implicitly assume a frequency-based model. An alternative stream model for load-shedding uses a stochastic process [20]. Although this model is more general, the primary focus in [20] is on scenarios in which the tuples arriving on one stream are independent of those that have already arrived on another stream. However, most scenarios we consider do not exhibit this independence, e.g., our age-based example in Section 2.2. Moreover, the process of inferring a general stochastic process merely by observing the stream is not clear.

The other category consists of randomized sketch-based solutions for approximately answering aggregate queries over joins, providing probabilistic error guarantees [1, 9]. These techniques do not extend to handle sliding-window joins or windowed aggregates which are required in many applications: although the techniques handle explicit deletions within streams, they cannot handle the implicit deletions generated by sliding windows.

In this paper, we only consider the stream system being memory-limited. The stream system could instead (or also) be CPU-limited, i.e., the rate of incoming tuples is higher than can be processed. Load-shedding for the CPU-limited case has been considered in [4, 17]. Sampling from a window is addressed in [3], but only for a single stream and not for a join result. Random sampling for joins has been considered in the relational context [5]. However, all sampling methods developed there require repeated access or indices on at least one of the relations, making these techniques inapplicable in the stream context.

1.2 Summary of Contributions

1. We show formally that the problem of approximating a sliding-window join with limited memory cannot be addressed effectively for arbitrary streams (Sections 3.1 and 4.1).
2. We introduce a novel *age-based model* for stream arrival that captures many applications not captured by the frequency-based model assumed in previous work (Section 2).
3. For a single two-way join with a fixed memory constraint, we provide novel algorithms for the max-subset problem under the age-based model (Section 3), and the sampling problem under both the frequency and age-based models (Section 4).
4. For multiple two-way joins with an overall memory constraint, we give an algorithm to allocate memory among the various joins so as to optimize a combined measure of approximation (Section 5).
5. We provide a thorough experimental evaluation showing the effectiveness of our techniques (Section 6).

2 Preliminaries and Models

We briefly describe our basic model of continuous query processing over data streams. Assume any discrete time domain. For a stream S_i, $i = 1, 2$, a variable number of tuples may arrive in each unit of time. However, we assume that over time, tuples on stream S_i arrive at a constant average rate of r_i tuples per unit time. $S_i[W_i]$ denotes a window on stream S_i. We consider *time-based* windows, where W_i denotes the length of the window in time units. At time t, a tuple s belongs to $S_i[W_i]$ if s has arrived on S_i in the time interval $[t - W_i, t]$. At time t, we say s is of age k if it arrived at time $t - k$. We consider time-based windows for generality; tuple-based windows can also be captured by assuming that a single tuple arrives every time unit.

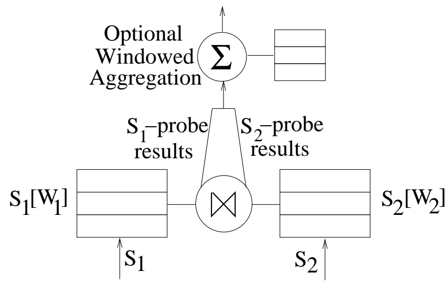

Figure 2: Sliding-window join with aggregation

The basic query we consider (shown in Figure 2) is a sliding-window equijoin between two streams S_1 and S_2 over a common attribute A, denoted $S_1[W_1] \bowtie_A S_2[W_2]$. The output of the join consists of all pairs of tuples $s_1 \in S_1$, $s_2 \in S_2$, such that $s_1.A = s_2.A$ and at some time t, both $s_1 \in S_1[W_1]$ and $s_2 \in S_2[W_2]$. Conceptually, a sliding-window join is executed as shown in Figure 3, which details the steps to be followed for a newly arriving tuple on S_1. A symmetric procedure is followed for a newly arriving tuple on S_2. We also consider queries with a *windowed aggregation* operator on top of the streamed join result. Other work [8] has focused on approximate windowed aggregation in memory-limited environments. We do not consider this aspect of memory usage in our calculations, however analyzing the tradeoff between memory allocation to joins and aggregation is an interesting subject of future work.

We classify every join-result tuple as either an S_1*-probe join tuple* or an S_2*-probe join tuple*. When a new tuple s arrives on S_1 and joins with a tuple $s' \in S_2[W_2]$ (line 3 of Figure 3), s and s' are said to produce an S_2-probe join tuple. S_1-probe join tuples are defined symmetrically. A tuple $s \in S_1$ may first produce S_2-probe join tuples when it arrives. Then, before it expires from $S_1[W_1]$, it may produce S_1-probe join tuples with newly arriving tuples on S_2. We use $n_i(s)$, $i = 1, 2$, to denote the number of S_i-probe join tuples produced by a tuple $s \in S_i$ before it expires from $S_i[W_i]$.

2.1 Frequency-Based Stream Model

Continue to consider the sliding-window join $S_1[W_1] \bowtie_A S_2[W_2]$. Let $\mathcal{D}$ denote the domain of join attribute A. The frequency-based model that has been assumed in previous work [7, 13] is defined as follows:

Definition 2.1 (Frequency-Based Model). *For each value $v \in \mathcal{D}$, a fixed fraction $f_1(v)$ of the tuples arriving on S_1, and a fixed fraction $f_2(v)$ of the tuples arriving on S_2, have value v in attribute A.* □

Assuming an average rate r_2 of arrivals per unit time on S_2, the expected number of S_1-probe join tuples that a tuple $s \in S_1$ produces is given by:

$$E[n_1(s)] \quad = \quad r_2 \cdot W_1 \cdot f_2(s.A) \qquad (1)$$

Example Scenario: Suppose we are monitoring a system with a fixed number of *components*. We have a stream of *actions* and a stream of *errors* on all components, and we want to perform a sliding-window join on `component-id` to look for possible correlations between actions and errors. Some components may be more heavily used than others, and some may be more error-prone than others, but each `component-id` may have a roughly fixed frequency of occurrence on each stream.

1. When a new tuple s arrives on S_1
2. Update $S_2[W_2]$ by discarding expired tuples
3. Emit $s \bowtie_A S_2[W_2]$
4. Add s to $S_1[W_1]$

Figure 3: Sliding-window join execution

2.2 Age-Based Stream Model

For many applications, the frequency-based model is inappropriate. As a simple example, consider online auction monitoring [16] with the following streams:

S_1: `OpenAuction(auction-id,seller-id)`
S_2: `Bid(auction-id,bid-amount)`

When a seller starts an auction, a tuple arrives on S_1. When a bid is placed on an auction, a tuple arrives on S_2. Suppose we are interested in knowing, for each seller, the average number of bids received on all of his auctions in the last 5 days. This query requires a sliding-window join between S_1 and S_2 with a window on S_1 equal to the maximum lifetime of an auction, followed by an aggregation operator with a 5-day window.

Suppose memory is insufficient to retain all the tuples in S_1's window, and suppose we use the frequency-based model for making load-shedding decisions in this scenario. Auction-ids are unique, so on stream S_1 we see each auction-id only once. On stream S_2, the arriving auction-ids are the currently open auctions, so this set changes over time. Thus, no fixed frequency distribution can be inferred through monitoring. In this case, load-shedding schemes based on the frequency model [7] will simply retain new tuples and discard old ones. However, that is exactly the wrong thing to do, since most bids are received on auctions that are about to close, i.e., are relatively old. To capture such scenarios, we propose a new *age-based* model defined as follows:

Definition 2.2 (Age-Based Model). *For a tuple $s \in S_1$, the S_1-probe join tuples produced by s obey the following two conditions:*

1. *The number of S_1-probe join tuples produced by s is a constant independent of s, and is denoted by n_1.*
2. *Out of the n_1 S_1-probe join tuples of s, $p_1(k)$ are produced when s is between age $k-1$ and k.*

A symmetric case holds for the S_2-probe join tuples produced by a tuple $s' \in S_2$. Define $C_i(k)$, $i = 1, 2$, as the cumulative number of S_i-probe join tuples that a tuple $s \in S_i$ produces by age k, i.e., $C_i(k) = \sum_{j=1}^{k} p_i(j)$. □

Thus, according to this model, the number of joins a tuple produces is independent of its join-attribute value, but is a function of the age of the tuple in the window. Assumption 1 in Definition 2.2 is not strictly necessary for our approach. However, in the scenarios we have considered, the set of join-attribute values changes over time. Thus, even if $n_i(s)$ depends on $s.A$ for a tuple $s \in S_i$, it would be difficult to infer this dependence by monitoring the stream.

Different Age Curves: Consider a curve that plots $p_i(k)$ against k; we call this the *age curve* for window $S_i[W_i]$. Intuitively, the age curve shows how likely a tuple in $S_i[W_i]$ is to produce join tuples, as it becomes older. Different applications that adhere to the age-based model may have very different age-curve shapes:

- *Increasing*: An example is the auction scenario described above. In a typical auction, relatively few bids are received in the beginning, followed by a large number of bids when the auction is about to close. Thus $p_1(k)$ is small for small k, and increases with k until the auction lifetime, after which it drops to 0.
- *Decreasing*: Consider a join between an *Orders* and a *Fulfillments* stream on `order-id`, with a window on the orders stream. Most parts of an order are fulfilled soon, but some may require backorder and are fulfilled later. Thus we expect to see a decreasing age curve.
- *Bell*: Consider a join between streams of readings from two different sensors, with a band-join condition on timestamp. This join may be used to discover correlations between readings from two different observation points taken at roughly the same time. In this case, the age curve is expected to be bell-shaped. The age k at which the peak of the age curve occurs will be determined by factors such as clock skew between the two sensors, and the difference in network latency from the sensors to the stream system. We perform an experiment of this form in Section 6.

2.3 Parameter Estimation

For using any of the models described above, the model parameters must be instantiated, i.e., we must determine the frequencies of occurrence $f_1(v)$ and $f_2(v)$ of values $v \in \mathcal{D}$ for the frequency-based model, and the age curves for the age-based model. We assume the standard technique of using the past to predict the future, so parameters are estimated by monitoring the streams. There is previous work on building histograms in an online fashion using small space [10, 12], which can be used to estimate the values of $f_1(v)$ and $f_2(v)$. For the age-based model, n_i, $i = 1, 2$, is estimated as the average number of S_i-probe join tuples that an S_i-tuple produces in its lifetime. Similarly, $p_i(k)$ is estimated as the average number of S_i-probe join tuples that an S_i-tuple produces between age $k-1$ and k.

We do not need to collect $p_i(k)$ for each time unit k, but can use a coarser time granularity. To accurately determine $p_i(k)$, we should execute the join with the full window $S_i[W_i]$ being retained. For now, we assume that we can allocate an extra chunk of "monitoring memory" that is circulated periodically to each window in turn to monitor its parameters accurately. If this memory is not available, $p_i(k)$ can be approximately estimated by retaining a small fraction of the tuples on S_i in $S_i[W_i]$ for their entire lifetime. Alternative schemes for approximately estimating the age curve when extra memory is not available is a topic of future work.

3 Max-Subset

Recall our basic algorithm for executing join $S_1[W_1] \bowtie_A S_2[W_2]$ shown in Figure 3. If memory is limited, we need to modify the algorithm in two ways. First, in Line 2, we update $S_1[W_1]$ in addition to $S_2[W_2]$ to free up memory occupied by expired tuples. More importantly, in Line 4, memory may be insufficient to add s to $S_1[W_1]$. In this case, we need to decide whether s is to be discarded or admitted into $S_1[W_1]$, and if it is to be admitted, which of the existing tuples is to be discarded. An algorithm that makes this decision is called a *load-shedding strategy* [4, 7, 17]. Due to load-shedding, only a fraction of the true result will actually be produced. We denote the fraction of the result tuples produced as *recall*.

$$Recall(t) = \frac{\text{Number of result tuples produced up to time } t}{\text{Number of actual result tuples up to time } t}$$

Definition 3.1 (Max-Subset Problem). *Given a fixed amount of memory for a sliding-window join $S_1[W_1] \bowtie_A S_2[W_2]$, devise an online load-shedding strategy that maximizes $\lim_{t\to\infty} Recall(t)$.* □

We first state a result on the hardness of the problem for arbitrary streams (Section 3.1), then present a load-shedding strategy for the age-based model (Section 3.2), and finally discuss the max-subset problem for the frequency-based model (Section 3.3).

3.1 Hardness Result

A load-shedding strategy is *optimal* if it eventually produces the maximum recall among all strategies using the same amount of memory. For bounded streams, an *offline* strategy is one that is allowed to make its load-shedding decisions after knowing all the tuples that are going to arrive in the future. We show that for arbitrary streams, it is not possible for any online strategy to be *competitive* with the optimal offline strategy.

Let S denote a bounded sequence of tuple arrivals on the streams S_1 and S_2. Consider any online strategy. Let $R_{on}(M, S)$ denote the recall obtained at the end of executing the online strategy with memory M on the sequence S. Similarly, let $R_{off}(M, S)$ denote the recall for the optimal offline strategy. We assume M is insufficient to retain the entire windows. The online strategy is *k-competitive* if for any sequence S, $R_{off}(M, S)/R_{on}(M, S) \leq k$.

Theorem 3.2. *For the max-subset problem, no online strategy (even randomized) can be k-competitive for any k that is independent of the length of the input sequence.* □

A detailed proof is omitted due to space constraints. The idea is to construct an input distribution and to lower-bound the expected competitive ratio of any deterministic strategy on that input distribution. We then obtain Theorem 3.2 by applying Yao's min-max theorem [21].

This result shows that we cannot expect to find an effective load-shedding strategy that addresses the max-subset problem for arbitrary streams.

3.2 Age-Based Model

Consider the max-subset problem for the age-based model. For presentation, we first assume a fixed amount of memory is available for $S_1[W_1]$, and consider the problem of maximizing the number of S_1-probe join tuples produced. A symmetric procedure applies for S_2-probe join tuples given a fixed memory for $S_2[W_2]$. Then we show how to allocate the overall available memory between $S_1[W_1]$ and $S_2[W_2]$ to maximize the recall of the entire join.

3.2.1 Fixed Memory for $S_1[W_1]$

Suppose the available memory for $S_1[W_1]$ is sufficient to store M_1 tuples of stream S_1. We denote the amount of memory required to store one tuple as a "cell". For now we assume $r_1 = 1$, i.e., one tuple arrives on S_1 at each time step. At the end of the section we show the easy generalization to other r_1.[1] We first give the optimal strategy for $M_1 = 1$, which forms the building block for our strategy for $M_1 > 1$. Recall that $C_1(k)$ denotes the total number of S_1-probe join tuples that a tuple $s \in S_1$ produces by age k. Let k_1^{opt} denote the k ($\leq W_1$) at which $\frac{C_1(k)}{k}$ is maximized.

Strategy 1 ($M_1 = 1$). *Retain the first tuple $s \in S_1$ in $S_1[W_1]$ for k_1^{opt} time units, discarding other tuple arrivals on S_1. After k_1^{opt} time units, discard s, retain the tuple arriving next for the next k_1^{opt} time units, and continue.* □

The relatively straightforward proof that Strategy 1 is optimal is omitted due to space constraints.

Example 3.3. *Let $r_1 = 1$ and $M_1 = 1$ as we have assumed so far. Let the window size $W_1 = 4$, and let the age curve be defined by $p_1(1) = 1, p_1(2) = 1, p_1(3) = 2, p_1(4) = 1$. $C_1(k)/k$ is maximized at $k_1^{opt} = 3$.*

Let s_i denote the tuple arriving at time i on S_1. The following diagram illustrates Strategy 1 on this example. Entries in the third row denote the number of S_1-probe join tuples produced between each time step and the next.

Time	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Cell 1	s_1	s_1	s_1	s_4	s_4	s_4	s_7	...
Discard		s_2	s_3	s_1	s_5	s_6	s_4	...
# Results	*1*	*1*	*2*	*1*	*1*	*2*	*1*	...

Strategy 1 produces 4 join tuples every 3 time units and is optimal among all possible strategies. □

[1] Note that all of our optimality claims assume constant rather than average r_1, however our experiments (Section 6) show that our algorithm performs well for a distribution of arrival rates.

Now suppose $M_1 > 1$. We must consider two cases:

1. If $k_1^{opt} \geq M_1$, the optimal strategy is to run Strategy 1 "staggered", for each of the M_1 cells. For example, if $M_1 = 2$ in Example 3.3, we get:

Time	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Cell 1	s_1	s_1	s_1	s_4	s_4	s_4	s_7	...
Cell 2		s_2	s_2	s_2	s_5	s_5	s_5	...
Discard			s_3	s_1	s_2	s_6	s_4	...

2. If $k_1^{opt} < M_1$, the problem becomes more complex because running a staggered Strategy 1 uses only k_1^{opt} cells, thereby underutilizing the available memory.

To address Case 2 ($k_1^{opt} < M_1$), we first define an age curve with a *minima*. The age curve $p_1(k)$ against k has a minima if there exist $k_1 < k_2 < k_3$ such that $p_1(k_1) > p_1(k_2)$ and $p_1(k_2) < p_1(k_3)$.

If the age curve has no minima, the optimal strategy is to retain every tuple for exactly M_1 time units. Once a tuple has been retained for k_1^{opt} time units, retaining it any further becomes less useful, and since the curve has no minima the tuple cannot become more useful in the future. Thus, it should be discarded as early as possible after k_1^{opt} time units. At the same time, tuples should not be discarded any earlier than M_1 time units, as that would lead to underutilization of memory.

If the age curve has a minima, retaining each tuple for exactly M_1 time units may be suboptimal. We illustrate the subtleties through an example.

Example 3.4. *Let $W_1 = 3$ and $M_1 = 2$. Let the age curve be defined by $p_1(1) = 3, p_1(2) = 0$, and $p_1(3) = 2$. Thus, the age curve has a minima at $k = 2$. We have $k_1^{opt} = 1$, so $k_1^{opt} < M_1$. The following strategy alternates between retaining every tuple for 1 and 3 time units, and by exhaustive search is seen to be optimal for this example:*

Time	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Cell 1	s_1	s_1	s_1	s_4	s_5	s_5	s_5	...
Cell 2		s_2	s_3	s_3	s_3	s_6	s_7	...
Discard			s_2	s_1	s_4	s_3	s_6	...
# Results	*3*	*3*	*5*	*3*	*5*	*3*	*5*	...

This strategy produces an average of 4 join tuples per time unit. Note that retaining every tuple for $M_1 = 2$ time units produces only 3 join tuples per time unit. □

We do not have an optimal strategy for the general case of age curves with minima, but in practice, age curves are unlikely to have minima (e.g., none of the examples discussed in Section 2.2 have minima). However, for completeness, we give the following greedy heuristic for this case. For each tuple $s \in S_1[W_1]$, assign a *priority* that represents the fastest rate at which s can produce S_1-probe join tuples. The priority of a tuple at age i is given by:

$$Priority(i) \quad = \quad \max_{i<j\leq W_1} \frac{C_1(j) - C_1(i)}{j - i}$$

When a tuple needs to be discarded due to a memory constraint, the tuple with the lowest priority is discarded.

This greedy strategy leads to the optimal solution for Example 3.4. Interestingly, this strategy reduces to the optimal strategy for all the previous cases as well. In the rest of this paper, we do not consider age curves with minima.

We shall refer to the overall approach for the age-based max-subset problem presented in this section as the *AGE* algorithm. We evaluate *AGE* experimentally in Section 6.

3.2.2 Fixed Memory for $S_1[W_1] + S_2[W_2]$

So far we have addressed the problem of maximizing the number of S_i-probe join tuples, $i = 1, 2$, given a fixed amount of memory for $S_i[W_i]$. Now suppose we have a fixed amount of memory M for the entire join. To determine how to allocate the available memory between $S_1[W_1]$ and $S_2[W_2]$, we need a function that relates the memory allocation to the overall recall obtained. Let M_i be the memory allocated to $S_i[W_i]$. Let R_i denote the rate at which S_i-probe join tuples are produced. If the *AGE* algorithm from Section 3.2.1 is applied:

$$R_i = \begin{cases} M_i \frac{C_i(k_i^{opt})}{k_i^{opt}} & \text{if } M_i \leq k_i^{opt} \\ C_i(M_i) & \text{if } M_i > k_i^{opt} \end{cases} \qquad (2)$$

Then the overall recall of the join is given by $\frac{R_1+R_2}{n_1+n_2}$. To determine the memory allocation between $S_1[W_1]$ and $S_2[W_2]$, we simply find M_1 and M_2 such that this expression for the recall of the join is maximized, subject to the constraint $M_1 + M_2 = M$.

Finally, so far we have assumed $r_i = 1$. If $r_i > 1$, and memory M_i is available for $S_i[W_i]$, Equation 2 becomes:

$$R_i = \begin{cases} M_i \cdot \frac{C_i(k_i^{opt})}{k_i^{opt}} & \text{if } M_i/r_i \leq k_i^{opt} \\ r_i \cdot C_i(M_i/r_i) & \text{if } M_i/r_i > k_i^{opt} \end{cases} \qquad (3)$$

The recall for the entire join is then given by $\frac{R_1+R_2}{r_1 n_1 + r_2 n_2}$.

3.3 Frequency-Based Model

We briefly consider the max-subset problem for the frequency-based model as covered in [7]. We derive the recall obtained given a fixed amount of memory for the join, This relationship between memory and recall is needed in Section 5 for overall memory allocation across joins.

Consider S_1-probe join tuples first. Recall Definition 2.1 of the frequency-based model. The following approach, called *PROB*, is suggested in [7]: Every tuple $s_1 \in S_1[W_1]$ is assigned a priority equal to $f_2(s_1.A)$. If a tuple needs to be discarded due to a memory constraint, the tuple with the lowest priority is discarded.

Without loss of generality, assume the values in $\mathcal{D}$ are $v_1, \ldots, v_n$, and for $i < j$, $f_2(v_i) \geq f_2(v_j)$. Then for $i < j$, *PROB* will prefer to retain all instances of v_i in $S_1[W_1]$ over any instance of v_j. Let M_1 be the memory allocated to $S_1[W_1]$. *PROB* will retain all instances of $v_1, v_2, \ldots, v_i$, where i is the largest number such that $r_1 W_1 \sum_{j=1}^{i} f_1(v_j) \leq M_1$. (A fraction of the instances of v_{i+1} will be retained too, but our analysis is not affected significantly.) Thus, S_1-probe result tuples are produced at a rate given by $R_1 = r_1 r_2 W_1 \sum_{j=1}^{i} f_1(v_j) f_2(v_j)$. A symmetric expression can be derived for the rate R_2 at which the S_2-probe join tuples are produced, given memory M_2 for $S_2[W_2]$. The overall recall of the join is then given by $\frac{R_1+R_2}{r_1 r_2 (W_1+W_2) \sum_{v \in \mathcal{D}} f_1(v) f_2(v)}$. Thus, given a total amount of memory M for the join, we can find M_1 and M_2 such that the overall recall of the join is maximized, subject to the constraint $M_1 + M_2 = M$.

4 Random Sampling

In this section, we address the problem of extracting a random sample of the $S_1[W_1] \bowtie_A S_2[W_2]$ join result with limited memory. We first state a result on the hardness of performing uniform random sampling on the join result for arbitrary streams (Section 4.1). We then give an algorithm for uniform random sampling that applies for both the age-based and frequency-based models (Section 4.2). Finally, in Section 4.3, we consider the case when a uniform sample is not required directly by the application, but is being gathered only for estimating an aggregate over the join result. For these cases, we consider a statistically weaker form of sampling called *cluster sampling* [6], which can be performed more easily than uniform sampling, and often yields a more accurate estimate of the aggregate.

4.1 Hardness Result

For sampling over the windowed join result of arbitrary streams, we have the following negative result:

Theorem 4.1. *If the available memory is insufficient to retain the entire windows, it is not possible to guarantee a uniform random sample for any sampling fraction* > 0. □

A detailed proof is omitted due to space constraints but the basic idea is as follows. Suppose we choose to discard a tuple s in $S_1[W_1]$ because memory is full. Then we must know that all S_1-probe join tuples that s would subsequently produce are guaranteed not to be needed in our sample. However, for arbitrary streams, at any time during the lifetime of s, there is no upper bound on the number of S_1-probe join tuples that s will produce before expiry. Thus, for any sampling fraction greater than 0, it cannot be guaranteed that we can discard s but preserve the sample.

This result shows that we cannot expect to find an effective procedure that performs uniform random sampling over the join result of arbitrary streams with limited memory. However, we can compute a sample when we have a model of stream arrivals, as we show next.

4.2 Uniform Random Sampling

For random sampling we can consider the frequency-based and the age-based models together. We shall assume *Bernoulli sampling*, or sampling under the coin-flip semantics [5]: for sampling a fraction p from a set of tuples, every tuple in the set is included in the sample with probability p independent of every other tuple.

s_1 : Tuple arriving on S_1
$n_1(s_1)$: Number of S_1-probe join tuples that s_1 produces
p : Sampling fraction

DecideNextJoin(s_1):
1. pick $X \sim \mathcal{G}(p)$
2. $s_1.next = s_1.num + X$
3. **if** $(s_1.next > n_1(s_1))$
4. discard s_1

Join(s_1, s_2):
1. $s_1.num = s_1.num + 1$
2. **if** $(s_1.num = s_1.next)$
3. output $s_1 \bowtie_A s_2$
4. DecideNextJoin(s_1)

Figure 4: Algorithm *UNIFORM*

4.2.1 Sampling Algorithm

Our algorithm *UNIFORM* for uniform random sampling over a sliding-window join with limited memory is shown in Figure 4. We only show the procedure for sampling from the S_1-probe join tuples by selectively retaining tuples in $S_1[W_1]$. The procedure for sampling from the S_2-probe join tuples is analogous. *UNIFORM* needs to know, for each arriving tuple $s_1 \in S_1$, the number of S_1-probe join tuples that s_1 will produce, i.e., $n_1(s_1)$. For the age-based model, $n_1(s_1) = n_1$. For the frequency-based model $n_1(s_1) = r_2 \cdot W_1 \cdot f_2(s_1.A)$ (recall Equation 1). We assume the sampling fraction p is known for now. In the next subsection, we show how p can be determined based on the amount of memory available.

When a tuple s_1 arrives on S_1, $s_1.num$ is initialized to 0, and the procedure *DecideNextJoin*(s_1) is called. *Join*(s_1, s_2) is called when a tuple s_2, that joins with s_1, arrives on S_2. $\mathcal{G}(p)$ denotes the geometric distribution with parameter p [15], and $X \sim \mathcal{G}(p)$ denotes that we pick X at random from $\mathcal{G}(p)$. When *DecideNextJoin*(s_1) is called, *UNIFORM* logically flips coins with bias p for deciding the next S_1-probe join tuple of s_1 that will be included in the sample. If all remaining S_1-probe join tuples of s_1 are rejected by the coin flips, s_1 is discarded.

4.2.2 Determining the Sampling Fraction p

To determine the sampling fraction p, we first obtain the expected memory usage of *UNIFORM* (i.e., the expected number of tuples retained) in terms of p. We then equate this expected memory usage to the amount of memory available for performing the join and solve for p. For robustness, we can also calculate the variance of the memory usage of *UNIFORM* and decide the sampling fraction such that the probability of the memory usage exceeding the available memory is sufficiently small. The following results about the expected memory usage follow from simple properties of the geometric distribution; proofs are omitted. Note that now the tuple size must include the space required to store the extra fields *next* and *num* (Figure 4).

Frequency-Based Model: Recall Definition 2.1. We assume that the S_1-probe join tuples of a tuple $s_1 \in S_1$ are produced uniformly throughout the lifetime of s_1 (because a uniform fixed fraction of tuples arriving on S_2 join with s_1).

Theorem 4.2. *For the frequency-based model, the expected memory usage of $S_1[W_1]$ is (let $q = 1 - p$):*

$$r_1 W_1 \sum_{v \in \mathcal{D}} f_1(v) \left(1 - \frac{q(1 - q^{r_2 W_1 f_2(v)})}{p r_2 W_1 f_2(v)}\right) \qquad \square$$

Age-Based Model: Recall Definition 2.2. Recall that $C_1(k)$ denotes the cumulative number of S_1-probe join tuples that a tuple $s_1 \in S_1$ produces by age k. Define the inverse of the C_1 function, $C_1^{-1}(m)$, as the smallest k such that $C_1(k) \geq m$. Thus, a tuple $s_1 \in S_1$ produces m S_1-probe join tuples by the time its age is $C^{-1}(m)$.

Theorem 4.3. *For the age-based model, the expected memory usage of $S_1[W_1]$ is $r_1 \sum_{i=1}^{n_1} p(1-p)^{n_1 - i} C^{-1}(i)$.* $\square$

In both models, a symmetric expression holds for the expected memory usage of $S_2[W_2]$, assuming we use the same sampling fraction p for the S_2-probe join tuples. Summing these expressions gives us the total memory usage for the join $S_1[W_1] \bowtie_A S_2[W_2]$.

4.3 Cluster Sampling

The correctness of *UNIFORM* depends heavily on the accuracy with which $n_i(s)$ is estimated for a tuple $s \in S_i$, $i = 1, 2$. For example, for a tuple $s_1 \in S_1$, if $n_1(s_1)$ is underestimated as $n_1'(s_1)$, then all the S_1-probe join tuples of s_1 subsequent to its first $n_1'(s_1)$ join tuples will never be selected for the sample. On the other hand, if $n_1(s_1)$ is overestimated, s_1 may remain in $S_1[W_1]$ until expiry, waiting for joins that never take place, and the overall memory usage may be considerably higher than the expected value derived in Theorems 4.2 and 4.3.

If a uniform random sample of the join is not required directly by the application, but the sample is being taken only to estimate an aggregate over the join results, these difficulties can be overcome by using a statistically weaker form of sampling called *cluster sampling* [6].

In general, cluster sampling is applicable when the population to be sampled can be divided into groups, or *clusters*, such that the cost of sampling a single element of a cluster is equal to that of sampling the entire cluster. Thus, for cluster sampling, a certain number of clusters are chosen at random, and all elements of selected clusters are included in the *cluster sample*. A cluster sample is *unbiased*, i.e., each element of the population has equal probability of being included in the sample. However, it is *correlated*, i.e., the inclusion of tuples is not independent of each other as in a uniform sample. A detailed analysis of cluster sampling can be found in [6]. In the remainder of this section we assume the sample of the join is being gathered for estimating either a sum or an average aggregate, and the objective is to minimize the error in the estimated aggregate.

4.3.1 Two Approaches

Consider sampling from the S_1-probe join tuples; a symmetric procedure applies for sampling from the S_2-probe join tuples. A tuple $s_1 \in S_1$ joins with $n_1(s_1)$ tuples arriving on S_2. These join tuples form a cluster, and the entire

cluster can be sampled by simply retaining s_1 in $S_1[W_1]$ until expiry. The fraction of clusters that can be sampled is determined by the number of tuples that can be retained until expiry in the memory available for $S_1[W_1]$. Thus we have the following naïve approach to cluster sampling.

Strategy 2 (*EQ-CLUSTER*). *Add an incoming tuple $s_1 \in S_1$ to $S_1[W_1]$ with probability p. If s_1 is added to $S_1[W_1]$, retain it until expiry and include all its S_1-probe join tuples in the sample.* □

Notice that this scheme does not depend on $n_1(s_1)$, and always produces an unbiased sample. The expected memory usage for $S_1[W_1]$ according to this scheme is $r_1 W_1 p$. Thus, p can be decided based on the amount of memory available.

EQ-CLUSTER is suitable when the clusters are roughly of equal size (e.g., as in the age-based model). However, if clusters are of unequal sizes, as in the frequency-based model, statistics literature [6] suggests that better estimates of the aggregate can be obtained by selecting a cluster with probability proportional to its size. Otherwise, if clusters are selected with equal probability, large clusters that contribute most to the aggregate may be missed altogether. We thus have the following approach:

Strategy 3 (*PPS-CLUSTER*). *Add an incoming tuple $s_1 \in S_1$ to $S_1[W_1]$ with probability proportional to $n_1(s_1)$. If s_1 is added to $S_1[W_1]$, retain it until expiry and include all its S_1-probe join tuples in the sample.* □

With *PPS-CLUSTER*, to get an unbiased estimate of the aggregate, we must perform weighted aggregation on the cluster sample: the contribution of each cluster to the aggregate is assigned a weight inversely proportional to the cluster size. Details can be found in [6]. Notice that even if $n_1(s_1)$ is incorrectly estimated, the same incorrect estimate is used in performing weighted aggregation. Hence, the resulting estimate of the aggregate is still unbiased.

Consider the application of *PPS-CLUSTER* for the frequency-based model. Since $n_1(s_1) \propto f_2(s_1.A)$, let s_1 be added to $S_1[W_1]$ with probability $p \cdot f_2(s_1.A)$ where p is a proportionality constant. The expected memory usage of $S_1[W_1]$ is $r_1 W_1 p \sum_{v \in \mathcal{D}} f_1(v) f_2(v)$. Thus, p can be determined according to the amount of memory available.[2]

4.3.2 Comparison of Approaches

To summarize, let us briefly consider which sampling approach is preferable in different scenarios. Recall that the objective is to minimize the error in an estimated aggregate. The relevant factors to be considered are:

- *Accuracy of model parameters*: If $n_i(s)$ is incorrectly estimated for a tuple $s \in S_i$, $i = 1, 2$, *UNIFORM* may perform poorly since it may produce a biased sample. In this case, cluster sampling should be used.
- *Inter-cluster variance*: Consider the variance in the values of the aggregate for different clusters. The lower this variance, the better the performance of cluster sampling compared to uniform sampling [6].
- *Cluster sizes*: *PPS-CLUSTER* should be used for unequal-size clusters. *PPS-CLUSTER* reduces to *EQ-CLUSTER* for equal-size clusters.

[2] A value of p obtained in this way can cause $pf_2(s_1.A)$ to exceed 1 for some s_1, resulting in an overestimate of memory usage. The correct value of p can be chosen by an iterative procedure; details are omitted.

5 Memory Allocation across Multiple Joins

Now suppose our stream system is executing a number of continuous queries, each of which involves a sliding-window join. In this section, we address the problem of allocating the available memory across these multiple joins. For now, let us assume the unweighted case, i.e., all joins are equally important. The goal of our allocation scheme is to ensure that no join does "too badly" in terms of approximation error, i.e., we seek to minimize the maximum approximation error in any join. It is important to observe that different joins may differ in the accuracy of their approximation even when given the same fraction of their memory requirement. Thus, simple proportional allocation of memory among the joins is generally not sufficient.

Suppose there are n sliding-window joins with an overall memory constraint M. Each join has a certain *approximation metric* which we denote by $\mathcal{Q}$: For the max-subset problem, $\mathcal{Q}$ is the recall of the join. If the output of the join is being aggregated, $\mathcal{Q}$ is the error in the estimated aggregate. We assume that each join uses the same approximation metric (i.e., either recall or aggregation error), otherwise the choice of a combined approximation metric is not clear. We shall focus on the case when $\mathcal{Q}$ is recall. A similar technique applies when $\mathcal{Q}$ is aggregation error.

For a particular memory allocation, let q_i be the recall obtained for the ith join. The optimal memory allocation we seek is the one that maximizes $\min_{1 \leq i \leq n} q_i$. The key to our scheme is the following observation (a similar observation is made in [4]).

Theorem 5.1. *To maximize the minimum recall, the optimal memory allocation is one that produces the same recall in all joins.*

By Theorem 5.1, in the optimal memory allocation the recall obtained in each join is the same, say q_{opt}. Let $f_i(q)$ denote the minimum amount of memory required to obtain recall q in the ith join. Then q_{opt} is the maximum q such that $\sum_{i=1}^{n} f_i(q) \leq M$. Assuming the functions f_i are known, q_{opt} can be found by an iterative binary search. The amount of memory to be allocated to the ith join is then given by $f_i(q_{opt})$.

Recall that we specified the relationship between the amount of memory available for a join and the recall that can be obtained, both for the age-based (Section 3.2.2) and the frequency-based (Section 3.3) models. These formulae can be used to calculate $f_i(q)$. When the metric $\mathcal{Q}$ is aggregation error, we can use the relationship between the available memory and the fraction that can be sampled (Theorems 4.2 and 4.3). The expected aggregation error

for a given sampling fraction can be derived in terms of population characteristics such as mean and variance [4]. Together, these can be used to calculate $f_i(q)$.

Finally, suppose that different joins have different relative importance. Let w_i be the weight of the ith join. Now our objective is to maximize $\min_{1 \leq i \leq n} q_i/w_i$. Our argument extends to show that the optimal solution is to allocate memory $f_i(w_i q_{opt})$ to the ith join, where q_{opt} is the maximum q such that $\sum_{i=1}^{n} f_i(w_i q) \leq M$.

We shall refer to the approach for memory allocation presented in this section as *ALLOC*, and evaluate its performance experimentally in Section 6.

6 Experiments

We now present an experimental evaluation of our techniques. Our experiments demonstrate the following:

1. In a real-life scenario that adheres to the age-based model, our algorithm *AGE* (Section 3.2.1) gives considerably higher recall than more naïve approaches.
2. Our sampling approaches *UNIFORM* and *PPS-CLUSTER* (Section 4) provide low-error estimates of windowed aggregates over the join result. Either of the two approaches may be preferable, depending on the specific scenario.
3. Our algorithm *ALLOC* for memory allocation across joins (Section 5) significantly outperforms simple proportional allocation in terms of maximizing the minimum recall.

6.1 Age-Based Experiment

For initial experimentation with the age-based model, we captured real data as follows. We set up two stream sources, ϕ_1 and ϕ_2, and a central server. Source ϕ_1 and the server run on the same physical machine, while source ϕ_2 runs on a geographically distant machine connected over a wide-area network (WAN). Each source produces tuples at a constant rate of $r_1 = r_2 = 50$ per second. Each tuple contains a timestamp `ts` from the local clock at the source. All tuples are streamed to the server using a connectionless UDP channel.

Denote the stream of tuples from sources ϕ_1 and ϕ_2 as S_1 and S_2 respectively. We execute a sliding-window join whose purpose is to identify causal correlation between the two streams—to do so, it matches tuples from S_2 with tuples from S_1 that were timestamped approximately one minute earlier. Specifically, the join predicate chosen is $S_2.\texttt{ts} - S_1.\texttt{ts} \in [59.9, 60.1]$ where time units are seconds. To ensure that S_1 tuples do not expire before matching S_2 tuples arrive (recall there may be significant network latency for S_2 tuples to arrive), we conservatively set the window on S_1 as $W_1 = 2$ minutes. Since joining tuples always arrive later on S_2 than on S_1, a window on S_2 need not be stored.

We generated a trace of approximately 40 minutes of activity , which we then used to perform repeatable experiments. Figure 5 shows the age curve ($p_1(k)$ vs. k) determined by an initial pass through our trace. We show $p_1(k)$

Figure 5: Age curve for WAN experiment

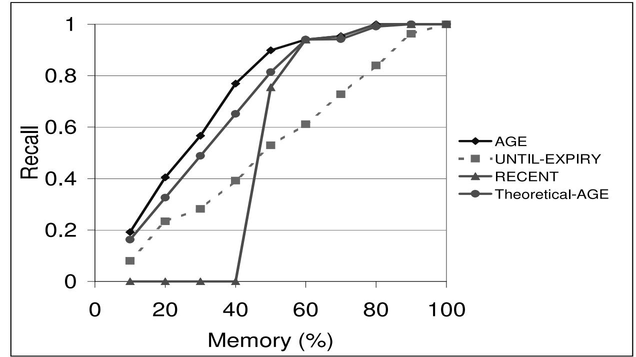

Figure 6: Recall obtained on WAN experiment

as a fraction after normalizing it with respect to n_1. The granularity chosen for k was 0.1 second. We see that a tuple $s \in S_1$ produces most join tuples at an age of approximately $k = 63$ seconds. Out of this, a 60 second delay is due to the join predicate, and the rest of the delay is due to clock skew between sources ϕ_1 and ϕ_2, and significantly higher network latency for tuples from ϕ_2 than from ϕ_1.

6.1.1 Results

Figure 6 shows the recall obtained on our trace by various possible load-shedding approaches as we vary the amount of allocated memory. Memory is shown as a percentage of the amount required to retain the entire window ($r_1 W_1$). The approaches being compared are (1) *AGE*: Section 3.2.1; (2) *UNTIL-EXPIRY*: A tuple is added to $S_1[W_1]$ only if memory is available, and then retained until expiry; (3) *RECENT*: The most recent tuples in the window are retained; and (4) Theoretical-*AGE*: The recall that should be theoretically obtained by applying the *AGE* approach, as given by Equation 3. Note that *RECENT* is the approach that we get if we simply apply the frequency-based model in this scenario.

Although in reality the age curve shown in Figure 5 has some minima, $p_1(k)$ never increases significantly after decreasing. Hence, for all practical purposes, we can apply our *AGE* approach assuming the curve has no minima. k_1^{opt} was calculated to be 68.8 seconds.

We see that *AGE* outperforms *RECENT* and *UNTIL-EXPIRY*. *RECENT* performs especially badly, producing no join tuples even when the allocated memory is as much

as 40%. However, when the allocated memory is high enough so that $M_1 \geq r_1 k_1^{opt}$, *AGE* reduces to *RECENT* (see Equation 3), and hence both approaches produce the same recall. Note that if W_1 had been conservatively set to be higher, the performance of *UNTIL-EXPIRY* would degrade, whereas the performance of *AGE* would not be affected. We also see that the actual recall obtained by *AGE* closely agrees with the theoretically predicted value.

6.2 Experiments on Synthetic Data

For the next set of experiments, we synthetically generate streams S_1 and S_2 for both the age-based and the frequency-based model, and perform the sliding-window join $S_1[W_1] \bowtie S_2[W_2]$ with limited memory. For simplicity, we consider only the S_1-probe join tuples in our experimental results. For both models, tuples on streams S_i, $i = 1, 2$, are generated at an average rate of r_i tuples per unit time. This is done by choosing the inter-arrival time uniformly at random between $1/2r_i$ and $2/r_i$ time units. For all experiments we fix $r_1 = 1, r_2 = 5$, and $W_1 = 500$.

6.2.1 Age-Based Data Generation

First stream S_1 is generated. Each tuple on S_1 contains a unique `id` which serves as the join attribute, emulating the example scenarios of Section 2.2 (e.g., in the auction scenario each tuple on S_1 has a unique `auction-id`). Next, we specify the age curve for $S_1[W_1]$ by dividing the window duration W_1 into m buckets and specifying $p_1(k)$ for the kth bucket. The first bucket consists of the newest tuples, and the mth bucket the oldest tuples. We fix $n_1 = 5$ and $m = 20$.

We then generate stream S_2. Suppose a tuple is to be generated on S_2 at time t. The value of its join attribute is determined as follows. We choose one of the m buckets at random with the kth bucket being chosen with probability $p_1(k)/n_1$. Then, we choose one tuple at random from all the tuples of $S_1[W_1]$ occupying the chosen bucket at time t. The `id` of this randomly-chosen tuple is assigned as the join-attribute value of the newly generated tuple on S_2.

6.2.2 Max-Subset

We experimented with three different age curves. (1) Increasing (*INC*): $p_1(k) \propto k^2$; (2) Decreasing (*DEC*): $p_1(k) \propto (m-k)^2$; and (3) Bell-shaped (*BELL*): $p_1(k) \propto k^2$ for $1 \leq k \leq m/2$ and $p_1(k) \propto (m-k)^2$ for $m/2 < k \leq m$. Figure 7 shows a comparison of the recall obtained by various approaches for different types of age curves. For the *INC* curve, *AGE* significantly outperforms *RECENT*. For the *DEC* curve, *AGE* reduces to *RECENT*, so we do not show their results separately. For the *BELL* curve, *AGE* outperforms *RECENT* until $M_1 < r_1 k_1^{opt}$ (see Equation 3). For $M_1 \geq r_1 k_1^{opt}$, *AGE* reduces to *RECENT*.

Note that for the same amount of allocated memory, the recall differs greatly depending on the shape of the age curve. This indicates that in the presence of multiple joins, in order to maximize the minimum recall, simple proportional memory allocation is not sufficient, which we verify empirically in Section 6.2.5.

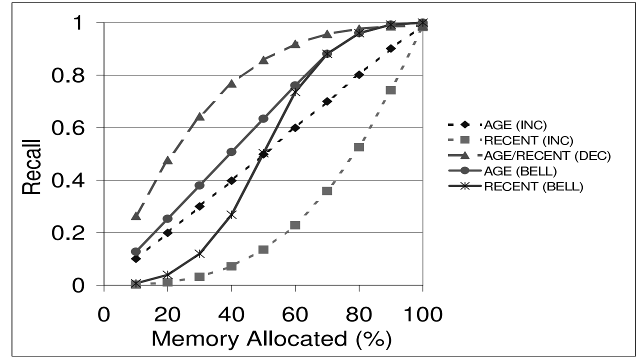

Figure 7: Recall obtained on synthetic age-based data

6.2.3 Frequency-Based Data Generation

Data generation for the frequency-based model is relatively easier than for the age-based model. We choose a domain $\mathcal{D}$. The domain size is fixed at $|\mathcal{D}| = 50$. For each stream, the join-attribute values are drawn from a Zipfian distribution of parameter Z over $\mathcal{D}$ [23]. The distribution used for both streams need not be the same. We consider three cases: (1) Directly Correlated (*DC*): The order of frequency of occurrence of values is the same for S_1 and S_2; (2) Inversely Correlated (*IC*): The order of frequency of occurrence of values for S_1 is opposite of that for S_2, i.e., the rarest value on S_1 is the most common on S_2 and vice-versa; and (3) Uncorrelated (*UC*): The order of frequency of occurrence of values for the two streams is uncorrelated.

6.2.4 Random Sampling

To study the performance of our sampling approaches, we perform a windowed average over the sampled result of $S_1[W_1] \bowtie S_2[W_2]$ and compare the approaches in terms of aggregation error. The aggregation window is fixed at $W_{aggr} = 500$. The aggregated attribute is part of either S_1 or S_2, and its values are drawn from a normal distribution having mean μ and variance σ. At each time step, the value of the windowed aggregate over the true result (U) and over the sampled result ($\hat{U}$) are calculated. The relative error in the aggregate is calculated as $|\hat{U} - U|/U$. We report the average of these errors over the entire run. In all experiments, while implementing *UNIFORM*, we assume a tuple size of 32 bytes. The two extra fields required (see Figure 4) are stored compactly in two bytes, thus giving a new tuple size of 34 bytes.

We first consider the case when the aggregated attribute is part of S_1. Recall that all the S_1-probe join tuples produced by a tuple $s \in S_1$ form a cluster. Thus, in this case, all tuples in a cluster have the same value in the aggregated attribute, which is the worst case for cluster sampling.

Effect of Allocated Memory: Figure 8 shows the aggregation error of the various sampling approaches as we vary the amount of allocated memory. We use the inversely correlated (*IC*) frequency-based model with $Z = 2$, and we fix $\mu = \sigma = 100$. We see that *PPS-CLUSTER* outperforms *EQ-CLUSTER*: in the *IC* case, there are a small number of large clusters in the result which may be missed

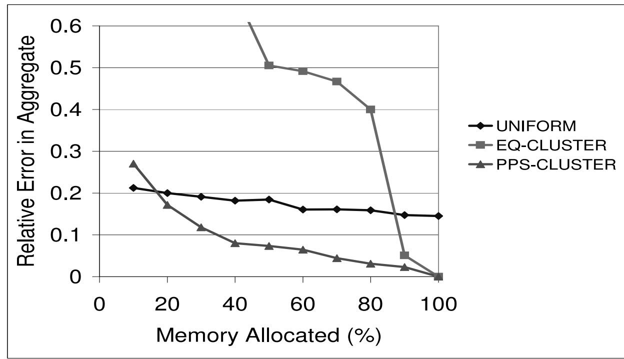

Figure 8: Aggregation error vs. memory allocated, *IC* frequency-based model, $Z = 2$, $\mu = \sigma = 100$

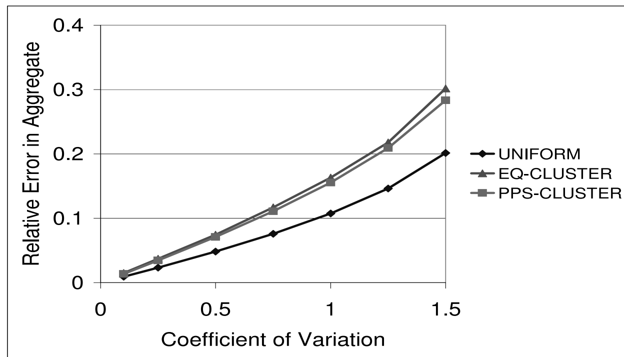

Figure 9: Aggregation error vs. population variance, *UC* frequency-based model, $Z = 2$, $\mu = 100$, Memory=10%

by *EQ-CLUSTER*. *UNIFORM* performs better than *PPS-CLUSTER* when the allocated memory is 10%. However, the fraction that can be sampled grows more rapidly for *PPS-CLUSTER* than for *UNIFORM*. Consequently, *PPS-CLUSTER* performs better at higher allocated memory. Note that the error of *UNIFORM* does not go down to 0 even when allocated memory is 100%. This is because even the synthetic data does not adhere perfectly to the model, as is required for the correctness of *UNIFORM* (Section 4.3).

Effect of Population Variance: Figure 9 shows the aggregation error of the various sampling approaches as the variance of the aggregated attribute is varied. We show the variance normalized by the mean, i.e., we show the coefficient of variation (σ/μ). The allocated memory is 10%, $\mu = 100$, and the model used is the uncorrelated (*UC*) frequency-based model with $Z = 2$. As the population variance increases, since all tuples in a cluster have the same value, the inter-cluster variance increases. As a result, the performance of cluster sampling approaches degrades as compared to *UNIFORM*.

If the aggregated attribute is a part of stream S_2, the values in a cluster are uncorrelated. Consequently, cluster sampling performs much better that *UNIFORM* since it produces a much bigger sample. We omit the results for this case due to lack of space. Finally, note that for comparing our sampling approaches, we have calculated the exact aggregate over the sampled result. In reality, when memory is limited, this aggregation may be approximated [8].

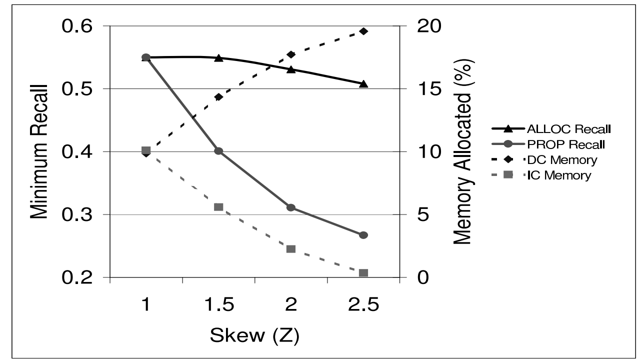

Figure 10: Memory allocation across joins: frequency-based model, Memory=20%

6.2.5 Memory Allocation across Multiple Joins

For memory allocation among multiple joins, we study the performance of our *ALLOC* scheme in comparison with simple proportional memory allocation (*PROP*). The metric for comparison is the minimum recall obtained in any join. We experimented with both the frequency-based and age-based models.

Frequency-Based Model: We allocate memory across two joins, where all parameters in the two joins are identical except one is the directly correlated (*DC*) case, and the other is the inversely correlated (*IC*) case (Recall Section 6.2.3). The total available memory is 20% of that required for executing both joins accurately. The load-shedding strategy used for each join is *PROB* [7]. Figure 10 shows a comparison of the minimum recall obtained by both approaches when we vary the skew parameter (Z) of the frequency-based model. As Z increases, the minimum recall remains almost constant for *ALLOC*, but decreases sharply for *PROP*. The amount of memory allocated to each join by *ALLOC* (as a percentage of the total memory required) is shown by the dashed plots on the secondary Y-axis. Note that *PROP* always splits the available memory evenly between the two joins, i.e., 10% to each join.

To understand these results, notice that the *IC* case is “easy”, i.e., a relatively higher recall can be produced using a small amount of memory: only the rare values of S_1 (which are frequent on S_2) need to be retained. In contrast, the *DC* case is “hard”, i.e., more memory is required to obtain the same recall because the common values on S_1 need to be retained. Moreover, as the skew (Z) increases, the *IC* case becomes easier, and the *DC* case becomes harder. *ALLOC* is able to outperform *PROP* by allocating less memory to the *IC* case, and using this extra memory to boost the performance of the *DC* case.

Age-Based Model: We again execute two joins, one with an increasing (*INC*) age curve, and another with a decreasing (*DEC*) one. The *INC* curve is chosen as $p_1(k) \propto k^p$ and the *DEC* curve as $p_1(k) \propto (m - k)^p$, where the exponent p is varied. The total available memory is 50% of that required for executing both joins accurately. The load-

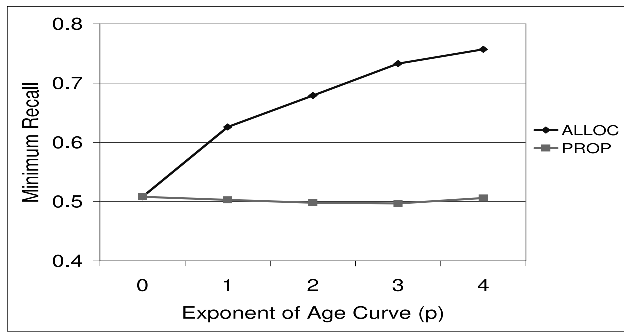

Figure 11: Memory allocation across joins: age-based model, Memory=50%

shedding strategy used for each join is *AGE* (Section 3.2.1). Figure 11 shows a comparison of the minimum recall obtained by both approaches when we vary the exponent p. As p increases, the minimum recall increases for *ALLOC* but remains constant for *PROP*. With increase in p, the *DEC* case becomes "easier", while the *INC* case remains equally "hard" (by Equation 3). Thus *ALLOC* is able to outperform *PROP* by allocating less memory to *DEC*, and using the extra memory to boost the performance of *INC*.

More Joins: We omit the results of experimenting with a greater number of joins, but the findings were similar: As more "hard" joins are added, the gain of *ALLOC* over *PROP* decreases, while if more "easy" joins are added, the gain of *ALLOC* over *PROP* increases. Intuitively, the performance of *PROP* is always limited by the hardest join, while *ALLOC* equalizes the recall among all joins.

7 Conclusion

In this paper we addressed memory-limited approximation of sliding-window joins. We defined a novel age-based model that often enables us to address the max-subset problem more effectively than the frequency-based model used previously. We introduced and addressed the problem of extracting a random sample of the join result with limited memory. Finally, we gave an optimal algorithm for memory allocation across joins to minimize the maximum approximation error.

One promising avenue for future work is to extend the approximation techniques developed here to address a related but distinct problem: memory-limited computation of exact answers. Now, instead of load-shedding we must store selected data on disk. The frequency-based and age-based models may help us develop algorithms that minimize disk I/O for this setting. A second interesting direction is to generalize our memory allocation strategy to handle a broader class of queries and plan operators. Finally, so far we have considered only the static version of the problem, where stream characteristics are assumed to be relatively stable. For volatile environments, we plan to develop adaptive versions of our algorithms.

Acknowledgements

We are grateful to Arvind Arasu, Rajeev Motwani, and the entire STREAM group at Stanford for useful discussions.

References

[1] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proc. of the 1999 ACM Symp. on Principles of Database Systems*, pages 10–20, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[3] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 633–634, 2002.

[4] B. Babcock, M. Datar, and R. Motwani. Load-shedding for aggregation queries over data streams. In *Proc. of the 2004 Intl. Conf. on Data Engineering*, 2004. To appear.

[5] S. Chaudhuri, R. Motwani, and V.R. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 1999.

[6] W. G. Cochran. *Sampling Techniques*. John Wiley & Sons, 1977.

[7] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[8] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[9] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 61–72, 2002.

[10] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. of the 2002 Annual ACM Symp. on Theory of Computing*, 2002.

[11] L. Golab and M. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[12] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 2001 Annual ACM Symp. on Theory of Computing*, pages 471–475, 2001.

[13] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003.

[14] S. Krishnamurthy et al. TelegraphCQ: An Architectural Status Report. *IEEE Data Engineering Bulletin*, 26(1):11–18, March 2003.

[15] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[16] SQR – A Stream Query Repository. http://www-db.stanford.edu/stream/sqr.

[17] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load-shedding in a data stream manager. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, September 2003.

[18] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, March 2003.

[19] T. Urhan and M.J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[20] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. Technical report, Duke University, Durham, North Carolina, November 2003.

[21] A. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. of the 1977 Annual IEEE Symp. on Foundations of Computer Science*, pages 222–227, 1977.

[22] S. Zdonik et al. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[23] G. E. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, Inc., 1949.