Load Shedding for Window Joins on Multiple Data Streams

Yan-Nei Law
Bioinformatics Institute
30 Biopolis Street, Singapore 138671
lawyn@bii.a-star.edu.sg

Carlo Zaniolo
Computer Science Dept., UCLA
Los Angeles, CA 90095, USA
zaniolo@cs.ucla.edu

Abstract

We consider the problem of semantic load shedding for continuous queries containing window joins on multiple data streams and propose a robust approach that is effective with the different semantic accuracy criteria that are required in different applications. In fact, our approach can be used to (i) maximize the number of output tuples produced by joins, and (ii) optimize the accuracy of complex aggregates estimates under uniform random sampling. We first consider the problem of computing maximal subsets of approximate window joins over multiple data streams. Previously proposed approaches are based on multiple pairwise joins and, in their load-shedding decisions, disregard the content of streams outside the joined pairs. To overcome these limitations, we optimize our load-shedding policy using various predictors of the productivity of each tuple in the window. To minimize processing costs, we use a fast-and-light sketching technique to estimate the productivity of the tuples. We then show that our method can be generalized to produce statistically accurate samples, as needed in, e.g., the computation of averages, quantiles, and stream mining queries. Tests performed on both synthetic and real-life data demonstrate that our method outperforms previous approaches, while requiring comparable amounts of time and space.

1. Introduction

There is growing research interest in supporting and processing efficiently high level queries on continuous data streams [2]. This interest is motivated by a growing number of applications, such as network monitoring [12, 14], security, sensor networks [4], web logs, click-streams, telecommunication data management [6] and many others. The UCLA Stream Mill project [11] is developing a data stream management system (DSMS) designed to support efficiently a wide range of applications and streaming data, including very advanced applications such as data stream mining [11, 10].

Many data stream applications require real-time, or near real-time, response and are characterized by very high arrival rates. Often it is not possible to provide exact real-time answers to all continuous queries currently running on the DSMS. Moreover, many data stream tasks do not actually require exact answers: they only require statistically accurate approximate answers. Typical examples of such tasks include stream mining queries, continuous aggregation queries, and queries involving sampling. For these queries, we want to provide useful answers that approximate the exact ones as best as possible given the available system resources and current (over)load conditions. To achieve this, the DSMS will apply load shedding techniques to lower-priority queries to reduce the system overload and return very useful approximate answers [15, 7, 13].

In this paper, we focus on the problem of semantic load shedding for continuous queries that contain window equijoins on multiple data streams. This problem is important since multijoins on data streams are among the most expensive operators, both in terms of computation cost and memory usage; thus, the availability of effective load shedding strategies for such operators is critical for a DSMS.

Since a fast response is required, the system must make load shedding decisions very fast, in order to optimize the expected outcome according to sensible approximation criteria. Two types of approximation that have been suggested are [13]: (1) Max-Subset results and (2) Sampled results. For (1), the objective is to maximize the size of the resulting join. For (2), the system must generate a random sample of the join result, since randomness is required to achieve accuracy in aggregate queries and stream mining queries computed on the results generated by such joins. We will address both problems in this paper.

Related Work. One semantic approach for making a load-shedding decision is based on the statistics or frequency distribution of the streams. In [7], Das et al. proposed an algorithm called *Prob* which keeps the tuples with most frequent value and an algorithm called *Life* which keep the tuples whose frequency multiplied by lifetime is highest, in order to produce the maximum subset of the exact result. In

[13], a solution was given to optimize the memory allocation across joins for producing a max-subset and a random sample of the result. However, both of them mainly focused on the binary join problem.

An important problem that deserves further attention is the problem of joining multiple streams. Recently there are many studies of multi-join processing in continuous queries. For instance, in [9], Golab et al. have analyzed multi-way join algorithms and developed a join order heuristics to minimize the processing cost per unit time. In [16], Viglas et al. have completed a prototype implementation of a multi-way join operator 'MJoin' which maximizes the output rate of the join queries. However, there is no focusing study of finding load shedding policies for computing multi-join. The closest discussion to this problem can be found in [7]. Das et al. mentioned a trivial m-approximation to the problem when the systems need to delete k_i tuples from join relation A_i, where $1 \leq i \leq m$. The idea is to independently choose k_i tuples from A_i for deletion which produce the fewest output tuples, says p_i. Then the optimal algorithm at least loses $\max p_1, \ldots, p_m$ and the approximation algorithm at most lose $\sum_{i=1}^{m} p_i$ tuples. Therefore, it is an m-approximation. However, the description was very sketchy. There are many problems such as how can we find the tuple productivity over the whole join, which have not been discussed in details. In fact, even the m-relation static (i.e. the content of the joined relations are known) join load shedding problem is not a trivial problem as it is NP-hard [7]. In this situation, load-shedding decision of one stream should depend on the result of the join of the remaining streams. The current approach to this problem is to use multiple binary join instead. For instance, users first convert a multi-join query to a multi-binary join query and use the idea of *Prob* to make load shedding decisions of each pair of streams. However, this cannot guarantee that the utility of the storage space is optimized because it disregards the content of streams outside the joined pair in load shedding decisions. We will show in the experiment section that the multiple binary join approach is not effective as the multi-join approach.

Paper Main Results and Outline. In this paper, we present novel techniques for approximating window joins; our techniques adapt quickly to changing data distribution and support joins on multiple data streams. The paper is organized as follows:

- In §2, we describe our model for computing approximate sliding window joins over multiple data streams.
- In §3, we propose two different tuple priority measures used in load shedding decisions for max-subset problem and random sampling problem, based on our analysis of the tuple expected gain (productivity). We then propose a method for estimating the productivity using a sketching technique.
- Our load shedding algorithm for computing approximate multi-join is then given in §4. It can achieve different objectives by using different priority measures.
- In §5, we compare the new algorithm on both synthetic and real-life data sets. The results of these extensive experiments, suggest that our newly proposed approach will outperform existing multi-binary join approach.
- Finally, §6 presents our conclusions and suggestions for future work.

2. Model

First we study a model of processing multi-way join operations. Let $S_1, \ldots, S_n$ be n streams and assume that the current average arrival rate is k tuples per second. The basic query we consider is a p-seconds sliding-window join among n streams defined by the conjunction of some equi-join constraints θ. Note that our method can be directly generalized to handle the case when every stream has different p_i-seconds sliding window. For simplicity, we restrict our discussion to the case of equal window for all streams, i.e. $p = p_i$ for all i. We denote $\mathbf{T} = W_1 \bowtie \ldots \bowtie W_n$, where W_i is the current window of S_i. We also consider windowed aggregation queries Σ over the join result.

There are two main parts in the model shown in Figure 1: a join operator and a queue. The join operator processes multi-way join and allocates a fixed amount of memory to each W_i to store the internal state. For simplicity, we assume that it can only process one tuple at every moment and the average join processing rate is l tuples per second. When a new tuple t_i of S_i arrives, first the expired tuples in all the windows are deleted. Then, the join result produced by t_i is computed and t_i is added into W_i.

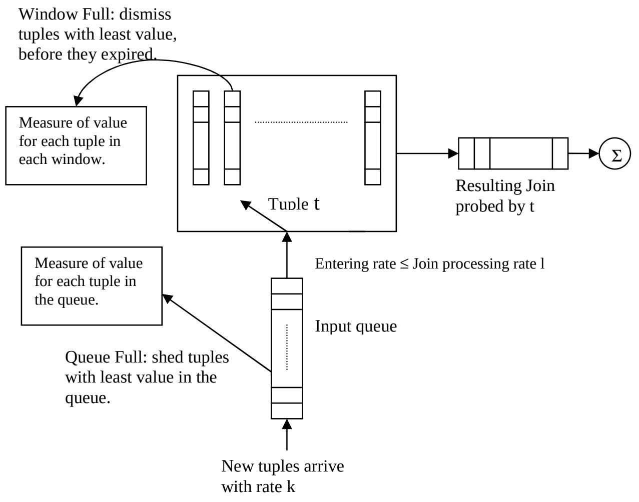

Window Full: dismiss tuples with least value, before they expired.
Measure of value for each tuple in each window.
Tuple t
Resulting Join probed by t
Σ
Entering rate ≤ Join processing rate l
Measure of value for each tuple in the queue.
Input queue
Queue Full: shed tuples with least value in the queue.
New tuples arrive with rate k

Figure 1. Model of Multiple-Join Operation.

The second part of the model is the queue. It is used to store the input tuples waiting for entrance. When the join operator is free, we push a tuple from the queue to the join operator for computing the join tuples produced by it.

If the available memory is smaller than $p \cdot \min(k, l)$, the system is not able to keep the entire window for providing the exact join result. If the buffer is full, we make a decision whether we keep the new tuple by dismissing one tuple in its buffer before it expires, or just drop the new tuple, according to priority of each tuple in the buffer. If a queue forms, it is soon filled to capacity. So, we need to make a load shedding decision to keep the tuples with highest priority in the queue. Note that load shedding decisions in the window and the queue need a priority measure. Therefore, in the following sections, we will analyze the expected tuple gain and propose a priority measure to optimize our decisions.

3. Load Shedding Decision

The productivity of a tuple determines its contribution to the multi-way join. In this section, we estimate the contribution to the multi-way join of each tuple in the window and in the queue. Then, we propose a shedding decision procedure for maximum subset or random sampling based on the tuple productivity measure so derived. For simplicity, we denote by $\mathbf{T}_{W_i=\{t\}}$ the join of all windows except W_i with the set $\{t\}$.

3.1. Productivity of Tuples

- **Window Tuples.** Consider a tuple $t \in S_i$. During its lifetime p seconds in W_i, there is a set of input tuples I_j arriving to each stream S_j for $j = 1, \ldots, i-1, i+1, \ldots, n$. For each I_j, its content can be estimated as W_j. Then for the arrival of I_j, t produces $|\mathbf{T}_{W_j=I_j, W_i=\{t\}}|$ tuples, which can be estimated as $|\mathbf{T}_{W_i=\{t\}}|$. Since there are $n-1$ sets of input tuples I_j, the total number of join tuples produced by t in W_i is $(n-1) \times |\mathbf{T}_{W_i=\{t\}}|$.

- **Tuples from Queues.** For a tuple $t \in S_i$ in the queue, at the moment that it reaches the join operator, there will be $|\mathbf{T}_{W_i=\{t\}}|$ join tuples produced. Moreover, when t is stored in window W_i, the expected number of join tuples that t produces is $(n-1) \times |\mathbf{T}_{W_i=\{t\}}|$.

Note that the productivity of a tuple $t \in W_i$ either in the windows and in the queue depends on the quantity $|\mathbf{T}_{W_i=\{t\}}|$. There are two observations: (1) the arrival rate of each stream except S_i is implicitly involved in this expression as the size of each window W_j is a product of the arrival rate of S_j and p; (2) the productivity of a tuple will not be affected by the arrival rate of its own stream. This assures fairness across different streams with very different arrival rates.

3.2. Two Priority Measures

Maximum Subset. To provide a maximum subset of the true result, we should shed the tuple with least productivity in order to minimize the loss caused by load shedding. Therefore, to maximize the output size of the approximate join result, we should pick the one such that $\arg\min_t |\mathbf{T}_{W_i=\{t\}}|$ for load shedding decision.

Random Sampling. To provide a random sample of the true result, one may control the fraction of the tuples produced by each tuple: For each tuple $t \in W_i$, we record the number of tuples that have been produced by t so far and estimate the number of tuples probed by t as $(n-1) \times |\mathbf{T}_{W_i=\{t\}}|$. Then the fraction of tuples that not yet produced by t can be obtained by

$$1 - \frac{|\text{tuples produced by } t|}{(n-1) \times |\mathbf{T}_{W_i=\{t\}}|}.$$

An efficient way to produce a statistically accurate random sample of the true result is to drop the tuple with the least percentage of tuples produced in its remaining lifetime when the window is full. An intuitive reason is that this can give equal opportunity for each tuple in the window to produce certain fraction of tuples and avoid the case that the memory will be dominated by the tuple with high productivity. For each tuple in the queue, its priority is always equal to 1. In this case, we can employ some advanced methods [5] to make load shedding decisions. However, for simplicity, when the queue is full, we randomly pick a tuple from the queue for load shedding.

To compute these two different priorities, we need to obtain the quantity $|\mathbf{T}_{W_i=\{t\}}|$, which is the cardinality of the multi-way join where $W_i = \{t\}$. However, the time and space for computing the exact cardinality of a join are very expensive [8]. Therefore, a significant amount of previous research has focused on the problem of estimating the cardinality of a join result [1, 8]. Next, we apply a sketching technique that is capable of estimating this measure for each tuple with minimal demands on computing resources.

Estimating Productivity. In [8], Dobra et al. extended the sketching techniques from [1] to find approximating complex query answers. The class of queries that they considered is of the form:

SELECT AGG FROM $R_1, \ldots, R_r$ WHERE θ

where AGG is an arbitrary aggregate operator such as COUNT, SUM or AVERAGE and θ represents the conjunction of equi-join conditions. The idea is to compute small sketch summaries of the streams that can then be used to provide approximate answers with approximation error guarantees. Its low time and space complexities make it practical for solving this kind of problems. However, no previous work further applied this method to query processing problems. And we will show that this can be effectively used to estimate tuple productivity.

First we discuss the key idea of this technique as follows: For each pair of join attributes $j \in \theta$, we build a family of four-wise independent random variables $\{\xi_{j,i} : i \text{ is an element in the domain of } j\}$ where each $\xi_{j,i} \in \{-1, +1\}$. For each relation R_k, its atomic sketch X_k for R_k is defined as

$$X_k = \sum_{i \in D} (f_k(i) \prod_{j \in R_k} \xi_{j,i}),$$

where $f_k(i)$ is the frequency of i in R_k and the COUNT estimator random variable is defined as $X = \prod X_k$. Note that each atomic sketch X_k can be efficiently computed when tuples of R_k are streaming in. First X_k is initialized to 0 and for each tuple t in R_k comes in, the quantity $\prod_{j \in R_k} \xi_{j,t[j]}$ is added to X_k, where $t[j]$ denotes the value of attribute j in tuple t. As shown in [8], the random variable X is an unbiased estimator for the result of COUNT query Q_{COUNT}, i.e. $E[X] = Q_{COUNT}$. To improve the accuracy, we may construct several random variable X. Indeed, the final estimate Y is chosen to be the median of s_2 random variables $Y_1, ...Y_{s_2}$, where each Y_i is the average of s_1 iid random variables X_{ij}, $1 \leq j \leq s_1$. Each X_{ij} is constructed identical to the construction of the random variable X above. Hence, the size for this sketch synopsis is

$$O(s_1 \times s_2 \times \sum_{A_i \in A} \log |(dom(A_i)|)$$

where A is the set of join attributes. As shown in [8], the values of s_1 and s_2 can be selected to achieve a prescribed degree of accuracy with high probability.

For our problem, our aim is to compute the cardinality of the multi-way join where the j-th stream window only contains t, i.e. $|\mathbf{T}_{W_j=\{t\}}|$. This quantity in fact is the answer of the query:

SELECT COUNT
FROM $W_1, \ldots, W_{j-1}, \{t\}, W_{j+1}, \ldots, W_n$ WHERE θ

and can be estimated by the sketching technique. This is the productivity of tuple t and denoted as $prod(t)$. Now we can use it to compute the priority of each tuple w.r.t. different objective as described in §3 for load shedding decisions.

4. Approximating Multi-Join Using Sketching Techniques

Our main algorithm *MSketch* is based on the multi-join model described in §2. There are two parts in the model: join operator and queue. They operate separately and simultaneously. The details of our algorithm are as follows:

```
Algorithm MSketch

0. Set up:
0.1 Assign a family of four-wise independent random variables $\xi$ as described in [8].
0.2 For each relation $R_k$, the tumbling sketch $X_k$ is initialized to 0.

1. Join Operator:
While the queue is non-empty{
1.1 Pull a tuple from the head of the queue. (Assume that $t_i$ is this input tuple and is from stream $S_i$.)
1.2 Update the current tumbling sketch $X_i$ of $R_i$ by
        $X_i = X_i + \prod_{j \in R_i} \xi_{j,t_i[j]}$.
    For each $R_j$, we expire its atomic sketch $X_j$ by replacing the atomic sketches of the last window $X_j^{last}$ to be the current one $X_j$ and resetting $X_j$ to be 0 for every $n$ seconds. Also, we reset all the priority queues to be empty.
1.3 Delete the expired tuples of every window.
1.4 Compute the join result produced by $t_i$ and compute $prod(t_i)$:
        $prod(t_i) = \prod_{j \in R_i} \xi_{j,t_i[j]} \times \prod_{k \neq i} X_k^{last}$
    to obtain the priority value based on our objective, then insert into the priority queue.
1.5 If $W_i$ is not full, store $t_i$ in $W_i$
    Else remove the tuple with lowest priority and in case of tie, pick arbitrarily.}

2. Queue: (for Max-Subset)
While a new tuple $t_i \in S_i$ arrives in the queue {
2.1 compute $prod(t_i)$:
        $prod(t_i) = \prod_{j \in R_i} \xi_{j,t_i[j]} \times \prod_{k \neq i} X_k^{last}$
    to obtain the priority value for Max-Subset, then insert into the priority queue.
2.2 If queue is not full, store $t$ in the queue
    Else remove the one with lowest priority and in case of tie, pick arbitrarily.}
```

Algorithm 1: *MSketch* for Join Operator and Queue.

The main idea of our algorithm is to make load shedding decisions using the priority measures proposed in §3. For estimating such measure, we compute an atomic sketch of each stream. To capture the up-to-date distribution, we use the atomic sketch of a recent tumbling window to estimate the productivity of a tuple. The idea of a tumbling window was introduced in [3] and its definition is as follows: For a predefined quantity n, tumbling window is a window containing tuples of every n seconds. It effectively partitions a stream into disjoint windows. Tumbling window is more practical than sliding window because sliding window requires extra memory buffer to store the data of the most current window, especially when the system is running out of memory. In our experiments, we set the size of the tumbling window to be the size of the join window, i.e. $n = p$. To compute each tuple productivity, we simply use the atomic sketch of the current window of each stream. However, the content of these atomic sketches change when a new tuple arrives. Therefore, we have to recompute the productivity of each tuple in the window for each new arrival and hence the time spent will be very large. Instead, we can use the atomic sketch of the last tumbling window to estimate the productivity of a tuple. This can save much computation time. To speed up the searching of a minimum, we employ a technique called priority queue to store the estimated productivity. Priority queue is a data structure which is preferable for extracting the minimum of a subset and for inserting a new element. When a new tuple arrives, we first update the atomic sketch of its current window. We then compute its priority and insert into the priority queue. If the window is full, we extract the minimum one from the priority queue and drop it from the window. For the case of the first window, since there is no information of the last tumbling window, we will use the current window instead. However, for simplicity, we will omit this case when we discuss our algorithm.

Complexity. To compute the exact size of the result for the multi-binary join approach mentioned in §1, we need to store the frequency of each value in the domain of join attributes. Therefore, the space required is $O(\sum_{A_i \in A} |(dom(A_i))|)$ where A is the set of join attributes. For our algorithm, the space required for storing $s_1 \times s_2$ groups of sketch synopsis is

$$O(s_1 \times s_2 \times \sum_{A_i \in A} \log |(dom(A_i))|).$$

Therefore, our algorithm needs much less space than the multi-binary join approach. Moreover, the number of atomic sketches constructed can be conveniently reduced when the available memory is so tense. For the time complexity issue, there are three main parts for both algorithms:

1. Updating the sketches for *MSketch* or the frequency array for multi-binary join for every new arrival.
2. Obtaining the priority value for each tuple.
3. Searching for the least priority tuple for load shedding.

For multi-binary join, the time spent on (1) and (2) are constant. For *MSketch*, the time spent on (1) is also constant since it is only a computation of a product of k integers where k is the number of attributes of the new tuple. For (2), *MSketch* spends constant time for each input tuple since the estimated productivity is only a product of n atomic sketches. In fact, because each tuple will only stay in the system at most two consecutive tumbling windows, we only need to compute its productivity at most twice during its lifetime. For (3), both algorithms should spend equal time on that. Therefore, *MSketch* and multi-binary join have the same time complexity. However, *MSketch* requires less space. To verify this, we will compare the time spent for different algorithms in the experiment section. We will also show the fact that *MSketch* does not add much time overhead for the whole multi-way join computation.

4.1. Tuple-Based Window Joins

Our algorithm is able to handle tuple-based window joins by a simple extension. We can model a tuple-based window join by a time-based window join where a single tuple arrives to its window every time unit. This can be obtained by assigning a sequence number as its timestamp when a tuple arrives to its window. Moreover, we modify the steps of *MSketch* for resetting the atomic sketch (Step 1.2) and for expiring tuples (Step 1.3). Indeed, when a tuple t_i arrives to its window W_i, we only check whether the atomic sketch of W_i is expired. For a tuple-based window, we reset its atomic sketch for every n input tuples. Also, we only expire tuple(s) in W_i.

5. Experimental Results

Algorithms for Multi-Joins. We compare our algorithm with the state of the art:

- *MSketch:* We dropped the tuple with lowest productivity in the window(queue) when it(queue) is full.
- *MSketch_RS:* We dropped the tuple with largest fraction of tuples produced in the window when it is full.
- *MSketch*Age:* Similar to *MSketch*, the priority of a tuple r with remaining lifetime t is computed by $t \cdot prod(r)$.
- *BJoin:* We convert the multi-join query to be a multi-binary join query. Then we dropped the tuple with the least partner frequency in the window(queue) when it(queue) is full, as mentioned in §1.
- *Rand:* We randomly dropped tuples in the window(queue) when it(queue) is full.
- *Aging:* We dropped oldest tuples in the window (queue) when the window (queue) is full.

For each algorithm, we allocate the same amount of memory for each window. For the algorithms that use sketching technique, we construct 100 random variables and return their average, i.e., $s_1 = 100, s_2 = 1$. When a new tuple arrives, we keep it if the window or the queue is not full. Otherwise, we make a load shedding decision based on different measures for different methods.

As described in §5.1 and §5,2, we tested these algorithms on both synthetic and real-life data sets. An in-depth discussion of the experiments so obtained is presented in §5.3.

Table 1. Description of the Synthetic Data.

Relations	$R_1(A_1, A_2), R_2(A_1, A_2), R_3(A_1, A_2)$
Size of each dataset	≈ 30000 (10000 for each relation)
Size of each domain	1024
Number of regions	10
Volume	100
z-inter	1
z-intra	0.1–0.5, 0.6–1.0, 1.1–1.5, 1.6–2.0

5.1. Synthetic Data Sets

We used the synthetic data generator employed in [17] to produce a wide variety of synthetic datasets using different skews and frequency distribution shifts as follows: it produces several rectangular regions that are uniformly distributed in the multi-dimensional attribute space of each relation by randomly picking a set of points in the attribute space as their centers. Tuples are then generated using a Zipfian distribution with parameters *z-inter* and *z-intra*, across different regions and within the same region, respectively. The frequency of a value depends on the distance from its center, where the one near the center is more frequent. We generate four data sets with different centers of regions and different range of *z-intra*. Each data set contains three relations. Table 1 summarizes the setting of our data sets.

To simulate distribution drifts, we input the tuples to the system from the sources alternatively in a prescribed order. Within each region, we input the data tuples in a random order. The average input rate is 1 tuple per second. We evaluate all the algorithms using a multi-join query

$$R_1 \bowtie R_2 \bowtie R_3|_{R_1.A_1=R_2.A_1, R_2.A_2=R_3.A_1}$$

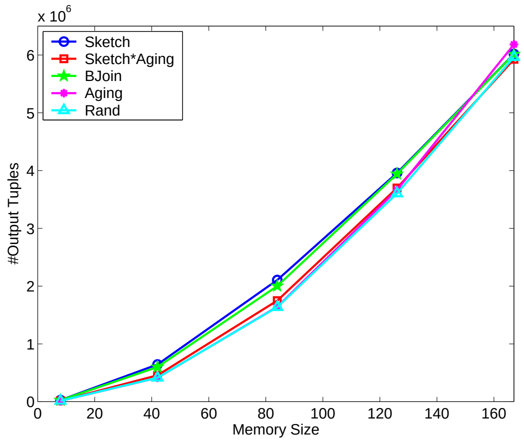
with time-based sliding window $w = 500$ seconds. For this simple query, there are two different join attributes—one for R_1 and R_2 and other for R_2 and R_3. Therefore, this query is able to test the effect of the frequency distribution of a stream outside the joined pair in order to conclude whether this should be considered when making load shedding decisions.

For the algorithm *BJoin*, there are several ways to implement the multi-join as a sequence of binary joins. However, we found that the results obtained by different sequences of binary joins are nearly the same. So, we will only report the results for $(R_1 \bowtie R_2) \bowtie R_3$.

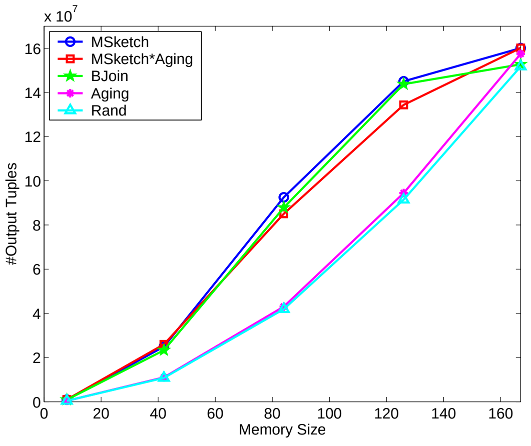
5.1.1 Max-Subset

To study the performance of our max-subset approach, we compare the output size produced by our algorithm *MSketch* with other algorithms under different situations. We first study the case that the input rate is smaller than the join processing rate. Hence, the queue is always empty.

◇ **Effect of Buffer Size.** Our first set of experiments is aimed to study the effect of memory sizes for different algorithms. Figures 2(a) and 2(b) show the performance of each algorithm for different buffer sizes, shown as number of tuples (or percentages with respect to the full window). For each of the following window sizes, 8 (5%), 42 (25%), 84 (50%), 126 (75%) and 167 (100%), we tested data sets with *z-intra* = 0.1–0.5, 0.6–1.0, 1.1–1.5, and 1.6–2.0. Since the results for the last three ranges are similar, we only report one of them.



(a)



(b)

Figure 2. Dataset with *z-intra* = (a) 0.1–0.5 (b) 1.6–2.0.

In Figure 2(a), the result of every algorithm is almost the same. That can be explained by the fact that the intra-Zipfian parameter for this data set is small. Therefore, the distribution of each value is nearly uniform and hence every tuple is equally important. When we increase the value of the intra-Zipfian parameter, some value near the center is more frequent. Figure 2(b) shows that our algorithm *MS*-

ketch is quite effective at determining the productivity of each tuple. Also, observe that the algorithms *MSketch* and *Bjoin* are the best while *Rand* and *Aging* perform poorly on these data sets.

Moreover, in Figure 3, we recorded the whole processing time(for making load shedding decisions and processing joins) spent by each algorithm for the data set with *z-intra* = 1.6–2.0. Note that the computation time for *MSketch* and *BJoin* are almost the same. Also, this is expected that *Rand* is the fastest one because it does not need to do any computation for making load shedding decisions. However, we observe that the difference is not large. Time spent by *MSketch* for making load shedding decisions is relatively small, comparing with join processing time. This shows that *MSketch* does not add much time overhead for the multi-way join computation.

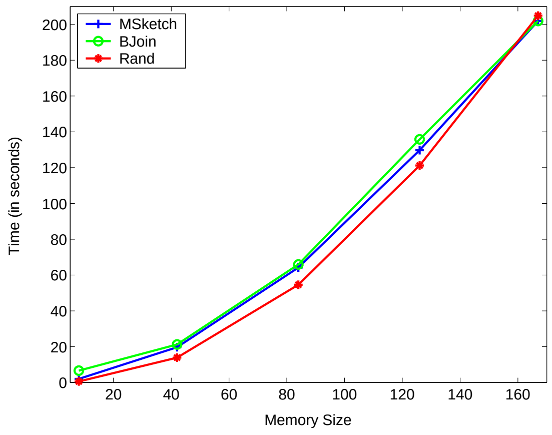


Figure 3. Time for Dataset with *z-intra* = 1.6–2.0.

The experimental results discussed so far were obtained by a fixed memory allocation for each window. We also compared the performance for the case when the memory allocation to each window can vary while the overall total memory used by the windows remains fixed. To achieve this, we simply computed the measure for every tuple in every window and drop the tuple with the least value (in Step 1.5 of *MSketch*). However, we found that the improvement is not so significant but the time spent for making load shedding decisions grew accordingly. Therefore, in the following experiments, we allocated equal amount of memory to each window.

◇ **Effect of Skew.** We now fix the amount of memory to be 42(25%) and vary the skew of the data sets by setting *z-intra* = 0.1–0.5, 0.6–1.0, 1.1–1.5, 1.6–2.0. Figure 4 shows that, when the Zipfian parameter is small (near 0.1), all the algorithms have almost identical performance because every tuple is equally important. However, as the value of the Zipfian parameter increases, our algorithm *MSketch* outperforms others because it is able to distinguish between tuples that have different priority values. Therefore, when the skew in the input data becomes larger, the gap between *MSketch* and other algorithms increases rapidly.

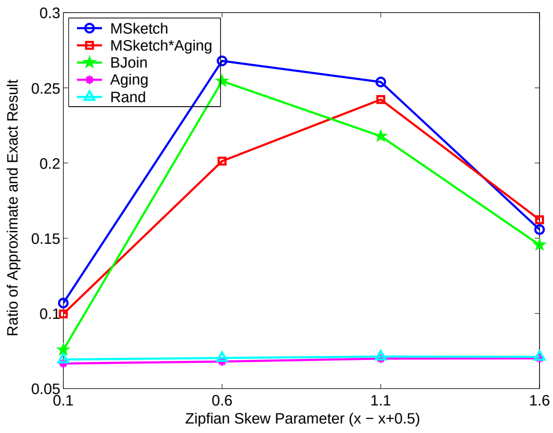


Figure 4. Different *z-intra* with 25% Memory.

◇ **Effect of Concept Drift.** We now use the data set with *z-intra* = 1.6–2.0 and fix the amount of memory to be 75%. In Figure 5, we record the number of output tuples produced for every 250s. The vertical dotted lines represent the existence of concept drifts. Observe that all the three algorithms have a suddenly drop when concept drift occurs. This is due to the fact that the distribution of the tuples in the windows and that of the incoming tuples should be different when there is a concept change, which will affect any kind of algorithms. Besides this, we expect that *Rand* will have no other effect caused by concept drifts because its load shedding decisions do not depend on any historical data. However, we find that our algorithm recovers as quick as *Rand*, which indicates that our algorithm did not suffer from the existence of concept drifts.

◇ **Effect of Arrival Rate.** In this experiment, we used the data set with *z-intra* = 1.6–2.0 and assumed that the input rate is 5 times faster than the join processing rate to study the behavior of the join algorithms *MSketch, BJoin, Aging, Rand*. As a result, the queue is formed and it only keeps 10 tuples. Figure 6 shows the performance of each algorithm with different buffer size. We found that our algorithm *MSketch* works much better when a queue is formed. This shows that our measure is also suitable for making load shedding decisions for the queue.

5.1.2 Random Sampling

In this set of experiments, we demonstrate the ability of our algorithm *MSketch* to provide a statistically accurate random sample of the true result. In this case, we use the fraction of the produced join tuples of each tuple as a priority measure and we simply call it as *MSketch_RS*. We will compare it with two algorithms:

- *BJoin*, which depends only on the distribution of the pairwise relation.

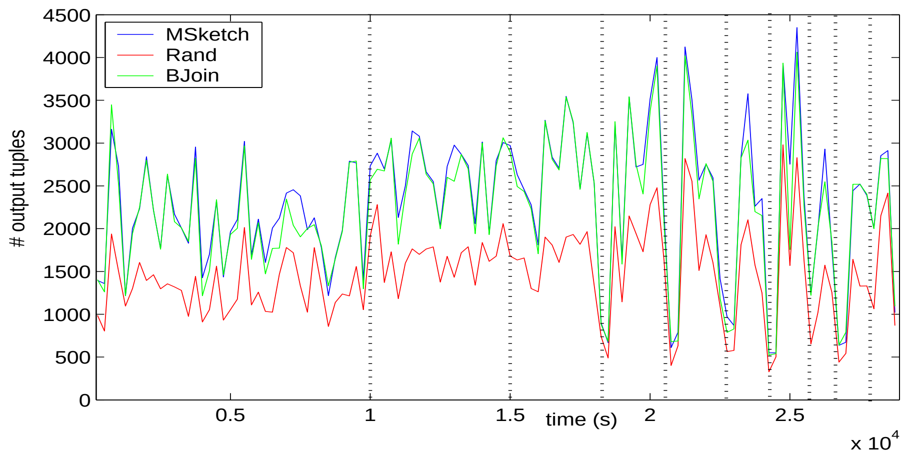


Figure 5. Number of tuples produced over time for dataset with *z-intra* = 1.6–2.0 and 75% Memory.

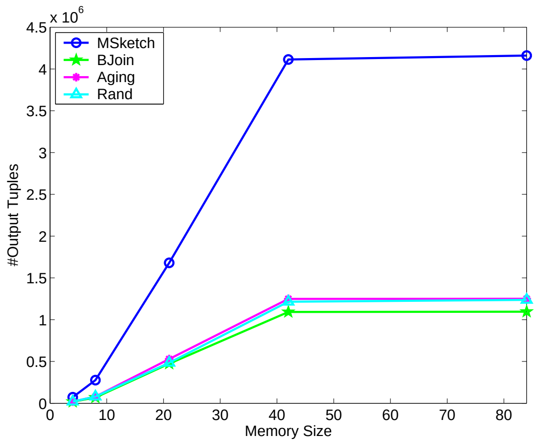


Figure 6. Performance with Queue Formed.

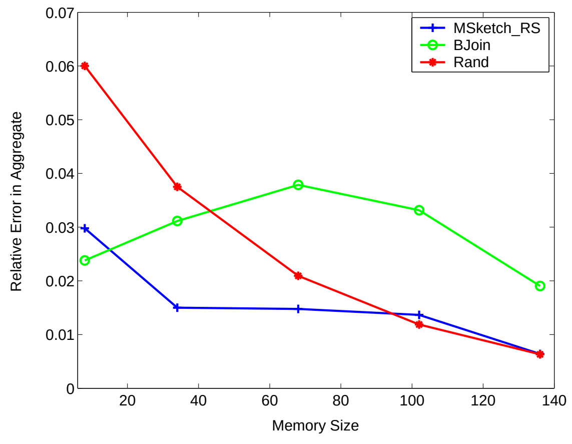
- *Rand*, which randomly drop the load from the input relations.

To study the performance of our sampling approach, we perform a windowed average over the sampled join result, which is a typical application of uniform random sample. We compare those algorithms in terms of aggregation error. In this experiment, we use the data set with *z-intra* = 1.6-2.0 with window size = 500. We choose A_2 of R_1 to be our aggregated attribute. However, in general, it can be chosen from any relations. At each step, the value of the windowed average over the true result AVG and the sampled result $A\hat{V}G$ are computed. The relative error is

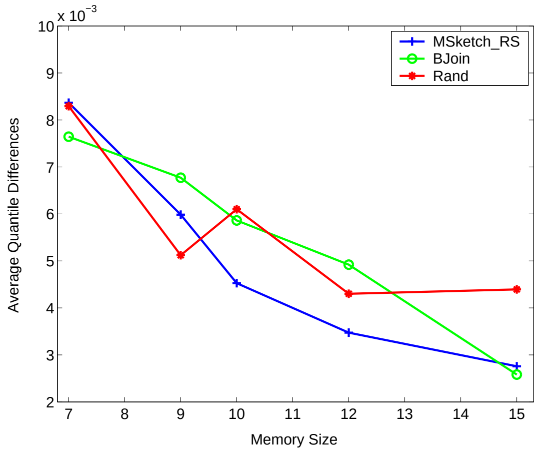
$$\frac{|AVG - A\hat{V}G|}{AVG}.$$

Then we return the average relative errors under different amount of memory allocation. Figure 7(a) shows that the errors produced by our sampling algorithm *MSketch_RS* are much smaller than the errors produced by other algorithms.

To further test the performance of *MSketch_RS*, we compute the quartiles of the join result of each window and return the average differences of the quartiles of the true result and the sampled result. This test can effectively show



(a)



(b)

Figure 7. (a)Average relative errors in aggregates (b)Average quantile differences.

the statistically accuracy of each sampled result, i.e., how similar is the frequency distributions of the sampled result to that of the true result. In figure 7(b), we observe that the average quartile differences of the sampled result produced by *MSketch_RS* and the true result is much smaller than produced by *Bjoin* and random sampling *Rand*.

Note that *MSketch_RS* beats the existing algorithms. From these two experiments, we learn from the poor performance of *Rand* that a random sample of each input relations may not produce a random sample of the join result in general. Moreover, the performance of *BJoin* shows that the information of the pairwise relation is not enough for a good load shedding decision.

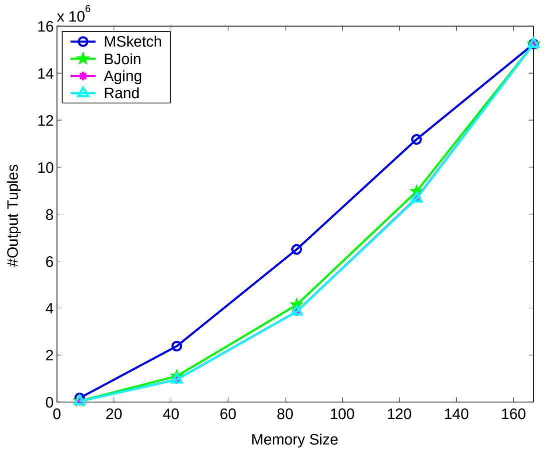
5.2. Tests on Real-Life Data Sets

In these experiments, we used the census data sets from `www.bls.census.gov`. This data set was taken from the Current Population Survey (CPS) data, which is a monthly survey of about 50,000 households conducted by the Bureau of the Census for the Bureau of Labor Statistics. There are about 135,000 tuples with 361 attributes for each month. In our experiments, we pick three attributes: Age, Income, Education. The domain of Age, Income, Education is discretized as 1–90, 1–16, 1–46 respectively. We used the data of October 2003(*Oct03*), April 2004(*Apr04*) and October 2004(*Oct04*). We remove all the tuples with missing value and get about 65000 tuples for each data set of each month. Therefore, the total number of tuples in this experiment is about 200k.

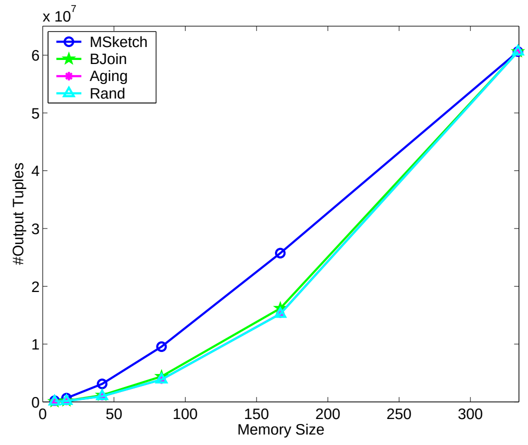
◇ **Effect on Window Size.** This set of experiments was designed to study the effects of window size on a real-life data set. In our experiments, we joined three data streams *Oct03*, *Apr04*, and *Oct04*, on their attributes Age (shared by *Oct03* and *Apr04*) and Education (shared by *Apr04* and *Oct04*). We tested for windows of size 500 and 1000. Figures 8(a) and 8(b) show the output size of the join using different algorithms for window size 500 and 100, respectively, under different memory allocation: for window size of 500, we allocated 5%, 25%, 50%, 75% and 100% memory for the buffers. For window size of 1000, we allocated 2.5%, 5%, 25%, 50% and 100% memory. In this application, we found that the performance of every algorithm is not affected by the size of the window. Our algorithm *MSketch* outperforms others for both settings.

5.3. Discussion

In the above experiments, we investigated the performance of the various algorithms under different settings of the following parameters: buffer size, intra-Zipfian skew parameter, window size, load shedding from the queue. For the max-subset problem, our algorithm *MSketch* outperforms all other algorithms in all the experiments. Likewise, in the random sampling problem, the algorithm *MSketch_RS* provides more accurate results for windowed aggregates and a statistically accurate random sample of the true result.



(a)



(b)

Figure 8. Window Size= (a) 500s (b) 1000s.

Some interesting observations on the various algorithms can be made from these experimental results. In the experiments of the max-subset problem, we observe that the algorithms *Aging* and *Rand* have similar performance. Now, *Aging* assigns a higher productivity to tuples with longer remaining lifetime, and therefore is less likely to drop them; however, *Rand* just makes random choices. The fact that the two algorithms deliver similar performance suggests that the remaining lifetime of tuples might not be critical in optimizing load shedding decisions.

The poor performance of *MSketch∗Aging* is consistent with the results obtained by Das et al. in [7]. They explained this by conjecturing that tuples with low probability of appearing in the other stream are unlikely to survive until they expire. Hence, the product of remaining lifetime and probability is not a good measure. Moreover, the remaining lifetime of a tuple increases its productivity at the same rate in which it increases the cost of keeping it in the buffer (and bringing about other load shedding decisions). This confirms the experimental observations on the behavior of *Aging*, and let us conclude that the remaining lifetime of a tuple should not be considered as a critical factor in setting the priority of that tuple.

For the algorithm *BJoin*, its performance is always worse than the performance of our algorithm *MSketch*. This confirms that, in making load shedding decisions, it is not enough to consider the frequency distribution of joined streams pairs and disregard the other streams in the multi-way join.

In the experiments of random sampling problem, the performance of our algorithm *MSketch_RS* is significantly better than that of *Rand*. This confirms that, in general, random samples of the input tables, do not produce a random sample of the join result: a specialized semantic load shedding technique is instead required. Moreover, the poor performance of *BJoin* confirms once more that it is not sufficient to consider only pairwise relations. We must consider all the other relations in the multi-way join. Thus, the load shedding problem of multi-joins must be viewed as a new problem that cannot be reduced to multi-binary joins. Therefore, our algorithm *MSketch_RS* uses a fraction of the tuples produced by each tuple to control the output result in order to provide a random sample of the true result. The effectiveness of this method was confirmed by many experiments.

6. Conclusion and Future Work

Computing multi-way joins on high speed data streams represent a serious challenge for a DSMS since this operation can significantly overburdens its resources. Therefore, it is important for DSMS to be able to provide effective load shedding strategies for multi-way joins. Previous approaches converted the query into multiple binary joins and disregarded the content of streams outside each joined pair when taking load-shedding decisions. In this paper, we have provided a thorough analysis of the expected tuple contribution over the whole join, and introduced a sketching-based technique to estimate such contribution. Using this measure to determine the priority of the tuples, our algorithm *MSketch* realizes a a very effective load-shedding policy, where only minimal time and space overheads are required. Experiments on synthetic and real-life data sets and a thorough study of different algorithms demonstrate that *MSketch* outperforms other approaches in terms of maximizing the output size and achieving randomization.

Random sampling is used in many different ways in advanced database and business intelligence applications. We are particularly interested in the application of random sampling for the purpose of mining data streams [11]. A statistically accurate random sample is usually sufficient to answer stream mining queries such as clustering and classification: this approach ca save much computation time and memory. Therefore, one promising direction for future work is to extend the load shedding techniques proposed in this paper for computing a wide range of stream-mining queries.

References

[1] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, 1999.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[3] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[6] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *SIGKDD*, 2000.

[7] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.

[8] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD*, 2002.

[9] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[10] Y.-N. Law and C. Zaniolo. An adaptive nearest neighbor classification algorithm for data streams. In *PKDD*, 2005.

[11] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of SQL for mining data streams. SIGMOD 2005 demo paper.

[12] R. Motwani, J. Widom, A. Arasu, B. Babcock, M. D. S. Babu, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[13] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, 2004.

[14] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[15] N. Tatbul, U. Setintemel, 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*, pages 309–320, 2003.

[16] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[17] J. Vitter and M.Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD*, 1999.