Approximating Streaming Window Joins Under CPU Limitations

Ahmed Ayad
Jeffrey Naughton
Stephen Wright
Utkarsh Srivastava

Technical Report #1542

November 2005

Approximating Streaming Window Joins Under CPU Limitations

Ahmed Ayad Jeffrey Naughton Stephen Wright Utkarsh Srivastava

University of Wisconsin-Madison
Computer Sciences Department
{ahmed, naughton, swright}@cs.wisc.edu

Stanford University
usriv@db.stanford.edu

Abstract

Data streaming systems face the possibility of having to shed load in the case of CPU or memory resource limitations. In this paper, we study the CPU limited scenario in detail. First, we propose a new model for the CPU cost. Then we formally state the problem of shedding load for the goal of obtaining the maximum possible subset of the complete answer, assuming an offline scenario. We then present an online strategy for semantic load shedding, assuming a frequency model. Moving on to random load shedding, we prove that having an open memory budget does not help the CPU constrained case. We then present a random load shedding strategy – Probe-No-Insert – based on decoupling the window maintenance and tuple production operations of the symmetric hash join, and prove that it always dominates the previously proposed coin flipping strategy. Finally, we investigate the problem in case the goal is to obtain a random sample of the join.

1. Introduction

In the context of data streaming systems, the system has no control over the rate of the incoming data. Hence, the adoption of a *push* model of computation is mandatory. In steady state, the system resources must be greater than what is required by the input otherwise the system is unstable and the query is infeasible [6]. The requirement of online answers adds a constraint on the ability to offload some of the input to disk for later processing, since this will add considerable lag on the time of producing an answer. Hence, the system is required to process the input data in memory, and is usually granted only one pass over the input. Examples of system resources that need management are:

1. *Computational resources*: the system CPU has to compute the results of the query faster than the arrival rate of the input. Otherwise, the input queues and the response time of the system will grow indefinitely.
2. *Memory*: the amount of available memory has to be enough to hold the required state of the query (e.g. window predicates) plus tuples arriving in the input queues of the data streams that await processing.

If the system resources are less than the input requirements, some of the input must be shed to bring the load down to within the available capacity. We motivate the problem of load shedding for CPU limited execution by the following arguments:

1. The work investigating memory limitations looks at the case when the system can not hold the complete state of the operators (e.g., the contents of the window predicate). It does not address the case when input queues are overflowing which is a result of high CPU utilization. This means CPU limitation can be a cause of memory limitations, even if enough memory is available to store the state of the operators. The reverse is not true, unless extra processing is required to spool data to secondary storage, which is usually not acceptable for streaming systems.
2. A side effect of shedding load to reduce the CPU utilization can be less state that needs to be stored. Thus, reducing the CPU load can be solution to the memory limited problem. The reverse is also true, reducing the memory consumption leads to a reduction in the CPU consumption.
3. It is easier to solve the memory limitations by adding more memory to the system without changing any of the available techniques for processing streaming queries. In the CPU limited case, however, the technology has to improve or, if more processors are available, parallel or distributed versions of the operators have to be used to utilize the additional processors.

In this paper, we investigate the problem of load shedding for the streaming window join operator under CPU limitations in detail. We start by revising the unit time model for the CPU cost of executing the join. For obtaining the maximum subset of the join, we look at offline and online algorithms for load shedding in case the data distribution is known and in the case where only the type of the distribution can be assumed. We also look at the problem of obtaining a random sample of the result. In particular, the contributions of the paper are the following:

- We present an accurate unit-time model for the CPU cost of executing a streaming window join.
- We formulate the theoretical problem of obtaining the maximum subset of a streaming window join under a limited CPU budget in an offline scenario. The work in [12] has looked at the same problem in the memory limited case.
- Unlike the memory limited case, we prove that the offline problem is NP-Hard, and propose a greedy heuristic for its solution to guide the online strategies.
- We develop an online strategy for semantic load shedding based on the heuristic solution to the offline problem.
- We study a different set of strategies for random load shedding that decouples the decision of inserting and probing tuples.
- We prove that one such strategy, the Probe-No-Insert, outperforms previously suggested methods for random load shedding. In doing so, we show that an unbounded memory budget does not help in the CPU limited case.
- If the goal is to have a random sample of the join result, we present an adaptation of a previously suggested technique for the memory limited case that works for the CPU limited case.

The rest of the paper is organized as follows: Section 2 discusses some preliminary definitions and presents the refined cost model. Section 3 investigates the problem for the maximum subset goal. Section 4 investigates it for producing a random sample. Section 5 discusses related work and Section 6 concludes the paper.

2. Preliminaries

This section lays the necessary foundation for the discussion. We start by formal definitions of the streaming model and the precise semantics of the windowed streaming join. Then we present an accurate unit time CPU cost model. Finally, the goals of load shedding are discussed.

2.1 The Data Streaming Model and Streaming Window Join Semantics

We adopt the definitions of data streams and sliding windows in [3].

Definition 1: Data Stream.

A streams S is a bag of elements $<s, t>$, where s is a tuple belonging to the schema of the stream, $t \in \mathcal{T}$ is the timestamp of the element, and $\mathcal{T}$ is a global, discrete, ordered time domain ❑

Table 1. List of variables used and their definitions

C_{Pr}	Cost to probe an active window for a matching tuple just arriving
C_I	Cost to insert an arriving tuple into the sliding window
C_V	Cost to invalidate an expired tuple from the sliding window
C_u	The cost of updating the state of the join window per tuple
C_p	The cost of producing a single tuple from the result of a window join
λ_i	Rate of arrival of tuples for stream i
$\lambda_{o/p}$	Output rate of a join
T	Size of a time-based window

Window predicates are a means to restrict an infinite stream for operations like stream joins to become feasible and a means to express interest in a portion of the input stream. For the sake of this work, we will only consider time-based windows.

Definition 2: Time-based Window.

At any time instant t, a time-based window of size T on a stream S, denoted as $S[T]$, defines a subset of S containing all elements of S with timestamp t' such that $t\text{-}t' \leq T$. ❑

We precisely define the semantics of the sliding window streaming join as follows:

Definition 3: The Sliding Window Streaming Join.

The sliding window join, $L[T_1] \bowtie R[T_2]$, is a symmetric operator that takes two input streams, L and R with window predicates T_1 and T_2 defined on them. For every arriving tuple on any of the two input streams, the operator joins it with the current window contents on the other input stream. The operator then streams out resulting tuples that satisfy the join predicate. The timestamp of a resulting element from the join is the greater of the two timestamps of its components. The resulting stream is ordered on the timestamps of its elements. ❑

2.2 Revising the CPU Cost Model

Before developing load shedding methods, we begin by examining the CPU cost model for performing a join between two streams. We assume steady state conditions and use the average rate to characterize the rate of arrivals of incoming tuples from external sources. This implicitly assumes a stable arrival rate. We also assume that there is enough memory to hold the buffering requirements for any query plan. Table 1 defines the notation used throughout the paper. All costs are in time units.

The previously proposed model [14] divided the cost into three distinct parts; insertion, invalidation, and probing costs. When a tuple arrives at one side of the join, the operator has to a) insert the tuple in the window, b) expire tuples whose timestamps fall outside the window predicate, and c) probe the window on the other side for matching tuples. Accordingly, the cost of the join between two streams L and R – assuming symmetric join methods on both sides – can be expressed by the following:

$$C_{L\bowtie R} = (C_I + C_V + C_{Pr})\cdot(\lambda_L + \lambda_R) \qquad (1)$$

One problem with this model is the assumption that the probing cost is fixed for all tuples regardless of the parameters of the join. Since the window contents are memory resident and the cost is dominated by in-memory CPU operations, it depends on the number of entries an incoming tuple needs to look at to check the join condition. Another problem of the model is that it does not account for the production cost of tuples. This includes the memory copy cost to form the joined tuple, and the cost of inserting it in the queue of an upstream operator or the query result output stream.

To model the CPU cost of the symmetric streaming hash join we will assume the following implementation:

- Tuples satisfying the window predicate are stored in a hash table such that each unique value of the join key maps to a unique hash bucket (this is assumed for ease of exposition. In practice, any index structure that provides, in the average case, a constant time access to all chains of tuples sharing the same join value can be modeled in a similar fashion).
- Duplicate key values are chained in the bucket according to the tuple timestamp values, with the newest tuple at the head of the chain.

Using these assumptions, we can describe the symmetric hash join algorithm as follows: when a tuple arrives at the left input stream three actions are taken (handling arrival from the right side is symmetric):

a) The tuple is hashed into the corresponding bucket in the window of the left side, and inserted at the head of the chain.
b) The corresponding hash bucket on the right side of the join is probed. The chain of tuples in the bucket is traversed until tuple timestamps fall out of the window predicate. A reference to the first expired tuple is kept. For every probe, a join result is produced.
c) Using the reference kept in (b), the rest of the chain is traversed expiring tuples from the window.

An important characteristic of the algorithm described above and the assumptions it is built on is that it represents an ideal CPU utilization case. In all three phases of execution (insertion, probing/production, and expiration) the algorithm has no overhead, this will be helpful when we later investigate the problem of load shedding in a CPU limited environment.

Using the above implementation, we can build a more accurate unit-time cost model of the CPU cost for the symmetric join. There are four operations performed for every incoming tuple. *Inserting* a tuple involves a hashing operation followed by an insertion at the head of the chain. *Expiring* a tuple involves one chain traversal to the tuple followed by releasing the tuple to memory. For every output tuple there is a *probe* operation which involves traversal to the tuple followed by a *produce* operation which involves concatenating the two joined tuples and producing the result. The four operations can be categorized into *update* operations; which include insertion and expiration, and *production* related operations; which include probing and producing the results. The cost of update operations is proportional to the tuple arrival rate (in steady state, every tuple inserted in the window has to expire), and the cost of production operations is proportional to the output rate of the join.

Let C_u be the cost of inserting and expiring a tuple from the window, C_p be the cost of probing and producing one result tuple, and $\lambda_{o/p}$ be the output rate. The cost of the join can then be expressed as

$$C_{L\bowtie R} = (\lambda_L + \lambda_R)\cdot C_u + \lambda_{o/p}\cdot C_P \qquad (2)$$

2.3 Goals of Load Shedding

Streaming applications differ in their requirements when faced with the inability to produce the full answer. Different applications require different characteristics in the approximate answer produced by the load shedding scheme. Depending on such requirements, the goal of load shedding is set. In some cases, the best course of action might be to produce as many tuples of the original answer as possible. This goal is known in the literature as the Max-subset [12] or the Max-recall [17]. However, if the result of the join is fed into an aggregate function for example, then Max-subset might not represent the best choice for approximation. In this case, a more representative sample of the complete answer might be more desirable in terms of producing a better approximation of the aggregate with known error bounds. The goal of load shedding should then be to obtain a uniform random sample of the result, or if none can be obtained, to get as close to uniform a sample as possible.

An orthogonal dimension to consider is the availability of statistical information on the data distribution of the input stream. Semantic load shedding, in which the strategy intelligently picks tuples to discard based on their values, are possible if such information is available. Otherwise, random load shedding is the only option.

In Section 3, we discuss the load shedding problem under CPU-limited constraints for the Max-subset problem. In Section 4 we discuss the problem for the goal of obtaining a random sample.

3. The Max-subset Goal

We investigate the problem for the goal of obtaining the maximum possible subset of the answer given the CPU budget. The problem is formally defined as follows:

Definition 4: Max-Subset Strategy

Given a sliding window join $L[T_1] \bowtie R[T_2]$ with the input rates of λ_L and λ_R respectively, and given a unit time CPU cost budget of C, if the cost of the join $C_{L\bowtie R}$ exceeds C, the Max-Subset method finds the best load shedding strategy such that $C_{L\bowtie R}$ is within C while maximizing $\lambda_{o/p}$. ❑

In the following, we tackle the problem assuming decreasing levels of knowledge about the input stream characteristics. We first start with examining an offline strategy assuming perfect knowledge of the join result in the future. The goal of this exercise is to have a baseline strategy against which to measure the online strategies – for which such knowledge does not exist. A similar approach for memory limited load shedding is taken in [12]. We then move on to an online semantic shedding strategy if we are given information about the frequency distribution of the input data. Finally, we investigate random shedding strategies.

3.1 The Optimum Offline Strategy

Using the model developed in Section 2.2, we investigate the optimum load shedding strategy for a streaming window join in the CPU-limited scenario. Since the join query is continuous, the sizes of the input streams are infinite. Hence, modeling of the complete result of join is infeasible. Instead, we model only a prefix of the join result that extends until a specific time in the future.

Consider the join $L[T_1] \bowtie R[T_2]$. Assuming we are looking T time units into the future, the total CPU cost budget available for the join operator K is $T*C$ where C is the unit time capacity of the CPU. According to equation (2), there is a cost C_u for maintaining each tuple in the input and a cost C_p for producing an output tuple. We can represent the join result as a bipartite graph in which the set of nodes on the right (left) hand side represents tuples of L (R) and an edge joining two input tuples represents the tuple resulting from their join, with C_u attached to the nodes and C_p to the edges. The optimum algorithm should select a subset of the output tuples such that the cost of their production is less than K while maximizing the number of selected edges. We give a more abstract definition of the problem as follows:

Definition 5: Offline Max-Subset Problem

<u>Input:</u>

- A bipartite graph $G = \langle U, V, E \rangle$
- Costs C_u and C_p attached to the nodes and edges respectively
- A cost function on the graph F(G) defined as: $F(G) = (|U| + |V|) \cdot C_u + |E| \cdot C_p$
- A total cost budget K

<u>Output:</u>

A subgraph $G' = \langle U', V', E' \rangle$ of G such that $F(G') \leq K$ and $\forall G'' = \langle U'', V'', E'' \rangle \subseteq G$, $F(G'') \leq K \Rightarrow |E''| \leq |E'|$ ❑

There are some characteristics to note on the composition of the input graph. The graph is divided into separate disjoint components, one for each unique value of the join key. Unlike an equijoin of relational tables though, the subcomponents of the graph in this problem need not be complete bipartite graphs since the window predicate may prevent two tuples on both sides with the same join key from joining together.

Notice that an algorithm solving the above problem has the choice of including and excluding edges in the answer regardless of whether this choice will involve adding additional nodes. Another variant of the problem is to restrict the strategy to consider only induced subgraphs as the answer. An induced subgraph is a subset of the vertices of a graph G together with any edges whose endpoints are both in this subset. This variant can be defined as follows:

Definition 6: Offline Induced Max-Subset Problem

<u>Input:</u>

- Same as Definition 5, plus a predicate, Induced(G', G), that is true iff G' is an induced subgraph of G
- A cost function on the graph F(G) defined as: $F(G) = (|U| + |V|) \cdot C_u + |E| \cdot C_p$

<u>Output:</u>

An induced subgraph $G' = \langle U', V', E' \rangle$ of G such that $F(G' \leq K) \wedge \forall G'' = \langle U'', V'', E'' \rangle \subseteq G$, $F(G'') \leq K \wedge Induced(G'', G) \Rightarrow |E''| \leq |E'|$ ❑

This models the case in which the strategy either accepts an incoming input tuple or drops it completely from consideration. The intuition behind this is that in practice, for the symmetric hash join, including part of the join result produced by an incoming tuple usually means that a probing procedure has to be performed. If we assume the implementation in Section 2.2, this means the list of matching join entries has already been traversed; including matching tuples that we will not be used to produce output. The answer produced by this variant is possibly suboptimal. It might be the case that the cost budget allows for the inclusion of a node and a subset of its incident edges. If this is the case, the take-all leave-all decision forces the strategy to exclude such node from consideration, hence underutilizing the budget.

The following statement about the complexity of these two variants is true:

Theorem 1

Both Offline Max-Subset and Offline Induced Max-Subset are NP-Hard.

Proof

The proof is by reduction from the balanced bipartite clique problem, which is NP-Hard [13]. The reduction is as follows:

Given a bipartite graph $G = \langle U, V, E \rangle$, it is required to know whether a K-balanced bipartite clique (a bipartite clique with K nodes on each side.

We call Offline Max-Subset with G as the input graph, $C_u = 1$, $C_p = 0$, and a cost budget of $2K$. Note that in this case, the cost of the graph is the number of nodes it has.

Examine the answer subgraph returned. If the number of edges is K^2, then the answer to the problem is yes. Otherwise, the answer is no.

This is true since for a budget of 2K nodes, a balanced bipartite clique of size $2K$ will contain K^2 edges, and it is easy to show that this is the maximum number of edges in a bipartite graph with $2K$ nodes. So, if one exists it will be within the budget, and it must be returned as the answer to Offline Max-Subset.

Since the reduction is done in polynomial time, we conclude that Offline Max-Subset is NP-Hard.

Offline Induced Max-Subset has an identical reduction. ❑

3.2 An Online Semantic Load Shedding Strategy

We now shift the focus to developing shedding strategies for the Max-Subset goal that executes online. Since the offline problem itself is hard, we explore simple heuristics that approximates the offline solution first then use them to inspire the online shedding strategies.

3.2.1 Offline Heuristics

We can envision the procedure of producing the optimum offline answer as an oracle that selects the edges of the optimum graph one by one, including them in the answer – along with the nodes they are induced on, if they are not already included – and then stopping when all the edges in the answer are included. This way, we can try to approximate this procedure by a deterministic algorithm that selects edges for inclusion in the answer according to a specific criterion. Two such criterion come to mind:

```
Algorithm: Greedy Max-Subset
Input:
  Bipartite graph $G = \langle U, V, E \rangle$, $C_u$, $C_p$ and K
Output:
  Bipartite graph $G' = \langle U', V', E' \rangle$ s.t. $F(G') \le K$
Procedure:
1. Divide the graph into its connected subcomponents.
2. For each subcomponent $G' = \langle U', V', E' \rangle$ compute the value $C(G')$.
3. Order all subcomponents descendingly by $C(G')$.
4. While the budget allows, pick the component with the highest $C$ value and include it in the answer decreasing the remaining budget by its cost $F(G')$.
5. For the last component that could not be completely included, sort its edges descendingly by $C(e)$. Include edges from the component in the answer until the budget is exceeded.
```

Figure 1. The Greedy Max-Subset Heuristic

1. At any point in the inclusion procedure, add the edge to the answer that costs the least to produce.
2. It makes sense to include completely connected components in the answer before moving to new ones. Note that to add an edge; the nodes it connects must be first included in the answer. It costs less to add an edge with at least one of its nodes already there.

To use the first technique, there must be a cost attached to the edges of the graph. According to the cost model, every edge has a cost for its production which should be part of the cost of including the edge in the answer. We then need a method to also account for the cost of maintaining nodes in the answer. A simple way to account for the cost of a node is to amortize it on all its incident edges. More formally, we define the cost of including an edge $e = \langle u, v \rangle$ as follows:

$$C(e) = C_p + C_u \cdot \left(\frac{1}{u_1} + \frac{1}{v_1} \right) \tag{3}$$

where u_1 and v_1 are the degrees of u and v respectively.

To use the second idea, we need a similar metric to measure the utility of a connected component so that we can prioritize the decision to include them in the answer. Notice that the above cost orders the edges descendingly on the quantity $u_1 \cdot v_1 / (u_1 + v_1)$. We can look at this as approximating the gain from including all edges incident on u and v divided by the cost of this inclusion. We can do the same thing for a connected component $G' = \langle U', V', E' \rangle$ of the original graph by defining the cost of including it as follows:

$$C(G') = |E'|/F(G') \tag{4}$$

which is the benefit to cost ratio of including G' completely in the answer.

Using the above we can define a greedy heuristic for the offline case as in Figure 1. It can be easily verified that the above algorithm runs in O(|E|).

3.2.2 Online Strategy

In this section, we suggest a semantic load shedding strategy for the CPU limited case inspired by the greedy heuristic proposed for the optimal offline algorithm.

There can be no online strategy that is competitive with an offline algorithm if we are dealing with arbitrary streams [17]. So, to devise an online strategy, we need to at least have some information on the type of data distribution of the input. For the sake of this work, we assume the frequency-based model [12][17]. Consider the sliding window join $L[T_1] \bowtie R[T_2]$ on an attribute $A \in$ domain $\mathcal{D}$. The frequency-based model can be defined as follows [17]:

Definition 7: The Frequency-Based Model

For all $v \in \mathcal{D}$, a fixed fraction $f_L(v)$ arrives on stream L, and a fixed fraction $f_R(v)$ arrive on stream R with the value v in attribute A. ❑

The work in [6][21][14] has assumed a simpler version of this model in which the frequency distribution is uniform for all values and on both sides of the join. Such value is usually modeled as a single selectivity value f.

We start here by assuming the complete frequency distribution is known. Consider a streaming join with input rates λ_L and λ_R, a tuple with a value v arriving on stream L is expected to join with $f_R(v)\cdot\lambda_R\cdot T_R$ tuples residing in the window of stream R, with $f_L(v)\cdot\lambda_L$ such tuples arriving per unit time. Similarly, a tuple with the same join value on stream R joins with $f_L(v)\cdot\lambda_L\cdot T_L$ tuples in stream L's window, with $f_R(v)\cdot\lambda_R$ such tuples arriving per unit time. Therefore the total rate of output tuples produced for the join attribute value v is:

$$\lambda_{o/p}(v) = f_L(v)\cdot f_R(v)\cdot \lambda_L \cdot \lambda_R \cdot (T_L + T_R) \tag{5}$$

with the total output rate of the join is:

$$\lambda_{o/p} = \sum_{v\in D} \lambda_{o/p}(v) \tag{6}$$

and, the cost of production for this component is:

$$C(v) = (f_L(v)\cdot\lambda_L + f_R(v)\cdot\lambda_R)\cdot C_u + \lambda_{o/p}(v)\cdot C_p \tag{7}$$

Summing over all values v gives the total join cost of equation (2).

For each distinct value v of the join attribute, the quantity

$$P(v) = \lambda_{o/p}(v)/C(v) \tag{8}$$

represents the benefit for cost ratio for this portion of the join.

```
Algorithm: Online Max-Subset
Input:
  $\lambda_L$, $\lambda_R$, $T_1$, $T_2$, the frequency distribution, $C_u$, and $C_p$
Procedure:
1.  Compute $P(v)$ for all $v \in \mathcal{D}$.
2.  Insert all $P(v)$'s in a priority queue.
3.  Initialize total cost
4.  while (true)
5.     pick $u$, the component with highest $P(u)$
6.     if (total cost + $C(u) \le 100\%$)
7.         Add the u component and update total cost
8.         Assign a probability 1 to $u$
9.     else
10.        Store the value of $u$
11.        for (all remaining components)
12.            Assign prob. 0 to corresponding values
13.        break
14. Optimize the component corresponding to the
    value stored in step 10 using random load
    shedding (section 3.3) with a cost constraint equal
    to the remaining cost budget.
15. while (join executing)
16.    $v$ = join attribute of incoming tuple
17.    switch (probability(v))
18.        case 1: process normally
19.        case 0: drop tuple
20.        default: apply strategy from line 14
```

Figure 2. Online Max-Subset Strategy

We propose the strategy Online Max-Subset as in Figure 2. The strategy uses the metric of equation (8) for prioritizing the distinct values of the domain of the join attribute. As long as the cost permits, higher priority components are assigned probability 1. For the component that cannot be included completely, the randomized strategies of section 3.3 are used to shed a portion of the load of this component until the cost is within the total budget. Notice that since the selectivity is fixed for all tuples in the same component, the assumptions made in previous work are true and the randomized method will be the optimum strategy. For tuples belonging to the rest of the components, the tuples are completely dropped.

The PROB heuristic, proposed in [12] for memory limited load shedding prioritizes tuples according the frequency of its join value on the other joining stream. In case of positive correlation of the distribution on both input streams, the Online Max-Subset strategy reduces to this one. However, the Online Max-Subset strategy should perform better in case of negatively correlated distributions, since it looks at the contribution of the complete component to which a tuple belongs to instead of just the contribution of the tuple to the join. We believe this is a better heuristic even for the memory limited case. The work in [18] has a similar loss/gain function for semantic load shedding based on frequency distributions and loss tolerance graph supplied by the application. It is not clear from this work however, how their technique works in case of a streaming join.

3.3 Random Load Shedding for the Max-Subset Goal

We finally examine the case in which the details of the distribution of the input are unknown. However, we can assume the input follows the frequency model. In this case, looking at the join attribute gives no additional info and only random shedding techniques can be applied.

We realize that if we can assume a frequency distribution, one can argue that the actual details can be obtained just by monitoring the input data. However, since we are dealing with a limited CPU case, storing and maintaining frequency information can waste precious cycles. In some cases, the extra benefit of producing more cannot justify the cost of maintaining the frequency data. These cases include, for example, if the distribution is close to a uniform one, in which the difference in behavior between the tuples because of the value of their join attribute is insignificant. Also, if the distribution changes rapidly and needs constant monitory and frequent updates.

For the purpose of this section, we will use the simplified frequency model with a single selectivity f for the whole join for ease of exposition. We note that all the results presented are identical for the generalized model.

The approach previously taken for random load shedding was to find the best setting of random sampling operators applied to the input stream so that the maximum subset is obtained while keeping the load within CPU limits. We shall call this the coin flipping strategy (or CF) [10]. Our contribution is to investigate whether more can be gained by decoupling the update and the production procedures of an incoming tuple in the shedding process. Recall that to execute the join, every incoming tuple has to be inserted in the window of its stream for later matching and it has to probe the window of the opposite stream for matching with tuples that arrived earlier to produce results. The coin flipping approach couples these two procedures by insisting that either the whole contribution of the tuple to the join be taken completely or none is. This is not necessary, since the two procedures are independent. Realizing this, two other approaches arise; namely the Insert-No-Probe (or INP) and the Probe-No-Insert (or PNI) strategies.

Consider the join $L[T_1] \bowtie R[T_2]$ with λ_L and λ_R as the rates of the input streams, with selectivity f. If the join is feasible, the output rate is [6]:

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L + T_R) \qquad (9)$$

In the following we describe each shedding strategy for this join in some detail.

Coin Flipping (CF)

In the CF strategy a sampling operator is placed in front of each of the two streams, with x_L and x_R being the sampling probability of the one on stream L and R respectively. The coin flipping strategy can be formalized as the following optimization problem [6]:

Max

$$\lambda_{o/p} = f \cdot \lambda_L \cdot x_L \cdot \lambda_R \cdot x_R \cdot (T_L + T_R) \qquad (10)$$

Subject to

$$(\lambda_L \cdot x_L + \lambda_R \cdot x_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1 \qquad (11)$$
$$0 \leq x_L, x_R \leq 1$$

Insert-No-Probe (INP)

In the INP strategy, instead of dropping some of the tuples completely, all incoming tuples are admitted into the window. Then a coin is flipped with probability of probing x_L and x_R for L and R respectively. If the flip is a success, the tuple probes the opposite window, otherwise the tuple is dropped. The strategy calls for the best setting of x_L and x_R while maximizing the output rate. It can be formalized as the following optimization problem:

Max

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L \cdot x_R + T_R \cdot x_L) \qquad (12)$$

Subject to

$$(\lambda_L + \lambda_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1 \qquad (13)$$
$$0 \leq x_L, x_R \leq 1$$

The intuition behind this strategy is that since we are not constrained in terms of memory, why not keep all the state we can in the hope that the extra kept state produces more tuples. This strategy appears at first sight to be the candidate for delivering the maximum possible subset, since it capitalizes on the asset which is not constrained; in this case memory.

Probe-No-Insert (PNI)

The PNI strategy is the inverse of the previous one. All incoming tuples are allowed to probe the opposite window, and then a coin is flipped on inserting them in the window for later matching. It can be formalized as follows:

Max

$$\lambda_{o/p} = f \cdot \lambda_L \cdot \lambda_R \cdot (T_L \cdot x_L + T_R \cdot x_R) \qquad (14)$$

Subject to

$$(\lambda_L \cdot x_L + \lambda_R \cdot x_R) \cdot C_u + \lambda_{o/p} \cdot C_p \leq 1 \qquad (15)$$
$$0 \leq x_L, x_R \leq 1$$

3.3.1 Analysis of the Random Strategies

The comparison between the three strategies is done by investigating their corresponding optimization problems. We denote the optimal objective by $\lambda^{*}_{o/p}$ and the optimal solution by x^{*}_{L} and x^{*}_{R}.

The first constraint in each problem insures that the result of load shedding is within the CPU limitations by making sure the system utilization is below 100%. The second set of constraints confine the values of the shedding variables to a fraction between 0 and 1. The following statement is true:

Lemma 1

If the join is infeasible, at least one of x^{*}_{L} or x^{*}_{R} is strictly less than 1 for all three problems.

Proof

Substituting $x_L = x_R = 1$ in all three problems reduces the first constraint to the cost of the join. If the join is infeasible, then this substitution does not satisfy the constraint and at least one of them has to be reduced to satisfy it. ❑

We start the analysis with the following statement about the INP strategy:

Theorem 2

For the same input parameters, the optimal objective of the INP strategy is less than or equal to the optimal output rate provided by the other two strategies, with the equality only in case the join is feasible.

Proof

If the join is infeasible, then from the proof of Lemma 1, the first constraint must be tight at the optimal solution. Hence, we can re-write it as follows for INP:

$$\lambda^{*}_{o/p} = \left(1 - (\lambda_L + \lambda_R) \cdot C_u\right) / C_p \qquad (16)$$

and for CF and PNI:

$$\lambda^{*}_{o/p} = \left(1 - (\lambda_L \cdot x^{*}_{L} + \lambda_R \cdot x^{*}_{R}) \cdot C_u\right) / C_p \qquad (17)$$

By Lemma 1, we are guaranteed that the value of equation (16) is strictly greater than (17).

If the join is feasible, it is easy to verify that the optimum objective for all three is equal to the join output rate. ❑

The above statement proves that, in the CPU limited case, having an extra amount of memory *does not help* if the objective is to maximize the output rate. This seems counter intuitive at first. However, note that the cost of outputting a single tuple of the join result is divided into production cost, which cannot be reduced, and an amortized cost of maintaining the input tuples in the window. In the INP strategy, the tuples that do not probe the opposite window actually pays the price of maintenance while missing on producing tuples, hence the amortized cost of maintenance is high compared to the other strategies.

Theorem 3

The optimum objective of the PNI strategy is greater than or equal to that of the CF strategy, with the equality only in case the join is feasible. ❑

The proof is provided in Appendix A. The theorem effectively states that the PNI is *the best* random load shedding strategy. A similar intuitive argument to the one above can explain why this is true. If the cost of producing a tuple can not be reduced, then the only saving in cost can be through producing as many tuples as possible for the amount of maintenance cost to the windows already incurred. The PNI strategy effectively does that by allowing tuples already inserted in the window to be probed by all the possible tuples it later matches with. The CF strategy misses on some of that by dropping some of the incoming tuples.

4. The Random Sample Goal

We now shift focus to another goal of load shedding, which is to get a uniform random sample of the join result. As could be seen from the online strategies of semantic load shedding in the previous section, the strategies are heavily biased towards components of the join that provide more output tuples. If the application requires a more representative sample, then these techniques are inadequate. Computing an aggregate on top of the join is one such scenario.

A uniform random sample with sampling factor p has two properties: 1) every element of the population has a probability p or inclusion in the sample, and 2) samples are pair-wise independent. A naïve algorithm is to sample the result of the join, which requires computing the full answer first. It was shown in [10] that obtaining a random sample of a relational join from a sample of the input is not possible for general many-to-many joins if the distribution of the data is unknown. The result extends to streaming joins [17].

4.1 Uniform Random Sampling

If the input streams follow the frequency model, and the distribution is known, the UNIFORM algorithm [17] produces a uniform random sample for the streaming join for memory limited execution. UNIFORM takes as input an incoming tuple and the number of tuples it should later join with until it expires. The algorithm then simply simulates a number of coin flips equal to this number and stores the result along with the tuple in the window. When a matching tuple arrives it is checked against the stored result of the simulation. An output is produced only if the corresponding coin flip is successful. After the last successful match is produced for a tuple, it can be discarded ahead of its expiration time. Savings in memory is realized through this early expiration.

UNIFORM can be used to provide a random sample in the CPU limited case. In this case, the savings in CPU cost come from a) the savings in update cost, if the algorithm decides that a tuple will not join with any later ones, so it is not admitted, and b) the savings in production time since a probing tuple will probe less tuples and output only a fraction of the ones probed. To calculate the sampling frequency p, we need to compute the cost of the join if the sampling frequency is p.

Consider the join $L[T_1] \bowtie R[T_2]$ with λ_L and λ_R as the rates of the input streams that follows the frequency model. A tuple arriving on stream L with a join attribute value v will be probed by an expected $f_R(v)\lambda_R T_L$ tuples from stream R. If the sampling fraction is p, the probability that the tuple will not be admitted into the window is:

$$P_{dis}(v) = q^{f_R(v)\lambda_R T_L} \tag{18}$$

where $q = 1\text{-}p$. This is the probability of failure for all coin flips corresponding to the probing tuples. The expected probability of dropping a tuple is:

$$P_{dis} = \sum_{v \in D} f_L(v) P_{dis}(v) \tag{19}$$

Hence, the expected number of tuples admitted into the window on stream L is

$$\lambda'_L = \lambda_L(1 - P_{dis}) \tag{20}$$

The corresponding value for stream R can be computed in a similar manner.

To compute production cost, we need to calculate for a tuple with value v, the average number of tuples it will have to probe on the other side. From [17], the expected lifetime of a tuple with value v inside the window of stream R is

$$t_{dis\_R}(v) = \left(f_L(v)\lambda_L T_R - \frac{q}{p}\left(1 - q^{f_L(v)\lambda_L T_R}\right)\right) \cdot \frac{T_R}{f_L(v)\lambda_L T_R} \tag{21}$$

with a similar value $t_{dis\_L}(v)$ for stream L.

Hence, the average number of probes for the whole join is:

$$JP = \lambda_L \lambda_R \sum_{v \in D} f_L(v) f_R(v)\left(t_{dis\_L}(v) + t_{dis\_R}(v)\right) \tag{22}$$

From (20) and (22), the total cost of the sampled join is:

$$Cost = (\lambda'_L + \lambda'_R)C_u + JP \cdot C_p \tag{23}$$

Using the above, we can compute the sampling fraction p.

Note that the above analysis is conservative, since it counts the cost of probing a tuple but not producing a join result as the cost of a complete probe/production. If we can assume that the cost of producing the join tuple dominates the cost of probing, then the production cost will only be proportional to the output rate. Since we are producing a random sample with parameter p, the output rate is $p \cdot \lambda_{o/p}$, where $\lambda_{o/p}$ is the un-sampled rate obtained by:

$$\lambda_{o/p} = \sum_{v \in D} f_L(v) f_R(v) \cdot \lambda_L \lambda_R \cdot (T_L + T_R) \tag{24}$$

We can then calculate the cost as:

$$Cost = (\lambda'_L + \lambda'_R)C_u + p \cdot \lambda_{o/p} \cdot C_p \tag{25}$$

5. Related Work

There are a number of systems recently developed for managing data streams, examples are Niagara [11], STREAM [20], Aurora [1], and Telegraph [19]. The survey in [7] contains a good documentation of earlier models and systems that are also targeted at such applications, together with a number of issues related to building a data stream management system.

The problem of load shedding for data streams has been discussed on a number of different levels. [16] talks about general approximation issues regarding streaming systems. For the memory limited case, the work in [4][5] discuss approximating aggregate queries. For join approximation, the work in [14] introduces the problem for both the CPU and memory limited settings, and suggests coin flipping strategies to solve it. The work in [12] discusses memory limited join approximation and provides an offline algorithm and online strategies for semantic load shedding for the Max-Subset goal. In this work, we do a similar analysis for the CPU limited case. The work in [17] defines two models for the stream inputs; the frequency and the age based models, and discusses semantic load shedding for streaming joins for the Max-Subset goal.

For the CPU limited case, the work in [6] discusses random load shedding techniques based on the coin flipping semantics and for join queries involving more than two input streams. The work in [8] is dealing with load shedding methods for aggregate queries and also uses random techniques. The only reference that explicitly deals with semantic load shedding methods for the CPU limited case is [18] in which a progressive strategy for adaptively shedding load with the increase of utilization is developed. In that work, they use a similar gain/loss utility metric to what we develop in this paper. However, it is not clear how their technique is applied for the join operator.

Random sampling is discussed in the context of relational database systems in [10]. [17] discusses random sampling for the streaming window join with memory limitations. This paper adapts the sampling techniques there to the CPU limited case.

6. Conclusions

In this work, we examined the problem of load shedding for streaming window joins under CPU limitations. We start by refining the CPU cost model previously proposed in the literature. Using the model, we formulated the load shedding problem in the static offline case, assuming the full result of the join is known and proved it is an NP-Hard problem. We proposed a greedy heuristic for the solving the offline problem and used it to guide the development of an online semantic load shedding strategy for the CPU limited scenario. For random load shedding, we proposed a number of strategies that decouples the decision of inserting and probing tuples for the symmetric hash join. We analyzed the new strategies and proved that one of them – the Insert-No-Probe alternative – is superior to the previously proposed random strategies. Finally, we adapted the UNIFORM algorithm proposed in [17] to produce a random sample of the join under CPU limitations.

Further experimental analysis of the proposed strategies is underway, in which we compare the effectiveness of the proposed semantic load shedding techniques and experimentally verify the utility of the Probe-No-Insert strategy. Also, we are investigating the tradeoffs between semantic and random load shedding to asses the point at which the extra load of maintaining the statistical information about the input outweighs its utility and develop methods to automatically account for it.

References

[1] D. Abadi, D. Carney, et al. Aurora: a new model and architecture for data stream management. The VLDB Journal, Vol.12(2), pp. 120 – 139, 2003.

[2] A. Arasu, B. Babcock, et al. Characterizing Memory Requirements for Queries over Continuous Data Streams. ACM PODS, June 2002.

[3] A. Arasu, S. Babu, J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report, Department of Computer Sciences, Stanford University, October 2003.

[4] A. Arasu, G. Manku. Approximate Counts and Quantiles over Sliding Windows. PODS, June 2004.

[5] A. Arasu, J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. VLDB September 2004.

[6] A. Ayad, J. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Stream. SIGMOD, June 2004.

[7] B. Babcock, S. Babu, et al. Models and Issues in Data Stream Systems. PODS, June 2002.

[8] B. Babcock, M. Datar, R. Motwani. Load Shedding for Aggregation Queries over Data Streams. ICDE 2004.

[9] S. Chandrasekaran, A. Deshpande, et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR, January 2003.

[10] S. Chaudri, R. Motwani, V. Narasaya. On random sampling over joins. SIGMOD, June 1999.

[11] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD, May 2000.

[12] A. Das, J. Gehrke, M. Riedewald, Approximate Join Processing Over Data Streams. SIGMOD, June 2003.

[13] M. Jaweed. Scalable Algorithms for Association Mining. TKDE Vol. 12, No. 3, June 2000

[14] J. Kang, J. F. Naughton, S. D. Viglas. Evaluating Window Joins over Unbounded Streams. ICDE 2003.

[15] S. Madden, M. Shah, et al. Continuously Adaptive Continuous Queries over Streams. SIGMOD, June 2002.

[16] R. Motwani, J. Widom, et al. Query Processing, Approximation, and Resource Management, in a Data Stream Management System. CIDR, January 2003.

[17] U. Srivastava, J. Widom. Memory-Limited Execution of Windowed Stream Joins. VLDB September 2004.

[18] N. Tatbul, U. Çetintemel, et al. Load Shedding in a Data Stream Manager. VLDB 2003.

[19] The Telegraph Project. http://telegraph.cs.berkeley.edu

[20] The Stanford Stream Data Manager. http://www-db.stanford.edu/stream.

[21] S. D. Viglas, J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources, SIGMOD, June 2002.

Appendix A

This appendix provides the proof for the statement in section 3.3.1 that the PNI strategy is the best choice for random online load shedding. We start by some helpful observations.

Note that, for all $(x_L,x_R)\in[0,1]\text{x}[0,1]$ we have

$$(T_L+T_R)x_Lx_R = T_Lx_Lx_R + T_Rx_Lx_R \le T_Lx_L + T_Rx_R \quad (26)$$

with the inequality strict unless both variables are 1.

Now, define

$$l(x_L, x_R) = (1 - (\lambda_L \cdot x_L + \lambda_R \cdot x_R)\cdot C_u)/C_p \quad (27)$$

$$\Omega_{CF} = \left\{ \begin{array}{l} (x_L, x_R)\in[0,1]\times[0,1] \mid \\ (\lambda_L x_L + \lambda_R x_R)C_u + \lambda_L\lambda_R(T_L+T_R)x_Lx_RC_p = 1 \end{array} \right\} \quad (28)$$

$$\Omega_{PNI} = \left\{ \begin{array}{l} (x_L, x_R)\in[0,1]\times[0,1] \mid \\ (\lambda_L x_L + \lambda_R x_R)C_u + \lambda_L\lambda_R(T_Lx_L+T_Rx_R)C_p = 1 \end{array} \right\} \quad (29)$$

Lemma 2

Let $(x_L,x_R) \in \Omega_{CF}$ such that at least one of x_L and x_R is strictly less than 1. Then there are $(x_L',x_R')\in[0,1]\text{x}[0,1]$ with $x_L' \le x_L$ and $x_R' \le x_R$, with at least one of the inequalities strict, such that $(x_L',x_R') \in \Omega_{PNI}$.

Proof

Start by setting $x_L' = x_L$ and $x_R' = x_R$, and allow one or both of x_L' and x_R' to decrease until the equality

$$(\lambda_L x_L' + \lambda_R x_R')C_u + \lambda_L\lambda_R(T_Lx_L' + T_Rx_R')C_p = 1 \quad (30)$$

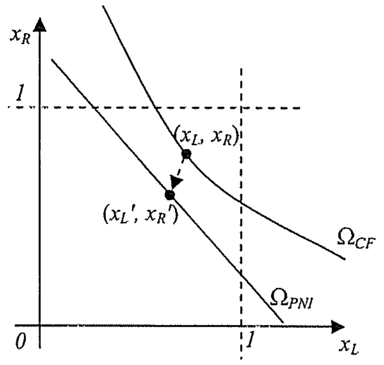

Figure 3. Pictorial depiction of Lemma 2

is satisfied. We know such decrease is always possible since by assumption that $(x_L,x_R) \in \Omega_{CF}$, and using equation (26), we know that the left hand side of the above equality is strictly greater than 1 for (x_L,x_R). ❑

Figure 3 pictorially depicts the result of the above lemma. We are now ready to prove the theorem.

Proof of Theorem 3

In case the join is feasible, we can easily verify that the optimum objective of the two techniques is the same as the output rate of the join and is obtained at both variables being 1.

The interesting case is when the join is infeasible. In this case, with an argument similar to the proof of Theorem 2, the first constraint is tight and at least one variable is less than 1. Hence, we can rewrite both problems as follows:

CF:

$$\textbf{Max } l(x_L, x_R) \quad \text{s.t. } (x_L, x_R) \in \Omega_{\text{CF}} \tag{31}$$

PNI:

$$\textbf{Max } l(x_L', x_R') \quad \text{s.t. } (x_L', x_R') \in \Omega_{\text{PNI}} \tag{32}$$

Consider the optimum solution (x_L, x_R) of CF in this case, $(x_L, x_R) \in \Omega_{\text{CF}}$ and $l(x_L, x_R)$ is the objective. By Lemma 2, there is $(x_L',x_R') \in \Omega_{PNI}$ such that $x_L' \leq x_L$ and $x_R' \leq x_R$ with at least one of the inequalities strict. From (32), (x_L',x_R') is a feasible solution for PNI. Hence, $l(x_L',x_R')$ is a lower bound on the optimum solution of PNI. But from (27), $l(x_L',x_R') > l(x_L, x_R)$. Therefore, the optimum solution of PNI is strictly greater than $l(x_L, x_R)$, which proves the theorem. ❑