RC23684 (W0508-015) August 2, 2005
Computer Science

IBM Research Report

Adaptive Load Shedding for Windowed Stream Joins

Bugra Gedik*, Kun-Lung Wu, Philip S. Yu, Ling Liu*
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*College of Computing
Georgia Institute of Technology

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Adaptive Load Shedding for Windowed Stream Joins

Buğra Gedik
College of Computing
Georgia Institute of Technology
bgedik@cc.gatech.edu

Kun-Lung Wu, Philip S. Yu
T.J. Watson Research Center
Hawthorne, NY 10532
{klwu, psyu}@us.ibm.com

Ling Liu
College of Computing
Georgia Institute of Technology
lingliu@cc.gatech.edu

Abstract

We present an adaptive load shedding approach for windowed stream joins. In contrast to the conventional approach of dropping tuples from the input streams, we explore the concept of *selective processing* for load shedding. We allow stream tuples to be stored in the windows and shed excessive CPU load by performing the join operations, not on the entire set of tuples within the windows, but on a dynamically changing subset of tuples that are learned to be highly beneficial. We support such dynamic selective processing through three forms of runtime *adaptations*: adaptation to input stream rates, adaptation to time correlation between the streams and adaptation to join directions. Indexes are used to further speed up the execution of stream joins. Experiments are conducted to evaluate our adaptive load shedding in terms of output rate. The results show that our selective processing approach to load shedding is very effective and significantly outperforms the approach that drops tuples from the input streams.

1 Introduction

With the ever increasing rate of digital information available from on-line sources and networked sensing devices [14], the management of bursty and unpredictable data streams has become a challenging problem. It requires solutions that will enable applications to effectively access and extract information from such data streams. A promising solution for this problem is to use declarative query processing engines specialized for handling data streams, such as data stream management systems (DSMS), exemplified by Aurora [3], STREAM [1], and TelegraphCQ [5].

Joins are key operations in any type of query processing engine and are becoming more important with the increasing need for fusing data from various types of sensors available, such as environmental, traffic, and network sensors. Here, we list some real-life applications of stream joins. We will return to these examples when we discuss assumptions about the characteristics of the joined streams.

– *Finding similar news items from two different sources*: Assuming that news items from CNN and Reuters are represented by weighted keywords (join attribute) in their respective streams, we can perform a windowed inner product join to find similar news items.

– *Finding correlation between phone calls and stock trading*: Assuming that phone call streams are represented as $\{\ldots, (P_a, P_b, t_1), \ldots\}$ where (P_a, P_b, t_1) means P_a calls P_b at time t_1, and stock trading streams are represented as $\{\ldots, (P_b, S_x, t_2), \ldots\}$ where (P_b, S_x, t_2) means P_b trades S_x at time t_2; we can perform a windowed equi-join on person to find hints, such as: P_a hints S_x to P_b in the phone call.

– *Finding correlated attacks from two different streams*: Assuming that alerts from two different sources are represented by tuples in the form of (*source*, *target*, {*attack descriptors*}, *time*) in their respective streams, we can perform a windowed overlap join on attack descriptors to find correlated attacks.

Recently, performing joins on unbounded data streams has been actively studied [9, 12, 10]. This is mainly due to the fact that traditional join algorithms need to perform a scan on one of the inputs to produce all the result tuples that match with a given tuple from the other input. However, data streams are unbounded. Producing a complete answer for a stream join requires unbounded memory and processing resources. To address this problem, several approaches have been proposed.

One natural way of handling joins on infinite streams is to use sliding windows. In a windowed stream join, a tuple from one stream is joined with only the tuples currently available in the window of another stream. A sliding window can be defined as a *time-based* or *count-based* window. An example of a time-based window is "last 10 seconds' tuples" and an example of a count-based window is "last 100 tuples." Windows can be either user defined, in which case we have *fixed* windows, or system-defined and thus *flexible*, in which case the system uses the available memory to maximize the output size of the join. Another way of handling the problem of blocking joins is to use *punctuated streams* [19], in which punctuations that give hints about the rest of the stream are used to prevent blocking. The two-way stream joins with user defined time-based windows constitute one of the most common join types in the data stream management research to date [2, 9, 12].

In order to keep up with the incoming rates of streams, CPU load shedding is usually needed in stream processing systems. Several factors may contribute to the demand for CPU load shedding, including (a) bursty and unpredictable rates of the incoming streams; (b) large window sizes; and (c) costly join conditions. Data streams can be unpredictable in nature [13] and incoming stream rates tend to soar during peak times. A high stream rate requires more resources for performing a windowed join, due to both increased number of tuples received per unit time and the increased number of tuples within a fixed-sized time window. Similarly, large window sizes imply that more tuples are needed for processing a windowed join. Costly join conditions typically require more CPU time.

In this paper, we present an adaptive CPU load shedding approach for windowed stream joins, aiming at maximizing the *output rate* of stream joins. The proposed approach is applicable to all kinds of join conditions, ranging from simple conditions such as equi-joins defined over single-valued attributes (e.g., the phone calls and stock trading scenario) to complex conditions such as those defined over set-valued attributes (e.g., the correlated attacks scenario) or weighted set-valued attributes (e.g., the similar news items scenario).

Summary of Contributions

Our adaptive load shedding approach has several unique characteristics.

First, instead of dropping tuples from the input streams as proposed in many existing approaches, our adaptive load shedding framework follows a selective processing methodology by keeping tuples within the windows, but processing them against a subset of the tuples in the opposite window.

Second, our approach achieves effective load shedding by properly adapting join operations to three dynamic stream properties: (i) incoming stream rates, (ii) time correlation between streams and (iii) join directions. The amount of selective processing is adjusted according to the incoming stream rates. Prioritized basic windows are used to adapt join operations to the time-based correlation between the input streams. Partial symmetric joins are dynamically employed to take advantage of the most beneficial join direction learned from the streams.

We employ indexes to speed up the selective processing of joins. Experiments were conducted to evaluate the effectiveness of our adaptive load shedding approach. Our experimental results show that the three adaptations can effectively shed the load in the presence of any of the following; bursty and unpredictable rates of the incoming streams, large window sizes, or costly join conditions.

Although our load shedding approach aims at maximization of output rate, utility-based load shedding can be coherently integrated with our adaptive framework (for this extension see our technical report [7]).

2 Related Work

Based on the metric being optimized, related work on load shedding in windowed stream joins can be divided into two categories.

The work in the first category aims at maximizing the utility of the output produced. Different tuples may have different importance values based on the application. For instance, in the news join example, certain type of news, e.g., security news, may be of higher value, and similarly in the stock trading example, phone calls from insiders may be of higher interest when compared to calls from regulars. In this case, an output from the join operator that contains highly-valued tuples is more preferable to a higher rate output generated from lesser-valued tuples. The work presented in [18] uses user-specified utility specifications to drop tuples from the input streams with low utility values. We refer to this type of load shedding as *utility based load shedding*, also referred to as *semantic load shedding* in the literature.

The work in the second category aims at maximizing the number of output tuples produced [6, 12, 17]. This can be achieved through rate reduction on the source streams, i.e., dropping tuples from the input streams, as suggested in [4, 12]. The work presented in [12] investigates algorithms for evaluating moving window joins over pairs of unbounded streams. Although the main focus of [12] is not on load shedding, scenarios where system resources are insufficient to keep up with the input streams are also considered.

In summary, in the context of windowed stream joins most of the existing techniques used for shedding load are tuple dropping for CPU-limited scenarios and memory allocation among windows for memory-limited scenarios. However, dropping tuples from the input streams without paying attention to the selectivity of such tuples may result in a suboptimal solution. Based on this observation, heuristics that take into account selectivity of the tuples are proposed in [6].

A different approach, called *age-based* load shedding, is proposed recently in [17] for performing memory-limited stream joins. This work is based on the observation that there exists a time-based correlation between the streams. Concretely, the probability of having a match between a tuple just received from one stream and a tuple residing in the window of the opposite stream, may change based on the difference between the timestamps of the tuples (assuming timestamps are assigned based on the arrival times of the tuples at the query engine). Under this observation, memory is conserved by keeping a tuple in the window since its reception until the average rate of output tuples generated using this tuple reaches its maximum value. For instance, in Figure 1 case I, the tuples can be kept in the window until they reach the vertical line marked. This effectively cuts down the memory needed to store the tuples within the window and yet produces an output close to the actual output without window reduction.

Obviously, knowing the distribution of the incoming streams has its peak at the beginning of the window, the age-based window reduction can be effective for shedding memory load. A natural question to ask is: "Can the age-based window reduction approach of [17] be used to shed CPU load?" This is a valid question, because reducing the window size also decreases the number of comparisons that have to be made in order to evaluate the join. However, as illustrated in Figure 1 case II, this technique cannot directly extend to the CPU-limited case where the memory is not the constraint. When the distribution does not have its peak close to the beginning of the window, the window reduction approach has to keep tuples until they are close to the end of the window. As a result, tuples that are close to the beginning of the window and thus are not contributing much to the output will be processed until the peak is reached close to the end of the window. This observation points out two important facts. First, time-based correlation between the windowed streams can play an important role in load shedding. Second, the window reduction technique that is effective for utilizing time-based correlation to shed memory load is not suitable for CPU load shedding, especially when the distribution of the incoming streams is unknown or unpredictable.

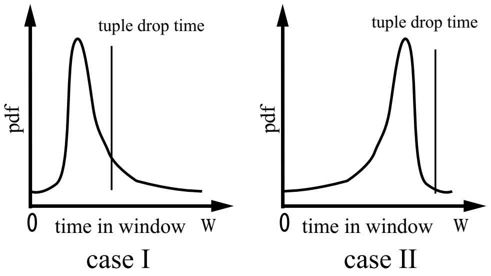

Figure 1: Examples of match probability density functions

3 Overview

Unlike the conventional load shedding approach of dropping tuples from the input streams, our adaptive load shedding encourages stream tuples to be kept in the windows. It sheds the CPU load by performing the stream joins on a dynamically changing subset of tuples that are learned to be highly beneficial, instead of on the entire set of tuples stored within the windows. This allows us to exploit the characteristics of stream applications that exhibit time-based correlation between the streams. Concretely, we assume that there exists a non-flat distribution of probability of match between a newly-received tuple and the other tuples in the opposite window, depending on the difference between the timestamps of the tuples.

There are several reasons behind this assumption. First, variable delays can exist between the streams as a result of differences between the communication overhead of receiving tuples from different sources [16]. Second and more importantly, there may exist variable delays between related events from different sources. For instance, in the news join example, different news agencies are expected to have different reaction times due to differences in their news collection and publishing processes. In the stock trading example, there will be a time delay between the phone call containing the hint and the action of buying the hinted stock. In the correlated attacks example, different parts of the network may have been attacked at different times. Note that, the effects of time correlation on the data stream joins are to some extent analogous to the effects of the time of data creation in data warehouses, which are exploited by join algorithms such as Drag-Join [11].

Although our load shedding is based on the assumption that the memory resource is sufficient, we want to point out two important observations. First, with increasing input stream rates and larger stream window sizes, it is quite common that CPU becomes limited before memory does. Second, even under limited memory, our adaptive load shedding approach can be used to effectively shed the excessive CPU load after window reduction is performed for handling the memory constraints.

3.1 Technical Highlights

Our load shedding approach is best understood through its two core mechanisms, each answering a fundamental question on adaptive load shedding without tuple dropping.

The first is called *partial processing* and it answers the question of "*how much we can process*" given a window of stream tuples. The factors to be considered in answering this question include the performance of the stream join operation under current system load and the current incoming stream rates. In particular, partial processing dynamically adjusts the amount of load shedding to be performed through rate adaptation.

The second is called *selective processing* and it answers the question of "*what should we process*" given the constraint on the amount of processing, defined at the partial processing phase. The factors that influence the answer to this question include the characteristics of stream window segments, the profitability of join directions, and the utility of different stream tuples. Selective processing extends partial processing to intelligently select the tuples to be used during join processing under heavy system load, with the goal of maximizing the output rate of the stream join.

Before describing the details of partial processing and selective processing, we first briefly review the basic concepts involved in processing windowed stream joins, and establish the notations that will be used throughout the paper.

3.2 Basic Concepts and Notations

A two-way windowed stream join operation takes two input streams denoted as S_1 and S_2, performs the stream join and generates the output. For notational convenience, we denote the opposite stream of stream i $(i = 1, 2)$ as stream $\overline{i}$. The sliding window defined over stream S_i is denoted as W_i, and has size w_i in terms of seconds. We denote a tuple as t and its arrival timestamp as $T(t)$. Other notations will be introduced in the rest of the paper as needed. Table 1 summarizes the notations used throughout the paper.

Notation	Meaning
t	tuple
$T(t)$	timestamp of the tuple t
S_i	input stream i
W_i	window over S_i
w_i	window size of W_i in seconds
λ_i	rate of S_i in tuples per second
$B_{i,j}$	basic window j in W_i
b	basic window size in seconds
n_i	number of basic windows in W_i
r	fraction parameter
δ_r	fraction boost factor
r_i	fraction parameter for W_i
$f_i(.)$	match probability density function for W_i
$p_{i,j}$	probability of match for $B_{i,j}$
$o_{i,j}$	expected output from comparing a tuple t with a tuple in $B_{i,j}$
s_i^j	k, where $o_{i,k}$ is the jth item in the sorted list $\{o_{i,l} \mid l \in [1..n_i]\}$
T_r	rate adaptation period
T_c	time correlation adaptation period
γ	sampling probability

Table 1: Notations used throughout the paper

A windowed stream join is performed by fetching tuples from the input streams and processing them against tuples in the opposite window. For a newly fetched tuple t from stream S_i, the join is performed in the following three steps.

First, tuple t is inserted into the beginning of window W_i. Second, tuples at the end of window $W_{\overline{i}}$ are checked in order and removed if they have expired. A tuple t_o expires from window $W_{\overline{i}}$ iff $T - T(t_o) > w_{\overline{i}}$, where T represents the current time. The expiration check stops when an unexpired tuple is encountered. The tuples in window W_i are sorted in the order of their arrival timestamps by default and the window is managed as a doubly linked list for efficiently performing insertion and expiration operations. In the third and last step, tuple t is processed against tuples in the window $W_{\overline{i}}$, and matching tuples are generated as output. Figure 2 summarizes the join processing steps.

```
JOIN PROCESSING()
for $i = 1$ to $2$
  if no tuple in $S_i$
    continue
  $t \leftarrow$ fetch tuple from $S_i$
  Insert $t$ in front of $W_i$
  repeat
    $t_o \leftarrow$ last tuple in $W_{\overline{i}}$
    if $T - T(t_o) > w_{\overline{i}}$
      Remove $t_o$ from $W_{\overline{i}}$
  until $T - T(t_o) \leq w_{\overline{i}}$
  foreach $t_a \in W_{\overline{i}}$
    Eval. join cond. on $t$, $t_a$
```

Figure 2: Join Processing

4 Partial Processing - How Much Can We Process?

The first step in our approach to shedding CPU load without dropping tuples is to determine how much we can process given the windows of stream tuples that participate in the join. We call this step the partial processing based load shedding. For instance, consider a scenario in which the limitation in processing power requires dropping half of the tuples, i.e. decreasing the input rate of the streams by half. A partial processing approach is to allow every tuple to enter into the windows, but to decrease the cost of join processing by comparing a newly-fetched tuple with only a fraction of the window defined on the opposite stream.

Partial processing, by itself, does not significantly increase the number of output tuples produced by the join operator, when compared to tuple dropping or window reduction approaches. However, as we will describe later in the paper, it forms a basis to perform selective processing, which exploits the time-based correlation between the streams, and makes it possible to accommodate utility-based load shedding, in order to maximize the output rate or the utility of the output tuples produced. (see [7] for output utility maximization)

Two important factors are considered in determining the amount of partial processing: (1) the current incoming stream rates, and (2) the performance of the stream join operation under current system load. Partial processing employs rate adaptation to adjust the amount of processing performed dynamically. The performance of the stream join under the current system load is a critical factor and it is influenced by the join algorithm and optimizations used for performing join operations.

In the rest of this section, we first describe rate adaptation, then discuss the details of utilizing indexes for efficient join processing. Finally we describe how to employ rate adaptation in conjunction with indexed join processing.

4.1 Rate Adaptation

The partial processing-based load shedding is performed by adapting to the rates of the input streams. This is done by observing the tuple consumption rate of the join operation and comparing it to the input rates of the streams to determine the fraction of the windows to be processed. This adaptation is performed periodically, at every T_r seconds. T_r is called the *adaptation period*. We denote the fraction parameter as r, which defines the ratio of the windows to be processed. In other words, the setting of r answers the question of *how much* load we should shed.

```
RateAdapt()
Initially: $r \leftarrow 1$
every $T_r$ seconds
    $\alpha_1 \leftarrow$ # of tuples fetched from $S_1$ since last adapt.
    $\alpha_2 \leftarrow$ # of tuples fetched from $S_2$ since last adapt.
    $\lambda_1 \leftarrow$ average rate of $S_1$ since last adaptation
    $\lambda_2 \leftarrow$ average rate of $S_2$ since last adaptation
    $\beta \leftarrow \frac{\alpha_1+\alpha_2}{(\lambda_1+\lambda_2)*T_r}$
    if $\beta < 1$ then $r \leftarrow \beta * r$
            else $r \leftarrow min(1, \delta_r * r)$
```

Algorithm 1: Rate Adaptation

Algorithm 1 gives a sketch of the rate adaptation process. Initially, the fraction parameter r is set to 1. Every T_r seconds, the average rates of the input streams S_1 and S_2 are determined as λ_1 and λ_2. Similarly, the number of tuples fetched from streams S_1 and S_2 since the last adaptation step are determined as α_1 and α_2. Tuples from the input streams may not be fetched at the rate they arrive due to an inappropriate initial value of the parameter r or due to a change in the stream rates since the last adaptation step. As a result, $\beta = \frac{\alpha_1+\alpha_2}{(\lambda_1+\lambda_2)*T_r}$ determines the percentage of the input tuples fetched by the join algorithm. Based on the value of β, the fraction parameter r is readjusted at the end of each adaptation step. If β is smaller than 1, r is multiplied by β, with the assumption that comparing a tuple with the other tuples in the opposite window has the dominating cost in join processing. Otherwise, the join is able to process all the incoming tuples with the current value of r. In this case, the r value is set to $min(1, \delta_r * r)$, where δ_r is called the *fraction boost factor*. This is aimed at increasing the fraction of the windows processed, optimistically assuming that additional processing power is available. If not, the parameter r will be decreased during the next adaptation step. Higher values of the fraction boost factor result in being more aggressive at increasing the parameter r. The adaptation period T_r should be small enough to adapt to the bursty nature of the streams, but large enough not to cause overhead and undermine the join processing.

4.2 Indexed Join and Partial Processing

Stream indexing [8, 20] can be used to cope up with the high processing cost of the join operation, reducing the amount of load shedding performed. However, there are two important points to be resolved before indexing can be employed together with partial processing and thus with other algorithms we introduce in the following sections. The first issue is that, in a streaming scenario the index has to be maintained dynamically (through insertions and removals) as the tuples enter and leave the window. This means that the assumption made in Section 4.1 about finding matching tuples within a window (index search cost) being the dominant cost in the join processing, no longer holds. Second, the index does not naturally allow processing only a certain portion of the window. We resolve these issues in the context of inverted indexes, that are predominantly used for joins based on set or weighted set-valued attributes. The same ideas apply to hash-indexes used for equi-joins on single-valued attributes. Our inverted-index implementation reduces to a hash-index in the presence of single-valued attributes.

4.2.1 Inverted Indexes

An inverted index consists of a collection of sorted identifier lists. In order to insert a set into the index, for each item in the set, the unique identifier of the set is inserted into the identifier list associated with that particular item. Similar to insertion, removal of a set from the index requires finding the identifier lists associated with the items in the set. The removal is performed by removing the identifier of the set from these identifier lists. In our context, the inverted index is maintained as an in-memory data structure. The collection of identifier lists are managed in a hash table. The hash table is used to efficiently find the identifier list associated with an item. The identifier lists are internally organized as sorted (based on unique set identifiers) balanced binary trees to facilitate both fast insertion and removal. The set identifiers are in fact pointers to the tuples they represent. Query processing on an inverted index follows a multi-way merging process, which is usually accelerated through the use of a heap [15].

4.2.2 Time Ordered Identifier Lists

Although the usage of inverted indexes speeds up the processing of joins based on set-valued attributes, it also introduces significant insertion and deletion costs. This problem can be alleviated by exploiting the timestamps of the tuples that are being indexed and the fact that these tuples are received in timestamp order from the input streams. In particular, instead of maintaining identifier lists as balanced trees sorted on identifiers, we can maintain them as linked lists sorted on timestamps of the tuples (sets). This does not effect the merging phase of the indexed search, since a timestamp uniquely identifies a tuple in a stream unless different tuples with equal timestamps are allowed. In order to handle the latter, the identifier lists can be sorted based on (timestamp, identifier) pairs.

5 Selective Processing - What Should We Process?

Selective processing extends partial processing to intelligently select the tuples to be used during join processing under heavy system load. Given the constraint on the amount of processing defined at the partial processing phase, the selective processing aims at maximizing the output rate or the output utility of the stream joins. Two important factors are used to determine what we should select for join processing: (1) the characteristics of stream window segments and (2) the profitability of join directions. We describe time correlation adaptation and join direction adaptation, which form the core of our selective processing approach. The main ideas are to prioritize segments (basic windows) of the windows in order to process parts that will yield higher output (time correlation adaptation) and to start load shedding from one of the windows if one direction of the join is producing more output than the other (join direction adaptation).

5.1 Time Correlation Adaptation

For the purpose of time correlation adaptation, we divide the windows of the join into *basic windows.* Concretely, window W_i is divided into n_i basic windows of size b seconds each, where $n_i = 1 + \lceil w_i/b \rceil$. $B_{i,j}$ denotes the jth basic window in W_i, $j \in [1..n_i]$. Tuples do not move from one basic window to another. As a result, tuples leave the join operator one basic window at a time and the basic windows slide discretely b seconds at a time. The newly fetched tuples are inserted into the first basic window. When the first basic window is full, meaning that the newly fetched tuple has a timestamp that is at least

b seconds larger than the oldest tuple in the first basic window, the last basic window is emptied and all the basic windows are shifted, last basic window becoming the first. The newly fetched tuples can now flow into the new first basic window, which is empty. The basic windows are managed in a circular buffer, so that the shift of windows is a constant time operation. The basic windows themselves can be organized as linked lists (if no indexing is used) or as inverted/hashed indexes (if indexing is used).

Time correlation adaptation is periodically performed at every T_c seconds. T_c is called the *time correlation adaptation period*. During the time between two consecutive adaptation steps, the join operation performs two types of processing. For a newly fetched tuple, it either performs *selective processing* or *full processing*. Selective processing is carried out by looking for matches with tuples in high priority basic windows of the opposite window, where the number of basic windows used depends on the amount of load shedding to be performed. Full processing is done by comparing the newly fetched tuple against all the tuples from the opposite window. The aim of full processing is to collect statistics about the usefulness of the basic windows for the join operation.

```
TimeCorrelationAdapt()
(1) every $T_c$ seconds
(2)    for $i = 1$ to 2
(3)        sort in desc. order $\{\hat{o}_{i,j} | j \in [1..n_i]\}$ into array $O$
(4)        for $j = 1$ to $n_i$
(5)            $o_{i,j} \leftarrow \frac{\hat{o}_{i,j}}{\gamma * r * b * \lambda_2 * \lambda_1 * T_c}$
(6)            $s_i^j \leftarrow k$, where $O[j] = \hat{o}_{i,k}$
(7)        for $j = 1$ to $n_i$
(8)            $\hat{o}_{i,j} \leftarrow 0$
```

Algorithm 2: Time Correlation Adaptation

```
ProcessTuple()
(1) when processing tuple $t$ against window $W_i$
(2)   if $rand < r * \gamma$
(3)       process $t$ against all tuples in $B_{i,j}, \forall j \in [1..n_i]$
(4)       foreach match in $B_{i,j}, \forall j \in [1..n_i]$
(5)           $\hat{o}_{i,j} \leftarrow \hat{o}_{i,j} + 1$
(6)   else
(7)       $a \leftarrow r * |W_i|$
(8)       for $j = 1$ to $n_i$
(9)           $a \leftarrow a - |B_{i,s_i^j}|$
(10)          if $a > 0$
(11)              process $t$ against all tuples in $B_{i,s_i^j}$
(12)          else
(13)              $r_e \leftarrow 1 + \frac{a}{|B_{i,s_i^j}|}$
(14)              process $t$ against $r_e$ fraction of $B_{i,s_i^j}$
(15)              break
```

Algorithm 3: Tuple Processing with Time Correlation Adaptation

The details of the adaptation step and full processing are given in Algorithm 2 and in lines 1-5 of Algorithm 3. Full processing is only done for a sampled subset of the stream, based on a parameter called *sampling probability*, denoted as γ. A newly fetched tuple goes through selective processing with probability $1 - r * \gamma$. In other words, it goes through full processing with probability $r * \gamma$. The fraction parameter r is used to scale the sampling probability, so that the full processing does not consume all processing resources when the load on the system is high. The goal of full processing is to calculate for each basic window $B_{i,j}$, the expected number of output tuples produced from comparing a newly fetched tuple t with a tuple in $B_{i,j}$, denoted as $o_{i,j}$. These values are used later during the adaptation step to prioritize windows. In particular, $o_{i,j}$ values are used to calculate s_i^j values. Concretely, we have:

$$s_i^j = k, \text{ where } o_{i,k} \text{ is the } j\text{th item in the sorted list } \{o_{i,l} | l \in [1..n_i]\}$$

This means that B_{i,s_i^1} is the highest priority basic window in W_i, B_{i,s_i^2} is the next, and so on.

Lines 7-14 in Algorithm 3 give a sketch of selective processing. During selective processing, s_i^j values are used to guide the load shedding. Concretely, in order to process a newly fetched tuple t against window W_i, first the number of tuples from window W_i, that are going to be considered for processing, is determined by calculating $r * |W_i|$, where $|W_i|$ denotes the number of tuples in the window. The fraction parameter r is determined by rate adaptation as described in Section 4.1. Then, tuple t is processed against basic windows, starting from the highest priority one, i.e. B_{i,s_i^1}, going in decreasing order of priority. A basic window B_{i,s_i^j} is searched for matches completely, if adding $|B_{i,s_i^j}|$ number of tuples to the number of tuples used so far from window W_i to process tuple t does not exceeds $r * |W_i|$. Otherwise an appropriate fraction of the basic window is used and the processing is completed for tuple t.

5.2 Join Direction Adaptation

Due to time-based correlation between the streams, a newly fetched tuple from stream S_1 may match with a tuple from stream S_2 that has already made its way into the middle portions of window W_2. This means that, most of the time, a newly fetched tuple from stream S_2 has to stay within the window W_2 for some time, before it can be matched with a tuple from stream S_1. This implies that, one direction of the join processing may be of lesser value, in terms of the number of output tuples produced, than the other direction. For instance, in the running example, processing a newly fetched tuple t from stream S_2 against window W_1 will produce smaller number of output tuples when compared to the other way around, as the tuples to match t has not yet arrived at window W_1. In this case, symmetry of the join operation can be broken during load shedding, in order to achieve a higher output rate. This can be achieved by decreasing the fraction of tuples processed from window W_2 first, and from W_1 later (if needed). We call this *join direction adaptation.*

```
JoinDirectionAdapt()
Initially: $r_1 \leftarrow 1, r_2 \leftarrow 1$
**upon** completion of RateAdapt() call
    $o_1 \leftarrow \frac{1}{n_1} * \sum_{j=1}^{n_1} o_{1,j}$
    $o_2 \leftarrow \frac{1}{n_2} * \sum_{j=1}^{n_2} o_{2,j}$
    **if** $o_1 \geq o_2$ **then** $r_1 \leftarrow min(1, r * \frac{w_1+w_2}{w_1})$
                  **else** $r_1 \leftarrow max(0, r * \frac{w_1+w_2}{w_1} - \frac{w_2}{w_1})$
    $r_2 \leftarrow r * \frac{w_1+w_2}{w_1} - r_1 * \frac{w_1}{w_2}$
```

Algorithm 4: Join Direction Adaptation

Join direction adaptation is performed immediately after rate adaptation. Specifically, two different fraction parameters are defined, denoted as r_i for window W_i, $i \in \{1, 2\}$. During join processing, r_i fraction of the tuples in window W_i are considered, making it possible to adjust join direction by changing r_1 and r_2. This requires replacing r with r_i in line 7 of Algorithm 3 and line 5 of Algorithm 2.

The constraint in setting of r_i values is that, the number of tuple comparisons performed per time unit should stay the same when compared to the case where there is a single r value as computed by Algorithm 1. The number of tuple comparisons performed per time unit is given by $\sum_{i=1}^{2} (r_i * \lambda_{\bar{i}} * (\lambda_i * w_i))$, since the number of tuples in window W_i is $\lambda_i * w_i$. Thus, we should have $\sum_{i=1}^{2} (r * \lambda_{\bar{i}} * (\lambda_i * w_i)) = \sum_{i=1}^{2} (r_i * \lambda_{\bar{i}} * (\lambda_i * w_i))$, i.e.:

$$r * (w_1 + w_2) \quad = \quad r_1 * w_1 + r_2 * w_2$$

The valuable direction of the join can be determined by comparing the expected number of output tuples produced from comparing a newly fetched tuple with a tuple in W_i, denoted as o_i, for $i = 1$ and 2. This can be computed as $o_i = \frac{1}{n_i} * \sum_{j=1}^{n_i} o_{i,j}$. Assuming $o_1 > o_2$, without loss of generality, we can set $r_1 = min(1, r * \frac{w_1+w_2}{w_1})$. This maximizes r_1, while respecting the above constraint. The generic procedure to set r_1 and r_2 is given in Algorithm 4.

Join direction adaptation, as it is described in this section, assumes that any portion of one of the windows is more valuable than all portions of the other window. This may not be the case for applications where both match probability distribution functions, $f_1(t)$ and $f_2(t)$, are non-flat. For instance, in a traffic application scenario, a two way traffic flow between two points implies both directions of the join are valuable. A more advanced join direction adaptation algorithm was introduced in [7], that can handle such cases, as part of utility-based load shedding.

6 Experiments

The adaptive load shedding algorithms presented in this paper have been implemented and successfully demonstrated as part of a large-scale stream processing prototype at IBM Watson Research. Here, we report two sets of experimental results to demonstrate their effectiveness. The first set demonstrates the performance of the partial processing-based load shedding step − keeping tuples within windows and shedding excessive load by partially processing the join through rate adaptation. The second set shows the performance gain in terms of output rate for selective processing, which incorporates time correlation adaptation and join direction adaptation. The effect of basic window size on the performance is also investigated experimentally. Note that the overhead cost associated with dynamic adaptation has been fully taken into account and it manifests itself in the output rate of the join operations. Hence, we do not separately show the overhead cost.

6.1 Experimental Setup

The join operation is implemented as a Java package, named `ssjoin.*`, and is customizable with respect to supported features, such as rate adaptation, time correlation adaptation and join direction adaptation, as well as various parameters associated with these features. Streams used in the experiments reported in this section are timestamp ordered tuples, where each tuple includes a single attribute, that can either be a set, weighted set, or a single value. The sets are composed of variable number of items, where each item is an integer in the range $[1..L]$. L is taken as 100 in the experiments. Number of items contained in sets follow a normal distribution with mean μ and standard deviation σ. In the experiments, μ is taken as 5 and σ is taken as 1. The popularity of items in terms of how frequently they occur in a set, follows a Zipf distribution with parameter κ. For equi-joins on single-valued attributes, L is taken as 5000 with $\mu = 1$ and $\sigma = 0$.

The time-based correlation between streams is modeled using two parameters, *time shift* parameter denoted as τ and *cycle period* parameter denoted as ς. Cycle period is used to change the popularity ranks of items as a function of time. Initially at time 0, the most popular item is 1, the next 2, and so on. Later at time T, the most popular item is $a = 1 + \lfloor L * \frac{T \bmod \varsigma}{\varsigma} \rfloor$, the next $a + 1$, and so on. Time shift is used to introduce a delay between matching items from different streams. Applying a time shift of τ to one of the streams means that the most popular item is $a = 1 + \lfloor L * \frac{(T-\tau) \bmod \varsigma}{\varsigma} \rfloor$ at time T, for that stream.

Figure 3 shows the resulting probability of match distribution f_1, when a time delay of $\tau = \frac{5}{8} * \varsigma$ is applied to stream S_2 and $\varsigma = 2 * w$, where $w_1 = w_2 = w$. The two histograms represent two different scenarios, in which κ is taken as 0.6 and 0.8, respectively. These settings for τ and ς parameters are also used in the rest of the experiments, unless otherwise stated. We change the value of parameter κ to model varying amounts of skewness in match probability distributions. Experiments are performed using time varying stream rates and various window sizes.

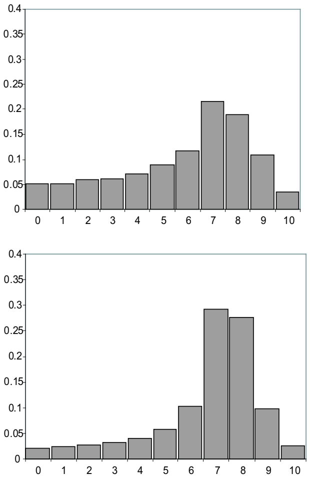

Figure 3: Probability match distributions, $\kappa = 0.6$ and $\kappa = 0.8$

The default settings of some of the system parameters are as follows: $T_r = 5$ seconds, $T_c = 5$ seconds, $\delta_r = 1.2$, $\gamma = 0.1$. We place input buffers of size 1 seconds in front of the inputs of the join operation. We report results from overlap and equality joins. The experiments are performed on an IBM PC with 512MB main memory and 2.4Ghz Intel Pentium4 processor, using Sun JDK 1.4.2.

For comparisons, we also implemented a random drop scheme. It performs load shedding by randomly dropping tuples from the input buffers and performing the join fully with the available tuples in the join windows. It is implemented separately from our selective join framework and does not include any overhead due to adaptations.

6.2 Rate Adaptation

We study the impact of rate adaptation on output rate of the join operation. For the purpose of the experiments in this subsection, time shift parameter is set to zero, i.e. $\tau = 0$, so that there is no time shift between the streams and the match probability decreases going from the beginning of the windows to the end. A non-indexed overlap join, with threshold value of 3 and 20 seconds window on one of the streams, is used.

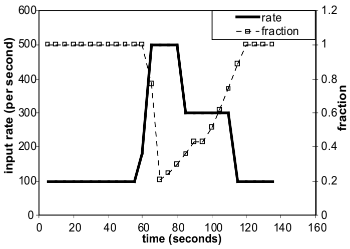

Figure 4: Stream rates and fraction parameter r

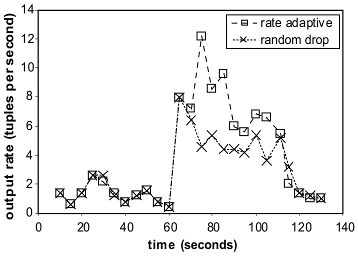

Figure 5: Improvement in output rate with rate adaptation

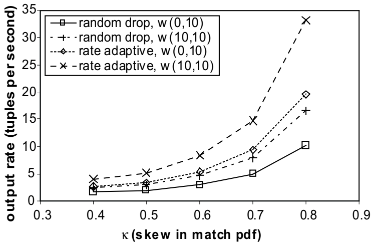

Figure 6: Improvement in average output rate with rate adaptation

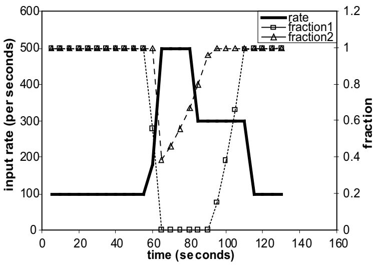

Figure 7: Stream rates and fraction parameters r_1 and r_2

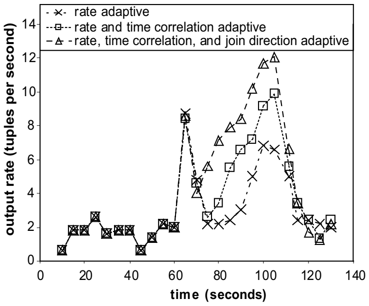

Figure 8: Improvement in output rate with time correlation and join direction adaptation

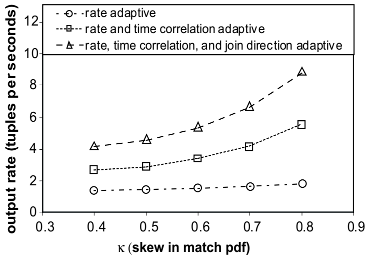

Figure 9: Improvement in average output rate with time correlation and join direction adaptation

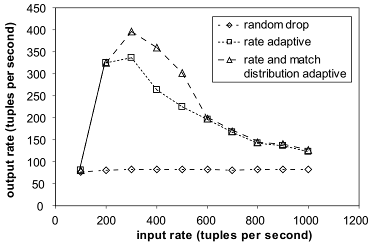

Figure 10: Improvement in average output rate for equi-joins on single-valued attributes

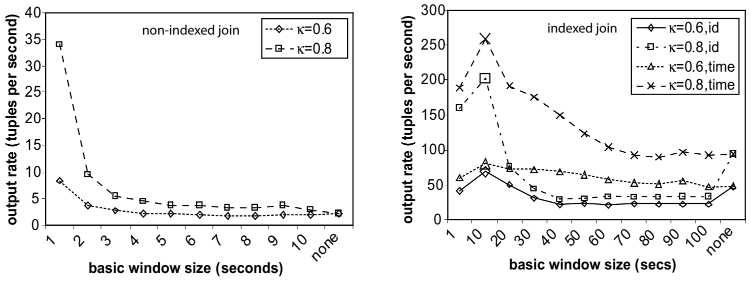

Figure 11: Impact of basic window size for indexed and non-indexed join, for various basic window sizes

Figure 4 shows the stream rates used (on the left y-axis) as a function of time. The rate of the streams stay at 100 tuples per second for around 60 seconds, then jump to 500 tuples per seconds for around 15 seconds and drop to 300 tuples per second for around 30 seconds before going back to its initial value. Figure 4 also shows (on the right y-axis) how fraction parameter r adapts to the changing stream rates.

The graphs in Figure 5 show the resulting stream output rates as a function of time with and without rate adaptation, respectively. No rate adaptation case represents random tuple dropping. It is observed that rate adaptation improves output rate when the stream rates increase. That is the time when tuple dropping starts for the non-adaptive case. The improvement is around 100% when stream rates are 500 tuples per second and around 50% when 300 tuples per second. The ability of rate adaptation to keep output rate high is mainly due to the time aligned nature of the streams. In this scenario, only the tuples that are closer to the beginning of the window are useful for generating matches and the partial processing uses the beginning part of the window, as dictated by the fraction parameter r.

The graphs in Figure 6 plot the average output rates of the join over the period shown in Figure 5 as a function of skewness parameter κ, for different window sizes. It shows that the improvement in output rate, provided by rate adaptation, increases not only with increasing skewness of the match probability distribution, but also with increasing window sizes. This is because, larger windows imply that more load shedding has to be performed.

6.3 Selective Processing

Here, we study the impact of time correlation adaptation and join direction adaptation on output rate of the join operation. For the purpose of the experiments in this subsection, time shift parameter is taken as $\tau = \frac{5}{8} * \varsigma$. A non-indexed overlap join, with threshold value of 3 and 20 seconds windows on both of the streams, is used. Basic window sizes on both windows are set to 1 second for time correlation adaptation.

Figure 7 shows the stream rates used (on the left y-axis) as a function of time. Figure 7 also shows (on the right y-axis) how fraction parameters r_1 and r_2 adapt to the changing stream rates with join direction adaptation. Note that the reduction in fraction parameter values start with the one (r_2 in this case) corresponding to the window that is less useful in terms of generating output tuples when processed against a newly fetched tuple from the other stream.

The graphs in Figure 8 show the resulting stream output rates as a function of time with three different join settings. It is observed that, when the stream rates increase, the time correlation adaptation combined with rate adaptation provides improvement on output rate (around 50%), when compared to rate adaptation only case. Moreover, applying join direction adaptation on top of time correlation adaptation provides additional improvement in output rate (around 40%).

The graphs in Figure 9 plot the average output rates of the join as a function of skewness parameter κ, for different join settings. This time, the overlap threshold is set to 4, which results in lower number of matching tuples. It is observed that the improvement in output rates, provided by time correlation and join direction adaptation, increase with increasing skewness in match probability distribution. The increasing skewness does not improve the performance of rate adaptive-only case, due to its lack of time correlation adaptation which in turn makes it unable to locate the productive portion of the window for processing, especially when the time lag τ is large and the fraction parameter r is small.

To strengthen and extend our observation that partial processing is superior to random dropping and that selective processing provides additional improvements in output rates on top of partial processing, in Figure 10 we compare random dropping to selective processing for equi-joins on single-valued attributes. The graphs in Figure 10 plot output rates of the join as a function of the input rates (from 100 tuples/sec to 1000 tuples/sec) for random dropping, rate adaptive, and rate and match distribution adaptive cases. The figure shows that selective processing with rate and match distribution adaptation provides up to **5 times** improvement over random dropping, and up to 35% improvement over rate adaptive-only case.

Note that the output rate first increases with increasing input rates and then shows a decrease with further increase in input rates. This is mainly due to the simulation setup, where workload generation takes increasingly more processing time with increasing input rates (similar observations are reported by others [12]).

6.3.1 Basic Window Size

We study the impact of basic window size on output rate of the join operation. The graphs in Figure 11 plot average join output rate as a function of basic window size, for different κ values. The graphs on the left represents a non-indexed overlap join, with threshold value of 3 and 20 seconds windows, respectively, on both of the streams. The graphs on the right represents an indexed overlap join, with threshold value of 3 and 200 seconds windows, respectively, on both of the streams. For the indexed case, both identifier sorted and time sorted inverted indexes are used. The "none" value on the x-axis of the graphs represent the case where basic windows are not used (note that this is *not* same as using a basic window equal in size to join window). For both experiments, a stream rate of 500 tuples per second is used.

As expected, small basic windows provide higher join output rates. However, there are two interesting observations for the indexed join case. First, for very small basic window sizes, we observe a drop in the output rate. This is due to the overhead of processing large number of basic windows with indexed join. In particular, the cost of looking up identifier lists for each basic window that is used for join processing, creates an overhead. Further decreasing basic window size does not help in better capturing the peak of the match probability distribution. Second, identifier sorted inverted indexes show significantly lower output rate, especially when the basic window sizes are high. This is because identifier sorted inverted indexes do not allow partial processing based on time.

7 Conclusion

We have presented an adaptive CPU load shedding approach for stream join operations. In particular, we showed how rate adaptation, combined with time-based correlation adaptation and join direction adaptation, can increase the number of output tuples produced by a join operation. Our load shedding algorithms employed a selective processing approach, as opposed to commonly used tuple dropping. Our experimental results showed that (a) our adaptive load shedding algorithms are very effective under varying input stream rates, varying CPU load conditions, and varying time correlations between the streams; and (b) our approach significantly outperforms the approach that randomly drops tuples from the input streams.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, March 2003.

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

[3] H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *VLDB Journal Special Issue on Data Stream Processing*, 2004.

[4] 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.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. 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.

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

[7] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. Technical Report GIT-CERCS-05-05, Georgia Institute of Technology, May 2005.

[8] L. Golab, S. Garg, and M. T. Ozsu. On indexing sliding windows over online data streams. In *EDBT*, 2004.

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

[10] M. A. Hammad and W. G. Aref. Stream window join: Tracking moving objects in sensor-network databases. In *Scientific and Statistical Database Management, SSDBM*, 2003.

[11] S. Helmer, T. Westmann, and G. Moerkotte. Diag-Join: An opportunistic join algorithm for 1:N relationships. In *VLDB*, 1998.

[12] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *IEEE ICDE*, 2003.

[13] J. Kleinberg. Bursty and hierarchical structure in streams. In *ACM SIGKDD*, 2002.

[14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *USENIX OSDI*, 2002.

[15] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *ACM SIGMOD*, 2003.

[16] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *ACM PODS*, 2004.

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

[18] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[19] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15, 2003.

[20] K.-L. Wu, S.-K. Chen, and P. S. Yu. Interval query indexing for efficient stream processing. In *ACM CIKM*, 2004.