RC24029 (W0608-058) August 16, 2006
Computer Science

IBM Research Report

CPU Load Shedding for Binary 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

*Georgia Institute of Technology

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. Our load shedding approach enables us to integrate *utility-based* load shedding with *time correlation-based* load shedding. 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 and utility. 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.

I. INTRODUCTION

With the ever increasing rate of digital information available from on-line sources and networked sensing devices [1], 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 [2], STREAM [3], and TelegraphCQ [4].

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, where $\{attack\ \ descriptors\}$ is a set-based attribute that includes keywords describing the attack, we can perform a windowed overlap join on attack descriptors to find correlated attacks from different sources.

Recently, performing joins on unbounded data streams has been actively studied [5], [6], [7]. 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* (tuple-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* [8], 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 [9], [5], [6].

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 [10] 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 both the *output rate* and the *output utility* 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 segments of the 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.

Third, but not the least, our selective processing approach enables a coherent integration of the three adaptations with the utility-based load shedding. Maximizing the utility of the output tuples produced is especially important when certain tuples are more valuable than others.

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.

II. 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 [11] 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 [12], [6], [13]. This can be achieved through rate reduction on the source streams, i.e., dropping tuples from the input streams, as suggested in [14], [6]. The work presented in [6] investigates algorithms for evaluating moving window joins over pairs of unbounded streams. Although the main focus of [6] is not on load shedding, scenarios where system resources are insufficient to keep up with the input streams are also considered.

There are several other works related to load shedding in DSMSs in general, including memory allocation among query operators [15] or inter-operator queues [16], load shedding for aggregation queries [17], and overload-sensitive management of archived streams [18].

In summary, 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 [12].

A different approach, called *age-based* load shedding, is proposed recently in [13] 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 [13] 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 streams is unknown or unpredictable.

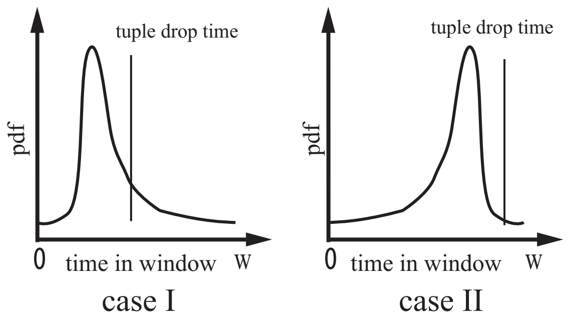


Fig. 1. Examples of match probability density functions

With the above analysis in mind, we propose an adaptive load shedding approach that is capable of performing selective processing of tuples in the stream windows by dynamic adaptation to input stream rates, time-based correlations between the streams, and profitability of different join directions. To the best of our knowledge, our load shedding approach is the first one that can handle arbitrary time correlations and at the same time support maximization of output utility.

Finally, it is worth mentioning that our windowed stream join operator semantics follow the input-triggered approach, in which the output of the join is a time ordered series of tuples, where an output tuple is generated whenever an incoming tuple matches with an existing tuple in the join windows. The same approach is used in most of the related work described in this section [5], [6], [7], [12], [13]. A different semantics for the windowed stream joins, one based on the negative tuples approach, is used in the Nile stream engine [19]. In Nile, at any time the answer of the join is taken as the answer of the snapshot join whose inputs are the tuples in the current window for each input stream. This is more similar to the problem of incremental materialized view maintenance. The extension of our selective processing approach to negative tuples is an interesting research direction. However, it is beyond the scope of this paper.

III. 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 [20]. 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

Diag-Join [21]. [1]

It is important to note that we do not assume pre-knowledge of the probability match distributions corresponding to the time correlations. Instead, they are learned by our join algorithms. However, we do assume that these distributions are usually not uniform and thus can be exploited to maximize the output rate of the join.

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.

A. 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 or the output utility of the stream join.

[1]Diag-Join is a join algorithm designed for efficiently executing equi-joins over 1:N relationships in data warehouses, which exploits the clusters formed within the base relations due to the time of data creation effects. These clusters are formed by tuples with temporarily close append times sharing the same value as their foreign key.

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

TABLE I

NOTATIONS USED THROUGHOUT THE PAPER

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.

B. 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 $\bar{i}$. 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 I summarizes the notations used throughout the paper.

A windowed stream join is performed by fetching tuples from the input streams and processing

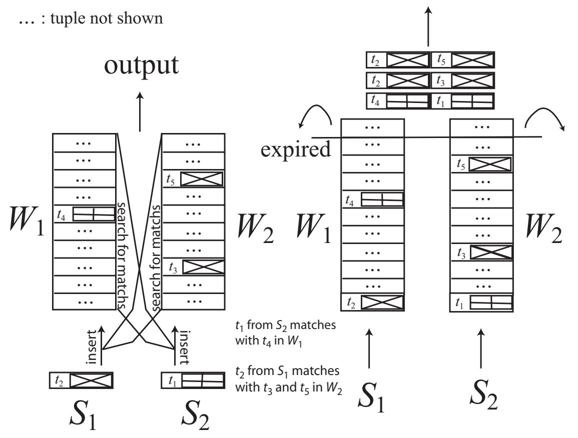


Fig. 2. Stream Join Example

```
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 $W_{\bar{i}}$
         if $T - T(t_o) > w_{\bar{i}}$
             Remove $t_o$ from $W_{\bar{i}}$
     until $T - T(t_o) \leq w_{\bar{i}}$
     foreach $t_a \in W_{\bar{i}}$
         Evaluate join condition on $t$, $t_a$
```

Fig. 3. Join Processing

them against tuples in the opposite window. Figure 2 illustrates the process of windowed stream joins. For a newly fetched tuple t from stream S_i, the join is performed in the following steps.

First, tuple t is inserted into the beginning of window W_i. Second, tuples at the end of window $W_{\bar{i}}$ are checked in order and removed if they have expired. A tuple t_o expires from window $W_{\bar{i}}$ iff $T - T(t_o) > w_{\bar{i}}$, 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 $W_{\bar{i}}$, and matching tuples are generated as output.

Figure 3 summarizes the join processing steps. Although not depicted in the pseudo-code, in practice buffers can be placed in the inputs of the join operator, which is common practice in DSMS query networks and also useful for masking small scale rate bursts in stand-alone joins. To handle joins defined on set or weighted set valued attributes, the following additional details are attached to the processing steps, assuming a tuple is a set of items (possibly with assigned weights). First, the items in tuple t are sorted as it is fetched from S_i. The tuples in $W_{\bar{i}}$ are expected to be sorted, since they have gone through the same step when they were fetched from $S_{\bar{i}}$. Then, for each tuple t_a in $W_{\bar{i}}$, t and t_a are compared by performing a simple merge of their sorted items. Equality, subset, superset, overlap and inner product joins all can be processed in a similar manner. For indexed joins, an inverted index is used to efficiently perform the join without going through all the tuples in $W_{\bar{i}}$. We discuss the details of indexed join in Section IV-B.

IV. 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.

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 concrete 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.

A. 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

Algorithm 1: Rate Adaptation

```
RATEADAPT()
(1) Initially: $r \leftarrow 1$
(2) every $T_r$ seconds
(3)     $\alpha_1 \leftarrow$ # of tuples fetched from $S_1$ since last rate adaptation step
(4)     $\alpha_2 \leftarrow$ # of tuples fetched from $S_2$ since last rate adaptation step
(5)     $\lambda_1 \leftarrow$ average rate of $S_1$ since last rate adaptation step
(6)     $\lambda_2 \leftarrow$ average rate of $S_2$ since last rate adaptation step
(7)     $\beta \leftarrow (\alpha_1 + \alpha_2) / ((\lambda_1 + \lambda_2) * T_r)$
(8)     if $\beta < 1$ then $r \leftarrow \beta * r$
(9)             else $r \leftarrow min(1, \delta_r * r)$
```

denote the fraction parameter as r, which defines the fraction of the windows to be processed. In other words, the setting of r answers the question of *how much* load we should shed.

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. Based on Nyquist's theorem and control theory, the adaptation period can be taken as half the period of the most frequent rate bursts in the streams. However, if this value increases the percentage of CPU time spent for performing the adaptation step beyond a small threshold ϵ (≤ 0.1%), then T_r should be set to its lower bound. This lower bound is the smallest value which results in spending ϵ percentage of the CPU time for adaptation.

B. Indexed Join and Partial Processing

Stream indexing [22], [23] 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 IV-A 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. Here, we first give a brief overview of inverted indexes and then describe the modifications required to use them in conjunction with our load shedding algorithms.

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 hashtable. The hashtable 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. Same type of processing is used for all different types of queries we have mentioned so far. Specifically, given a query set, the identifier lists corresponding to items in the query set are retrieved using the hashtable. These sorted identifier lists are then merged. This is done by inserting the frontiers of the lists into a min heap and iteratively removing the topmost set identifier from the heap and replacing it with the next set identifier (new frontier) in its list. During this process, the identifier of an indexed set, sharing k items with the query set, will be picked from the heap k consecutive times, making it possible to process relatively complex overlap and inner product[2] queries efficiently [24].

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 affect 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. This requires very small reordering, as the event of receiving different tuples with equal timestamps is expected to happen very infrequently, if it happens at all. Figure 4 provides an illustration of timestamp ordered identifier lists in inverted indexes.

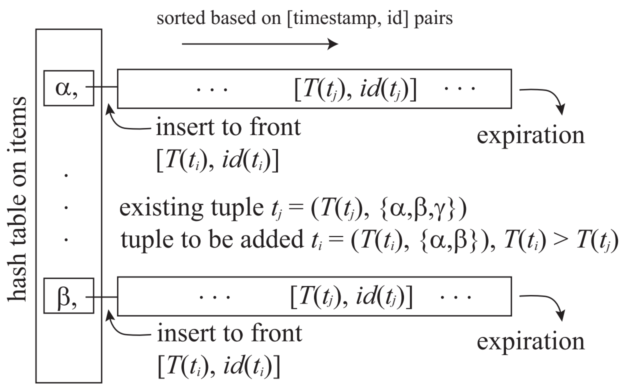

sorted based on [timestamp, id] pairs
hash table on items
α,
· · · $[T(t_j), id(t_j)]$ · · ·
insert to front $[T(t_i), id(t_i)]$
expiration
existing tuple $t_j = (T(t_j), \{\alpha,\beta,\gamma\})$
tuple to be added $t_i = (T(t_i), \{\alpha,\beta\}), T(t_i) > T(t_j)$
β,
· · · $[T(t_j), id(t_j)]$ · · ·
insert to front $[T(t_i), id(t_i)]$
expiration

Fig. 4. Illustration of an inverted index that uses (timestamp, identifier) pairs for sorting the identifier lists, which improves tuple insertion and expiration performance, without negatively effecting the search performance, since tuples with equal timestamps are rare.

Using timestamp ordered identifier lists has the following three advantages:

1. It allows inserting a set identifier into an identifier list in constant time, as opposed to logarithmic time with identifier sorted lists.

[2]For weighted sets, the weights should also be stored within the identifier lists, in order to answer inner product queries.

2. It facilitates piggybacking of removal operations on insertion and search operations, by checking for expired tuples at the end of identifier lists at insertion and search time. Thus, the removal operation is performed in amortized constant time as opposed to logarithmic time with identifier sorted lists.
3. Timestamp sorted identifier lists make it possible to end the merging process, used for search operations, at a specified time within the window, thus enabling time based partial processing.

V. 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. Three important factors are used to determine what we should select for join processing: (1) the characteristics of stream window segments, (2) the profitability of join directions, and (3) the utility of different stream tuples. We first describe time correlation adaptation and join direction adaptation, which form the core of our selective processing approach. Then we discuss utility-based load shedding. The main ideas behind time correlation adaptation and join direction adaptation 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).

A. *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,

Algorithm 2: Time Correlation Adaptation

```
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)             $o_{i,j} \leftarrow \hat{o}_{i,j} / (\gamma * r * b * \lambda_2 * \lambda_1 * T_c)$
(6)             $s_i^j \leftarrow k$, where $O[j] = \hat{o}_{i,k}$
(7)         for $j = 1$ to $n_i$
(8)             $\hat{o}_{i,j} \leftarrow 0$
```

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 either 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. This can not be done with selective processing, since it introduces bias as it processes only certain segments of the join windows and can not capture changing time correlations.

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

Algorithm 3: Tuple Processing and Time Correlation

```
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)                 $r_e \leftarrow 1 + a/|B_{i,s_i^j}|$
(14)                 process $t$ against $r_e$ fraction of tuples in $B_{i,s_i^j}$
(15)                 break
```

particular, $o_{i,j}$ values are used to calculate s_i^j values. Here, s_j^i is the index of the basic window which has the jth highest o value among the basic windows within W_i. Concretely:

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

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 IV-A. 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.

1) Impact of Basic Window Size: The setting of basic window size parameter b involves trade-offs. Smaller values are better to capture the peak of the match probability distribution, while they also introduce overhead in processing. For instance, recalling Section IV-B.1, in an indexed join operation, the identifier lists have to be looked up for each basic window. Although the lists themselves are shorter and the total merging cost does not increase with smaller basic windows, the cost of looking up the identifier lists from the hashtables increases with increasing number of basic windows, n_i.

Here we analyze how well the match probability distribution, which is dependent on the time correlation between the streams, is utilized for a given value of the basic window size parameter b, under a given load condition. We use r' to denote the fraction of tuples in join windows that can be used for processing. Thus, r' is used to model the current load of the system. We assume that r' can go over 1, in which case processing resources are abundant.

We use $f_i(.)$ to denote the match probability distribution function for window W_i, where $\int_{T_{\Delta_1}}^{T_{\Delta_2}} f_i(y)dy$ gives the probability that a newly fetched tuple will match with a tuple t in W_i that has a timestamp $T(t) \in [T - T_{\Delta_1}, T - T_{\Delta_2}]$. Note that, due to discrete movement of basic windows, a basic window covers a time varying area under the match probability distribution function. This area, denoted as $p_{i,j}$ for basic window $B_{i,j}$, can be calculated by observing that the basic window $B_{i,j}$ covers the area over the interval $[max(0, x*b+(j-2)*b), min(w_i, x*b+(j-1)*b)]$ on the time axis ($[0, w_i]$), when only $x \in [0, 1]$ fraction of the first basic window is full. Then, we have:

$$p_{i,j} \;=\; \int_{x=0}^{1} \int_{t=max(0,x*b+(j-2)*b)}^{min(w_i,x*b+(j-1)*b)} f_i(y)\;dy\;dx$$

For the following discussion, we overload the notation s_i^j, such that $s_i^j = k$, where $p_{i,k}$ is the jth item in the sorted list $\{p_{i,l} | l \in [1..n_i]\}$. The number of basic windows whose tuples are all considered for processing is denoted as c_e. The fraction of tuples in the last basic window used, that are considered for processing, is denoted as c_p. c_p is zero if the last used basic window is completely processed. We have:

$$\begin{aligned} c_e &= min(n_i, \lfloor r' * w_i / b \rfloor) \\ c_p &= \begin{cases} \frac{r' * w_i - c_e * b}{b} & c_e < n_i \\ 0 & otherwise \end{cases} \end{aligned}$$

Then the area under f_i that represents the portion of window W_i processed, denoted as p_u, can be calculated as:

$$p_u \;\approx\; c_p * p_{s_i^{c_e+1}} + \sum_{j=1}^{c_e} p_{i,s_i^j}$$

Let us define $g(f_i, a)$ as the maximum area under the function f_i with a total extent of a on the time axis. Then we can calculate the optimality of p_u, denoted as ϕ, as follows:

$$\phi \;=\; \frac{p_u}{g(f_i, w_i * min(1, r'))}$$

When $\phi = 1$, the join processing is optimal with respect to output rate (ignoring the overhead of small basic windows). Otherwise, the expected output rate is ϕ times the optimal value, under current load conditions (r') and basic window size setting (b). Figure 5 plots ϕ (on z-axis) as a function of b/w_i (on x-axis) and r' (on y-axis) for two different match probability distributions, the bottom one being more skewed. We make the following three observations from the figure:

- Decreasing availability of computational resources negatively influences the optimality of the join for a fixed basic window size.
- The increasing skewness in the match probability distribution decreases the optimality of the join for a fixed basic window size.
- Smaller basic windows sizes provide better join optimality, when the available computational resources are low or the match probability distribution is skewed.

The intuition behind these observations is as follows. When only a small fraction of the join windows can be processed, we have to pick a small number of basic windows that are around the peak of the match probability distribution. When the granularity of the basic windows is not sufficiently fine grained, the limited resources are spent processing less beneficial segments that appear within the basic windows. This effect is more pronounced when the match probability distribution is more skewed. As a result, small basic window sizes are favorable for skewed probability match distributions and heavy load conditions. We report our experimental study on the effect of overhead, stemming from managing large number of basic windows, on the output rate of the join in Section VI.

B. 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, 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*.

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

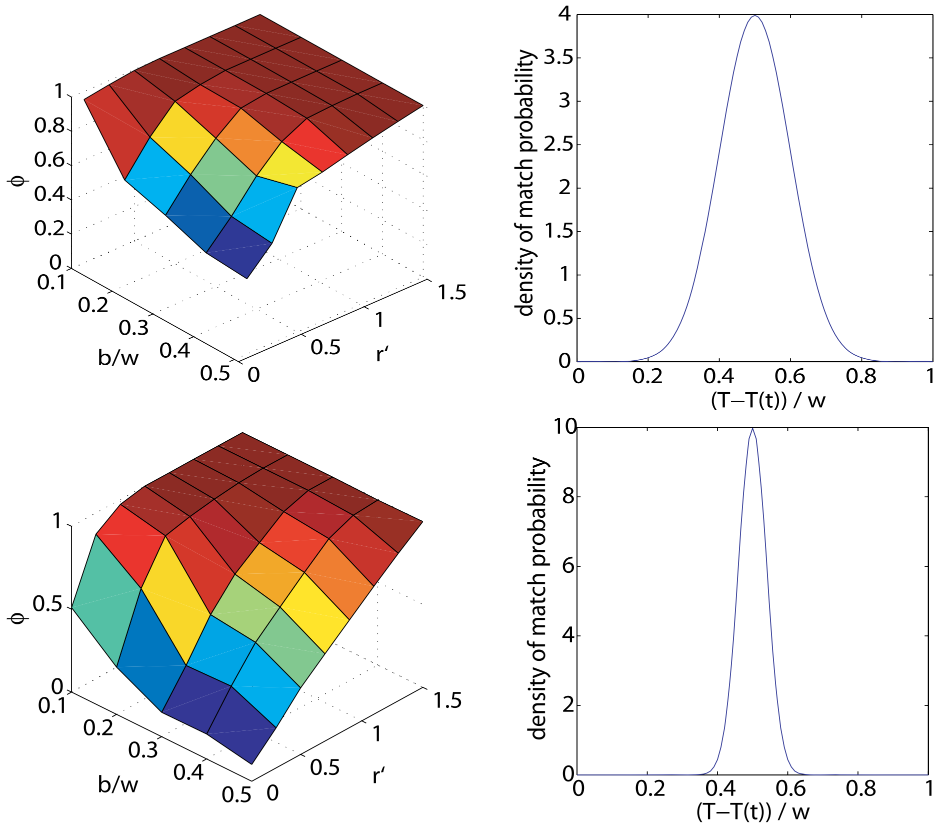


Fig. 5. Optimality of the join for different loads and basic window sizes under two different match probability distributions

Algorithm 4: Join Direction Adaptation

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

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. We introduce a more advanced join direction adaptation algorithm, that can handle such cases, in the next subsection as part of utility-based load shedding.

C. Utility-based Load Shedding

So far, we have targeted our load shedding algorithms toward maximizing the number of tuples produced by the join operation, a commonly used metric in the literature [12], [13]. Utility-based load shedding, also called semantic load shedding [11], is another metric employed for guiding load shedding. It has the benefit of being able to distinguish high utility output from output containing large number of tuples. In the context of join operations, utility-based load shedding promotes output that results from matching tuples of higher importance/utility. In this section, we describe how utility-based load shedding is integrated into the mechanism described so far.

We assume that each tuple has an associated importance level, defined by the *type* of the tuple, and specified by the *utility* value attached to that type. This type and utility value assignment is application specific and is external to our join algorithm. 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. It is the responsibility of the domain expert to specify what types of tuples are considered important. In some cases, data mining operators can also be used to mark tuples with their importance types. As long as each tuple has an assigned utility value, our utility-based load shedding algorithm can maximize output utility while working in tandem with the selective processing based load shedding algorithm that has rate, time correlation, and join direction adaptation.

We denote the tuple type domain as $\mathcal{Z}$, type of a tuple t as $Z(t)$, and utility of a tuple t, where $Z(t) = z \in \mathcal{Z}$, as $\mathcal{V}(z)$. Type domains and their associated utility values can be set based on application needs. In the rest of the paper, the utility value of an output tuple of the the join operation that is obtained by matching tuples t_a and t_b, is assumed to contribute a utility value of $max\,(\mathcal{V}(Z(t_a)), \mathcal{V}(Z(t_b)))$ to the output. Our approach can also accommodate other functions, like average $(0.5 * (\mathcal{V}(Z(t_a)) + \mathcal{V}(Z(t_b))))$. We denote the frequency of appearance of a tuple of type z in stream S_i as $\omega_{i,z}$, where $\sum_{z \in \mathcal{Z}} \omega_{i,z} = 1$.

The main idea behind utility-based load shedding is to use a different fraction parameter for each different type of tuple fetched from a different stream, denoted as $r_{i,z}$, where $z \in \mathcal{Z}$ and $i \in \{1, 2\}$. The motivation behind this is to do less load shedding for tuples that provide higher output utility. The extra work done for such tuples is compensated by doing more load shedding for tuples that provide lower output utility. The expected output utility obtained from comparing a tuple t of type z with a tuple in window W_i is denoted as $u_{i,z}$, and is used to determine $r_{i,z}$'s.

In order to formalize this problem, we extend some of the notation from Section V-A.1. The number of basic windows from W_i whose tuples are all considered for processing against a tuple of type z, is denoted as $c_e(i, z)$. The fraction of tuples in the last basic window used from W_i, that are considered for processing, is denoted as $c_p(i, z)$. $c_p(i, z)$ is zero if the last used basic window is completely processed. Thus, we have:

$$\begin{aligned} c_e(i,z) &= \lfloor n_i * r_{i,z} \rfloor \\ c_p(i,z) &= n_i * r_{i,z} - c_e(i,z) \end{aligned}$$

Then, the area under f_i that represents the portion of window W_i processed for a tuple of type z, denoted as $p_u(i,z)$, can be calculated as follows:

$$p_u(i,z) \approx c_p(i,z) * p_{i,s_i^{ce(i,z)+1}} + \sum_{j=1}^{c_e(i,z)} p_{i,s_i^j}$$

With these definitions, the maximization of the output utility can be defined formally as

$$max \sum_{i=1}^{2} \left(\lambda_{\bar{i}} * (\lambda_i * w_i) * \sum_{z \in \mathcal{Z}} \left(\omega_{\bar{i},z} * u_{i,z} * p_u(i,z) \right) \right)$$

subject to the processing constraint:

$$r * (w_2 + w_1) = \sum_{i=1}^{2} \left(w_i * \sum_{z \in \mathcal{Z}} \left(\omega_{\bar{i},z} * r_{i,z} \right) \right)$$

The r value used here is computed by Algorithm 1, as part of rate adaptation. Although the formulation is complex, this is indeed a fractional knapsack problem and has a greedy optimal solution. This problem can be reformulated[3] as follows: Consider $\mathcal{I}_{i,j,z}$ as an item that represents processing of a tuple of type z against basic window $B_{i,j}$. Item $\mathcal{I}_{i,j,z}$ has a volume of $\lambda_1 * \lambda_2 * \omega_{\bar{i},z} * b$ units (which is the number of comparisons made per time unit to process incoming tuples of type z against tuples in $B_{i,j}$) and a value of $\lambda_1 * \lambda_2 * \omega_{\bar{i},z} * b * u_{i,z} * p_{i,s_i^j} * n_i$ units (which is the utility gained per time unit, from comparing incoming tuples of type z with tuples in $B_{i,j}$). The aim is to pick maximum number of items, where fractional items are acceptable, so that the total value is maximized and the total volume of the picked items is at most $\lambda_1 * \lambda_2 * r * (w_2 + w_1)$. $r_{i,j,z} \in [0,1]$ is used to denote how much of item $I_{i,j,z}$ is picked. Note that the number of unknown variables here ($r_{i,j,z}$'s) is $(n_1 + n_2) * |\mathcal{Z}|$, and the solution of the original problem can be calculated from these variables as, $r_{i,z} = \sum_{j=1}^{n_i} r_{i,j,z}$.

The values of the fraction variables are determined during join direction adaptation. A simple way to do this, is to sort the items based on their value over volume ratios, $v_{i,j,z} = u_{i,z} * p_{i,s_i^j} * n_i$ (note that $o_{i,j} / \sum_{k=1}^{n_i} o_{i,k}$ can be used as an estimate of p_{i,s_i^j}), and to pick as much as possible

[3]assuming that some buffering is performed outside the join

Algorithm 5: Join Direction Adapt, Utility-based Shedding

```
VJOINDIRECTIONADAPT()
(1)  upon completion of RATEADAPT() call
(2)      heap: $H$
(3)      for $i = 1$ to $2$
(4)          foreach $z \in \mathcal{Z}$
(5)              $r_{i,z} \leftarrow 0$
(6)              $v_{i,s_i^1,z} \leftarrow u_{i,z} * n_i * o_{i,s_i^1} / \sum_{k=1}^{n_i} o_{i,k}$
(7)      Initialize $H$ with $\{v_{i,s_i^1,z} | i \in [1..2], z \in \mathcal{Z}\}$
(8)      $a \leftarrow \lambda_1 * \lambda_2 * r * (w_1 + w_2)$
(9)      while $H$ is not empty
(10)         use $i, j, z$ s.t. $v_{i,j,z} =$ topmost item in $H$
(11)         pop the first item from $H$
(12)         $a \leftarrow a - \omega_{\bar{i},z} * \lambda_1 * \lambda_2 * b$
(13)         if $a > 0$
(14)             $r_{i,z} \leftarrow r_{i,z} + \frac{1}{n_i}$
(15)         else
(16)             $r_e \leftarrow 1 + a/(\lambda_1 * \lambda_2 * \omega_{\bar{i},z} * b)$
(17)             $r_{i,z} \leftarrow r_{i,z} + r_e/n_i$
(18)             return
(19)         if $j < n_i$
(20)             $v_{i,s_i^{j+1},z} \leftarrow u_{i,z} * n_i * o_{i,s_i^{j+1}} / \sum_{k=1}^{n_i} o_{i,k}$
(21)             insert $v_{i,s_i^{j+1},z}$ into $H$
```

of the item that is most valuable per unit volume. However, since the number of items is large, the sort step is costly, especially for large number of basic windows and large sized domains. A more efficient solution, with worst case complexity $\mathcal{O}(|\mathcal{Z}| + (n_1 + n_2) * \log |\mathcal{Z}|)$, is described in Algorithm 5, which replaces Algorithm 4. Algorithm 5 makes use of the s_i^j values that define an order between value over volume ratios of items for a fixed type z and window W_i. The algorithm keeps the items representing different streams and types with the highest value over volume ratios ($2 * |\mathcal{Z}|$ of them), in a heap. It iteratively picks an item from the heap and replaces it with the item having the next highest value over volume ratio with the same stream and type subscript index. This process continues until the capacity constraint is reached. During this process $r_{i,z}$ values are calculated progressively. If the item picked represents window W_i and type z, then $r_{i,z}$ is incremented by $1/n_i$ unless the item is picked fractionally, in which case the increment on $r_{i,z}$ is adjusted accordingly.

VI. 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. We report four sets of experimental results to demonstrate effectiveness of the algorithms introduced in this paper. The first set of experiments illustrates the need for shedding CPU load for both indexed (using inverted indexes) and non-indexed joins. The second 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 third 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. The final set of experiments presents results on the utility-based load shedding mechanisms introduced and their ability to maximize the output utility under different workloads. 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.

A. 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, join direction adaptation, and utility-based load shedding, 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 6 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.

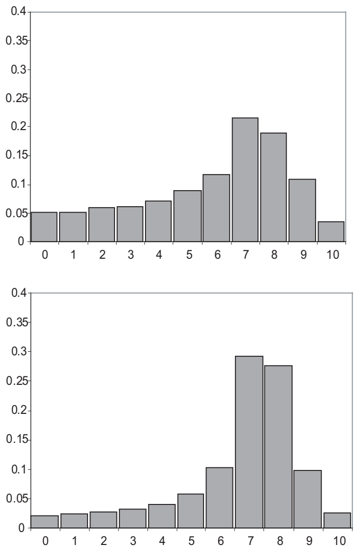


Fig. 6. Probability match distributions, $\kappa = 0.6$ and $\kappa = 0.8$

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. Other types of joins show similar results. 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.

B. Processing Power Limitation

We first validate that the processing power happens to be the limiting resource, for both indexed and non-indexed join operations. Graphs in Figure 7 plot input tuple drop rates (sum of the drop rates of the two streams) for non-indexed (left) and indexed joins (right), as a function of window size for different stream rates. The join operation performed is an overlap join with threshold value of 3. It is observed from the figure that, for a non-indexed join, even a low stream rate of 50 tuples per second results in dropping approximately half of the tuples, when the window size is set to 125 seconds. This corresponds to a rather small window size of around 150 KBytes, when compared to the total memory available. Although indexed join improves

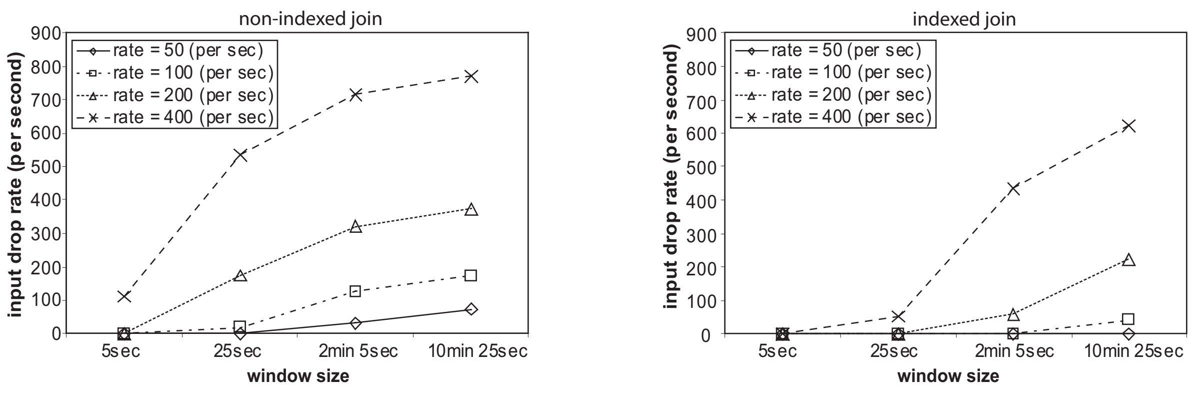


Fig. 7. Tuple drop rates for non-indexed and indexed join operations, as a function of window size, with varying stream rates

performance by decreasing tuple drop rates, it is only effective for moderate window sizes and low stream rates. A stream rate of 100 tuples per second results in dropping approximately one quarter of the tuples for a 625 seconds window (approx. 10 minutes). This corresponds to a window size of around 1.5 MBytes. As a result, CPU load shedding is a must for costly stream joins, even when indexes are employed.

C. 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 8 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 8 also shows (on the right y-axis) how fraction parameter r adapts to the changing stream rates.

The graphs in Figure 9 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

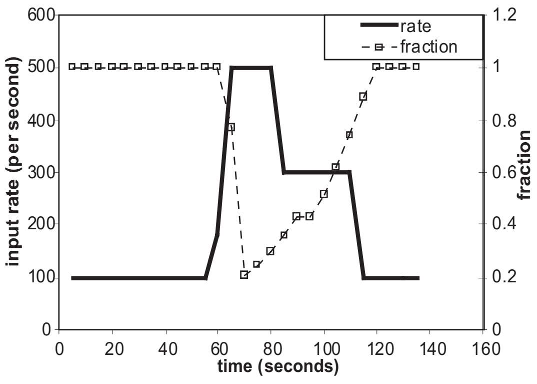


Fig. 8. Stream rates and fraction parameter r

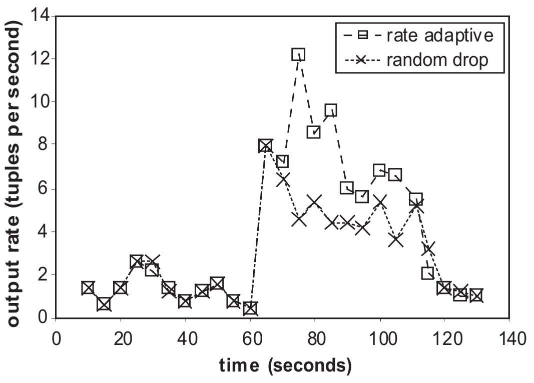


Fig. 9. Improvement in output rate with rate adaptation

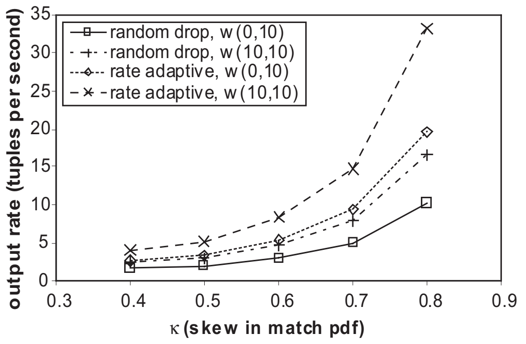


Fig. 10. Improvement in average output rate with rate adaptation

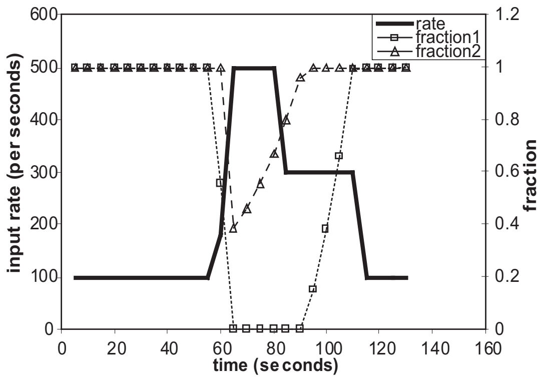


Fig. 11. Stream rates and fraction parameters r_1 and r_2

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 10 plot the average output rates of the join over the period shown in Figure 9 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.

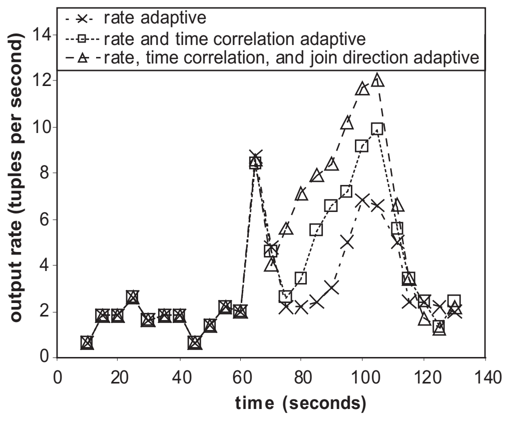


Fig. 12. Improvement in output rate with time correlation and join direction adaptation

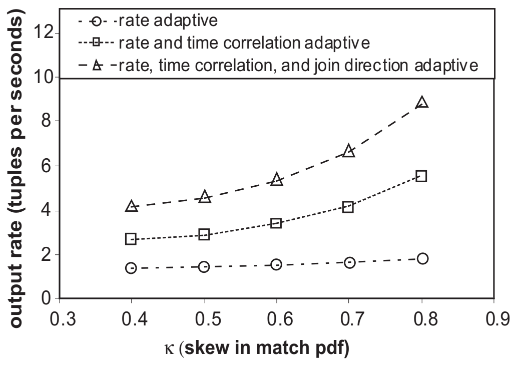
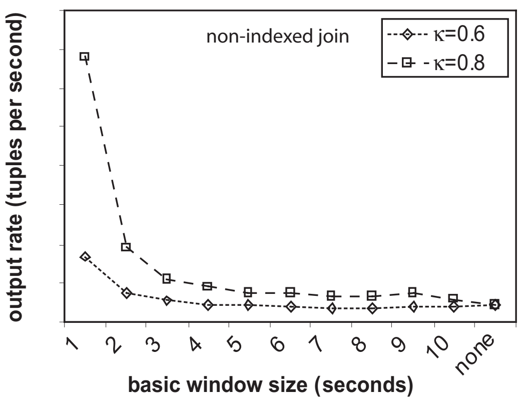


Fig. 13. Improvement in average output rate with time correlation and join direction adaptation



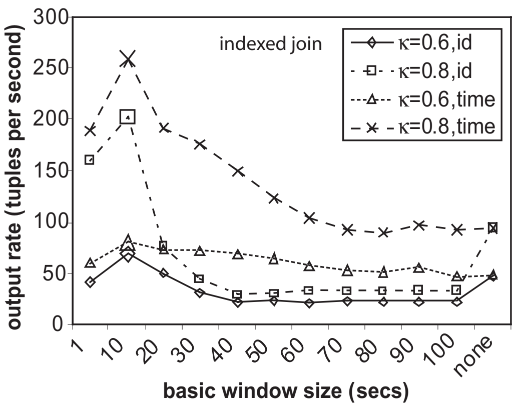


Fig. 14. Impact of basic window size for indexed and non-indexed join, for various basic window sizes

D. 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 11 shows the stream rates used (on the left y-axis) as a function of time. Figure 11 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 12 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 13 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.

1) Indexed Joins: The idea of selective processing applies equally well to indexed joins, as it applies to non-index joins, as long as the cost of processing a tuple against the window defined on the opposite stream monotonically increases with the increasing fraction of the window processed. This assumption is valid for most type of joins, such as the inverted-index supported set joins discussed before. One exception to this is the hash-table supported equi-joins, since probing the opposite window takes constant time for equi-joins with hash tables. As a result, no gain is to be expected from using selective processing for equi-joins.

The graphs in Figure 15 plot the improvement provided by selective processing, with all adaptations applied, over random dropping in terms of the output rate of the join as a function of stream rates, for equi-joins and overlap joins with and without indexing support. For the non-indexed scenarios windows of size 20 seconds are used, whereas for indexed scenarios windows of size 200 seconds are used to increase the cost of the join to necessitate load shedding (recall Figure 7).

We make three observations from Figure 15. First, for non-indexed joins (equi-join or overlap join), selective processing provides up to 500% higher output rate, the improvement being higher for overlap joins, as they are more costlier to evaluate. Second, for overlap joins with inverted indexes the improvement provided by selective processing is not observed until the stream rates get high enough, 300 tuples/sec in this example. This is intuitive, since with indexes the join is evaluated much faster and as a result the load shedding starts when the stream rates are high enough to saturate the processing resources. From the figure, we observe up to around 100% improvement over random dropping for the case of indexed overlap join. The relative improvement in output rate is likely to increase for rates higher than 500 tuples/sec, as the trend of the line for indexed overlap join suggests in Figure 15. Third, we see no improvement in output rate when indexed equi-join is used. As the zoomed-in portion of the figure shows, there is no deterioration in the output rate either. As a result, selective sampling is performance-wise indifferent in the case of equi-joins, compared to random dropping.

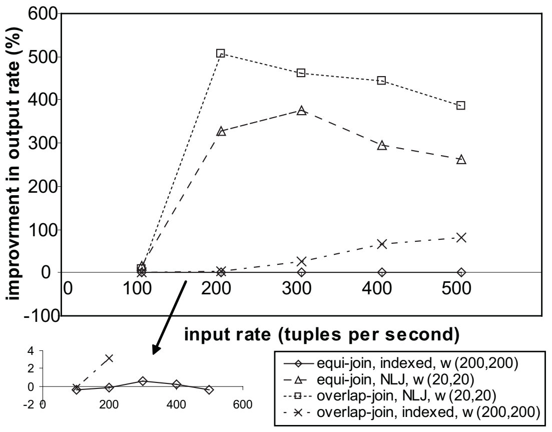


Fig. 15. Improvement in output rate compared to random dropping, for indexed and non-indexed joins, for overlap and equality join conditions

2) Basic Window Size: We study the impact of basic window size on output rate of the join operation. The graphs in Figure 14 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/sec 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 large. This is because identifier sorted inverted indexes do not allow partial processing based on time.

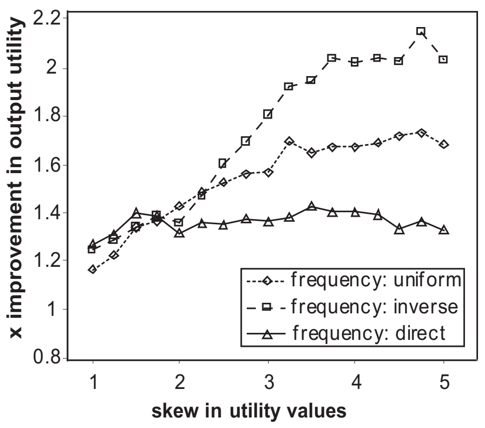


Fig. 16. Improvement in output utility for different type frequency models

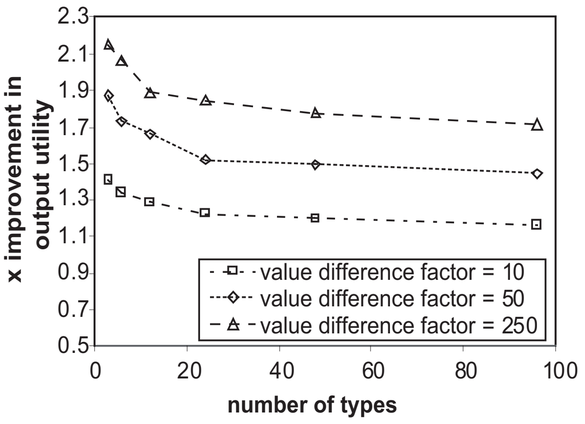


Fig. 17. Improvement in output utility for different type domain sizes

E. Utility-based Load Shedding

We study the effectiveness of utility-based load shedding in improving the output utility of the join operation. We consider three different scenarios in terms of setting type frequencies; (i) uniform, (ii) inversely proportional to utility values, and (iii) directly proportional to utility values. For the experiments in this subsection, we use a non-indexed overlap join, with threshold value of 3 and 20 seconds windows on both of the streams. 500 tuples per second is used as the stream rate. The graphs in Figure 16 plot the improvements in the output utility of the join, compared to the case where no utility-based load shedding is used, as a function of skewness in utility values. Both joins are rate, time correlation, and join direction adaptive. In this experiment, there are three different tuple types, i.e. $|\mathcal{Z}| = 3$. For a skewness value of k, the utility values of the types are $\{1, 1/2^k, 1/3^k\}$. It is observed from the figure that, the improvement in the output utility increases with increasing skewness for uniform and inversely proportional type frequencies, where it stays almost stable for directly proportional type frequencies. Moreover, the best improvement is provided when item frequencies are inversely proportional to utility values. Note that this is the most natural case, as in most applications, rare items are of higher interest.

The graph in Figure 17 studies the effect of domain size on the improvement in the output utility. It plots the improvement in the output utility as a function of type domain size, for different *value difference factors*. A value difference factor of x means the highest utility value is x times the lowest utility value and the utility values follow a Zipf distribution (with parameter $\log_{|\mathcal{Z}|} x$). The type frequencies are selected as inversely proportional to utility values. It is observed from the figure that, there is an initial decrease in the output utility improvement with increasing type domain size. But the improvement values stabilize quickly as the type domain size gets larger. Same observation holds for different amounts of skewness in utility values.

VII. 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. This enabled our algorithms to nicely integrate utility-based load shedding with time correlation-based load shedding in order to improve output utility of the join for the applications where some tuples are evidently more valued than others. 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] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “TAG: a Tiny AGgregation service for ad-hoc sensor networks,” in *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002.

[2] 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.

[3] 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*, vol. 26, March 2003.

[4] 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 *Proceedings of the Conference on Innovative Database Research*, 2003.

[5] L. Golab and M. T. Ozsu, "Processing sliding window multi-joins in continuous queries over data streams," in *Proceedings of the International Conference on Very Large Data Bases*, 2003.

[6] J. Kang, J. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," in *Proceedings of the IEEE International Conference on Data Engineering*, 2003.

[7] M. A. Hammad and W. G. Aref, "Stream window join: Tracking moving objects in sensor-network databases," in *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2003.

[8] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, 2003.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the ACM Symposium on Principles of Database Systems*, 2002.

[10] J. Kleinberg, "Bursty and hierarchical structure in streams," in *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining*, 2002.

[11] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of the International Conference on Very Large Data Bases*, 2003.

[12] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *Proceedings of the ACM International Conference on Management of Data*, 2003.

[13] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *Proceedings of the International Conference on Very Large Databases*, 2004.

[14] 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 *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[15] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: operator scheduling for memory minimization in data stream systems," in *Proceedings of the ACM International Conference on Management of Data*, 2003.

[16] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, resource management, and approximation in a data stream management system," in *Proceedings of the Conference on Innovative Database Research*, 2003.

[17] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proceedings of the IEEE International Conference on Data Engineering*, 2004.

[18] S. Chandrasekaran and M. J. Franklin, "Remembrance of streams past: Overload-sensitive management of archived streams," in *Proceedings of the International Conference on Very Large Databases*, 2004.

[19] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid, "Query processing using negative tuples in stream query engines," Purdue University, Tech. Rep. CSD TR# 04-040, 2005.

[20] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the ACM Symposium on Principles of Database Systems*, 2004.

[21] S. Helmer, T. Westmann, and G. Moerkotte, "Diag-Join: An opportunistic join algorithm for 1:N relationships," in *Proceedings of the International Conference on Very Large Data Bases*, 1998.

[22] L. Golab, S. Garg, and M. T. Ozsu, "On indexing sliding windows over online data streams," in *Proceedings of the International Conference on Extending Database Technology*, 2004.

[23] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Interval query indexing for efficient stream processing," in *Proceedings of the ACM International Conference on Information and Knowledge Management*, 2004.

[24] N. Mamoulis, "Efficient processing of joins on set-valued attributes," in *Proceedings of the ACM International Conference on Management of Data*, 2003.