GRUBJOIN: An Adaptive Multi-Way Windowed Stream Join with Time Correlation-Aware CPU Load Shedding

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

Kun-Lung Wu, Philip S. Yu
IBM Tomas J. Watson
Research Center
[klwu,psyu]@us.ibm.com

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

ABSTRACT

Dropping tuples has been commonly used for load shedding. However, tuple dropping generally is inadequate to shed load for multi-way windowed stream joins. The output rate can be unnecessarily and severely degraded because tuple dropping does not recognize time correlations likely to exist among the streams. This paper introduces *GrubJoin*: an adaptive multi-way windowed stream join that efficiently performs time correlation-aware CPU load shedding. GrubJoin maximizes the output rate by achieving near-optimal *window harvesting* within an *operator throttling* framework, i.e., regulating the fractions of the join windows that are processed by the multi-way join. Window harvesting performs the join using only certain more useful segments of the join windows. Due mainly to the combinatorial explosion of possible multi-way join sequences involving various segments of individual join windows, GrubJoin faces a set of unique challenges, such as determining the optimal window harvesting configuration and learning the time correlations among the streams. To tackle these challenges, we formalize window harvesting as an optimization problem, develop greedy heuristics to determine near-optimal window harvesting configurations and use approximation techniques to capture the time correlations among the streams. Experimental results show that GrubJoin is vastly superior to tuple dropping when time correlations exist among the streams and is equally effective as tuple dropping in the absence of time correlations.

1. INTRODUCTION

In today's highly networked and digital world, businesses often rely on time critical tasks that require analyzing data from on-line sources and generating response in real-time. In many industries, the on-line data to be analyzed comes in the form of data streams, i.e. as time ordered series of events or readings. Examples include stock tickers in financial services, link statistics in networking and telecommunications, sensor network readings in environmental monitoring and emergency response, and surveillance data in Homeland Security. In these examples, rapidly increasing rates of data streams and stringent response time requirements of applications force a paradigm shift in how the data is processed, moving away from the traditional "store and then process" model of database management systems (DBMSs) to "on-the fly processing" model of emerging data stream management systems (DSMSs). This shift has recently created a strong interest in research on DSMSs, in both academia [1, 4, 5] and industry [17].

CPU load shedding is needed in DSMSs when the available processing resources are not sufficient to handle the processing demands of the continuous queries installed in the system, under current rates of the input streams. Without load shedding, the mismatch between the available resources and the demands will result in delays that violate the response time requirements of the queries. It will also cause unbounded growth in system queues that overloads memory capacity and further bogs down the system. As a solution to these problems, CPU load shedding can be broadly defined as a mechanism to reduce the amount of processing performed for evaluating stream queries, in an effort to match the service rate of the DSMS to its input rate, at the cost of producing a degraded output. Depending on the stream operators, the output degradation may take different forms, such as a lower resolution in a multimedia encoder operator or a subset result in a join operator.

Joins are key operators in DSMSs and costlier to evaluate when compared with others, such as selections and projections. They are usually used to correlate events from different data streams. As examples, we list two stream join applications.

Example 1 [8] - *Finding similar news items from different news sources*: Assuming that news items from CNN, Reuters, and BBC are represented by weighted keywords (join attribute) in their respective streams, we can perform a windowed inner product join to find similar news items from different sources.

Example 2 [11] - *Tracking objects using multiple video (sensor) sources*: Assuming that scenes (readings) from video (sensor) sources are represented by multi-attribute tuples of numerical values (join attribute), we can perform a distance-based similarity join to detect objects that appear in all of the video (sensor) sources.

Hence, it is important to study load shedding techniques in the context of stream joins, particularly in the face of busrty and unpredictable stream rates. In this paper, we focus on CPU load shedding for multi-way windowed stream joins. Join operations are performed on the tuples stored within user-defined time-based join windows, which constitute one of the most common join types in the DSMS research [3, 10, 13].

So far, the predominantly used approach to CPU load shedding in stream joins has been tuple dropping [2, 18]. This can be seen as a *stream throttling* approach, where the rates of the input streams are sufficiently reduced via the use of tuple dropping, in order to sustain a stable system. However, tuple dropping generally is inadequate to shed load for multi-way windowed stream joins. The output rate of a multi-way join can be unnecessarily and severely

degraded because tuple dropping does not recognize and exploit the time correlations likely to exist among the streams.

The time correlation assumption indicates that for pairs of matching tuples from two streams, there exists a non-flat match probability distribution which is a function of the time difference between the timestamps of the tuples. For instance, in Example 1 above, it is more likely that a news item from one source will match with a temporally close news item from another source. In this case the streams are almost aligned and the probability that a tuple from one stream will match with a tuple from another stream decreases as the difference between their timestamps increase. However, the streams can also be unaligned, either due to delays in the delivery path, such as network and processing delays, or due to the inherent time of event generation effects in the application. As an illustration to the latter, in Example 2 above, similar tuples appearing in different video streams or similar readings found in different sensor streams will have a *lag* between their timestamps, due to the time it takes for an object to pass through all cameras or all sensors.

In this paper, we introduce *GrubJoin*[1]: an adaptive multi-way windowed stream join that efficiently performs time correlation-aware CPU load shedding. While shedding load, GrubJoin maximizes the output rate by achieving near-optimal *window harvesting* within an *operator throttling* framework. In contrast to stream throttling, operator throttling performs load shedding within the stream operator, i.e., regulating the amount of work performed by the join, similar in spirit to the concept of partial processing described in [8] for two-way stream joins. This requires altering the processing logic of the multi-way join by parameterizing it with a *throttle fraction*. The parameterized join incurs only a throttle fraction of the processing cost required to perform the full join operation. As a side effect, the quality or the quantity of the output produced may be decreased when load shedding is performed.

Window harvesting performs effective load shedding by executing the join operations on tuples only from certain more useful segments of the join windows. For efficient implementation, GrubJoin divides each join window into multiple small-sized segments of basic windows. Due mainly to the combinatorial explosion of possible multi-way join sequences involving various segments of individual join windows, GrubJoin faces a set of challenges in performing window harvesting. These challenges are unique for a multi-way windowed stream join and do not exist for a two-way windowed stream join. In particular, there are three major challenges that should be resolved:

– First, mechanisms are needed to configure window harvesting parameters to match the required throttle fraction imposed by operator throttling. We should also be able to assess the optimality of these mechanisms in terms of output rate, with respect to the best achievable for a given throttle fraction and known time correlations between the streams.

– Second, in order to be able to react and adapt to the possibly changing stream rates in a timely manner, the reconfiguration of window harvesting parameters must be a lightweight operation, so that the processing cost of reconfiguration does not consume the processing resources used to perform the join.

– And third, we should develop low cost mechanisms for learning the time correlations among the streams, in case they are not known or are changing and should also be adapted.

We tackle the first challenge by developing a cost model and formulating window harvesting as an optimization problem. We handle the latter two challenges by developing GrubJoin - a multi-way stream join algorithm that employs i) greedy heuristics for making near-optimal window harvesting decisions, and ii) approximation techniques to capture time correlations among the streams.

To the best of our knowledge, this is the first work on time correlation-aware CPU load shedding for multi-way windowed stream joins that are adaptive to the input stream rates. However, we are not the first to point out and take advantage of the time correlation effects in join processing. In the context of limited memory binary stream joins, the *age-based* load shedding framework of [16] pointed out the importance of time correlation effects and exploited it to make tuple replacement decisions. Furthermore, in the context of traditional joins, the database literature includes join operators such as Drag-Join [12], that capitalized on the time of data creation effects in data warehouses, that are very similar to the time correlation effects in stream joins. Moreover, similar time correlation assumptions are used to develop load shedding techniques for two-way stream joins [8]. However, the window harvesting problem, as it is formulated in this paper, involves unique challenges stemming from the multi-way nature of the join operation.

Summary of Contributions

In summary, this paper makes three major contributions:

1) We introduce window harvesting as an in-operator load shedding technique for multi-way windowed stream joins, that can adjust the amount of shedding performed based on the throttle fraction defined by our operator throttling framework. We formalize window harvesting configuration as an optimization problem and show how it can be utilized to exploit the time correlations among the streams to maximize the output rate of the join.

2) We develop the GrubJoin algorithm, that can adapt to the changes in the input stream rates, the current system load, and the time correlations among the streams. It performs near-optimal window harvesting and has very low overhead, thanks to the heuristic methods it employs for performing reconfiguration of harvesting parameters, and the approximation techniques it uses to learn the time correlations among the streams.

3) We report results from our experimental studies and show that GrubJoin performs vastly superior to tuple dropping in terms of output rate when time correlations exist among the streams, and is equally effective as tuple dropping in the absence of correlations.

2. PRELIMINARIES

Before going into the details of operator throttling and window harvesting, in this section we present our window-based stream join model, introduce some notations, and describe the basics of multi-way windowed stream join processing.

We denote the ith input stream by S_i, where $i \in [1..m]$ and $m \geq 2$ denotes the number of input streams of the join operator, i.e. we have an m-way join. Each stream is a sequence of tuples ordered by an increasing timestamp. We denote a tuple by t and its timestamp by $T(t)$. Current time is denoted by T. We assume that tuples are assigned timestamps upon their entrance to the DSMS. We do not enforce any particular schema type for the input streams. Schemas of the streams can include attributes that are single-valued, set valued, user defined, or binary. The only requirement is to have timestamps and an appropriate join condition defined over the input streams. We denote the current rate, in terms of tuples per second, of an input stream S_i as λ_i.

An m-way stream join operator has m join windows, as shown in the 3-way join example of Figure 1. The join window for stream S_i is denoted by W_i, and has a user defined size, in terms of seconds, denoted by w_i. A tuple t from S_i is kept in W_i only if

[1]As an intransitive verb, grub means "to search laboriously by digging". It relates to the way join windows are processed with window harvesting.

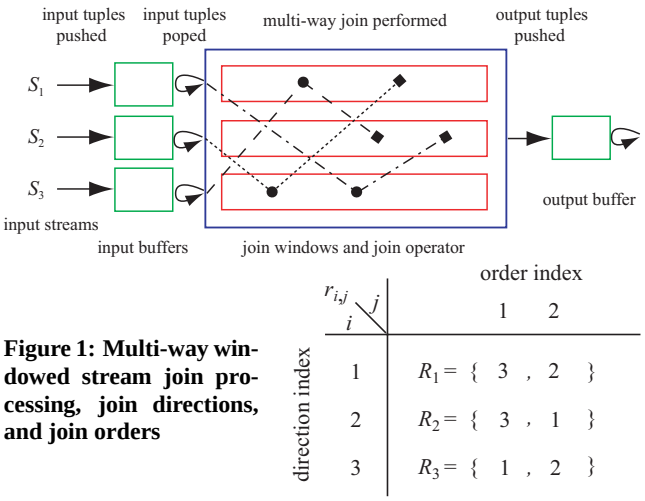

Figure 1: Multi-way windowed stream join processing, join directions, and join orders

$T \geq T(t) \geq T - w_i$. The join operator has buffers (queues) attached to its inputs and output. The input stream tuples are pushed into their respective input buffers either directly from their source or from output of other operators. The join operator processes tuples by fetching them from its input buffers, processes the join, and pushes the resulting tuples into the output buffer.

The GrubJoin algorithm we develop in this paper can be seen as a descendant of MJoin [20]. MJoins have been shown to be very effective for performing fine-grained adaptation and are very suitable for streaming scenarios, where the rates of the streams are bursty and may soar during peak times. In an MJoin, there are m different *join directions*, one for each stream, and for each join direction there is an associated *join order*. ith direction of the join describes how a tuple t from S_i is processed by the join algorithm, after it is fetched from the input buffer. The join order for direction i, denoted by $R_i = \{r_{i,1}, r_{i,2}, \ldots, r_{i,m-1}\}$, defines an ordered set of window indexes that will be used during the processing of $t \in S_i$. In particular, tuple t will first be matched against the tuples in window W_l, where $l = r_{i,1}$. Here, $r_{i,j}$ is the jth join window index in R_i. If there is a match, then the index of the next window to be used for further matching is given by $r_{i,2}$, and so on. For any direction, the join order consists of $m-1$ distinct window indices, i.e. R_i is a permutation of $\{1, \ldots, m\} - \{i\}$. Although there are $(m-1)!$ possible choices of orderings for each join direction, this number can be smaller depending on the join graph of the particular join at hand. We will talk more about join order selection in Section 5. Figure 1 illustrates join directions and orders for an example 3-way join. Once the join order for each direction is decided, the processing is carried out in an NLJ (nested-loop join) fashion. Since we do not focus on any particular type of join condition, NLJ is a natural choice. See Section 7 for a discussion on applying indexed processing to our approach, for special types of joins.

3. OPERATOR THROTTLING

Operator throttling is a load shedding framework for stream operators, that regulates the amount of load shedding to be performed by calculating and maintaining a throttle fraction, and relies on an in-operator load shedding technique to reduce the CPU cost of executing the operator in accordance with the throttle fraction. We denote the throttle fraction by z. It has a value in the range $(0, 1]$. Concretely, $z = \phi$ means that the in-operator load shedding technique should adjust the processing logic of the operator such that the CPU cost of executing it is reduced to ϕ times the original. As expected, this will have side-effects on the quality or quantity of the output from the operator. In the case of stream joins, applying in-operator load shedding will result in a reduced output rate. Note that the concept of operator throttling is general and applies to operators other than joins. For instance, an aggregation operator can use the throttle fraction to adjust its aggregate re-evaluation interval to shed load [19], or a data compression operator can decrease its compression ratio based on the throttle fraction [14].

3.1 Setting of the Throttle Fraction

The correct setting of the throttle fraction depends on the performance of the join operator under current system load and the incoming stream rates. We capture this as follows.

Let us denote the adaptation interval by Δ. This means that the throttle fraction z is adjusted every Δ seconds. Let us denote the tuple consumption rate of the join operator for S_i, measured for the last adaptation interval, by α_i. In other words, α_i is the tuple pop rate of the join operator for the buffer attached to S_i, during the last Δ seconds. On the other hand, let λ_i' be the tuple push rate for the same buffer during the last adaptation interval. Using α_i's and λ_i''s we capture the performance of the join operator under current system load and incoming stream rates, denoted by β, as:

$$\beta = \sum_{i=1}^{m} \alpha_i / \sum_{i=1}^{m} \lambda_i'$$

The β value is used to adjust the throttle fraction as follows. We start with a z value of 1, optimistically assuming that we will be able to fully execute the operator without any overload. At each adaptation step (Δ seconds), we update z from its old value z^{old} based on the formula:

$$z = \begin{cases} \beta \cdot z^{old} & \beta < 1 \\ min(1, \gamma \cdot z^{old}) & \text{otherwise} \end{cases}$$

If β is smaller than 1, z is updated by multiplying its old value with β, with the aim of adjusting the amount of shedding performed by the in-operator load shedder to match the tuple consumption rate of the operator to tuple production rate of the streams. Otherwise ($\beta \geq 1$), the join is able to process all the incoming tuples with the current setting of z, in a timely manner. In this latter case, z is set to minimum of 1 and $\gamma * r$, where γ is called the *boost factor*. This is aimed at increasing the throttle fraction, assuming that additional processing resources are available. If not, the throttle fraction will be readjusted during the next adaptation step. Note that, higher values of the boost factor result in being more aggressive at increasing the throttle fraction.

Figure 2 shows an example of throttle fraction adaptation from our implementation of GrubJoin using operator throttling. In this example Δ is set to 4 seconds and γ is set to 1.2. Other experimental parameters are not of interest for this example. The input stream rates are shown as a function of time using the left y-axis, and the throttle fraction z is shown as a function of time using the right y-axis. Looking at the figure, besides the observation that the z value adapts to the changing rates by following an inversely proportional trend, we also see that the reaction in the throttle fraction

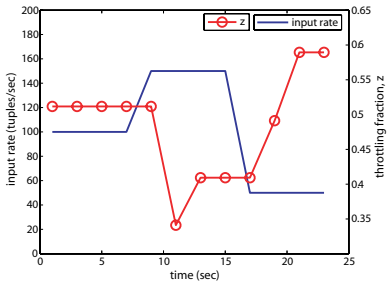

Figure 2: z adaptation example

follows the rate change events with a delay due to the length of the adaptation interval. Although in this example Δ is sufficiently small to adapt to the bursty nature of the streams, in general its setting is closely related with the length of the bursts. Moreover, the time it takes for the in-operator load shedder to perform reconfiguration in accordance with the throttle fraction is an important limitation in how frequent the adaptation can be performed, thus how small Δ can be. We discuss more about this in Section 4.2.

3.2 Buffer Capacity vs. Tuple Dropping

As opposed to stream throttling, operator throttling does not necessarily drop tuples from the incoming streams. The decision of how the load shedding will be performed is left to the in-operator load shedder, which may choose to retain all unexpired tuples within its join windows. However, depending on the size of the input buffers, operator throttling framework may still result in dropping tuples outside the join operator, albeit only during times of mismatch between the last set value of the throttle fraction and its ideal value. As an example, consider the starting time of the join, at which point we have $z = 1$. If the stream rates are higher than the operator can handle with z set to 1, then the gap between the incoming tuple rate and the tuple consumption rate of the operator will result in growing number of tuples within buffers. This trend will continue until the next adaptation step, at which time throttle fraction will be adjusted to stabilize the system. However, if during this interval the buffers fill up, then some tuples will be dropped. The buffer size can be increased to prevent tuple dropping, at the cost of introducing delay. If buffer sizes are small, then tuple dropping will be observed only during times of transition, during which throttle fraction is higher than what it should ideally be.

4. WINDOW HARVESTING

Window harvesting is an in-operator load shedding technique we develop for multi-way windowed stream joins. The basic idea behind window harvesting is to use only certain fragments of the join windows for processing, in an effort to reduce the CPU demand of the operator, as dictated by the throttle fraction. By making use of the time correlations among the streams in deciding which segments of the join windows are more valuable for output tuple generation, window harvesting aims at maximizing the output rate of the join. In the rest of this section, we first describe the fundamentals of window harvesting and then formulate window harvesting as an optimization problem.

4.1 Fundamentals

Window harvesting involves organizing join windows into a set of *basic windows* and for each join direction selecting the segments of the windows to use for performing the join.

4.1.1 Basic Windows

Each join window W_i is divided into basic windows of size b seconds. Basic windows are treated as integral units, thus there is always one extra basic window in each join window to handle tuple expiration. In other words, W_i consists of $1 + n_i$ basic windows, where $n_i = \lceil w_i/b \rceil$. The first basic window is partially full, and the last basic window contains some expired tuples (tuples whose timestamps are out of the join window's time range, i.e. $T(t) < T - w_i$). Every b seconds the first basic window fills completely and the last basic window expires totally. Thus, the last basic window is emptied and it is moved in front of the basic window list as the new first basic window.

At any time, the unexpired tuples in W_i can be though of organized into n_i *logical* basic windows, where the jth logical basic window ($j \in [1..n_i]$), denoted by $B_{i,j}$, corresponds to the ending ϑ portion of the jth basic window plus the beginning $1 - \vartheta$ portion of the $j + 1$th basic window. We have $\vartheta = \delta/b$, where δ is the time elapsed since the last basic window expiration took place. This small distinction between logical and real basic windows become handy when selecting segments of the join windows to process. Note that, accessing the tuples in a logical basic window does not require a search operation. A basic window is organized as timestamp ordered doubly linked list. As a result, for accessing tuples in $B_{i,j}$ we can perform iteration over the $i + 1$th basic window and backward iteration over the ith basic window. Concepts related with basic windows are illustrated in Figure 3.

There are two major advantages of using basic windows. First, basic windows make expired tuple management more efficient [9]. This is because the expired tuples are removed from the join windows in batches, i.e. one basic window at a time. Second, without basic windows, accessing tuples in a logical basic window will require a search operation to locate a tuple within the logical basic window's time range.

4.1.2 Configuration Parameters

There are two sets of configuration parameters for window harvesting, which together determine the segments of the windows that will be used for join processing. These are:

- *Harvest fractions*; $z_{i,j}, i \in [1..m], j \in [1..m-1]$: For the ith direction of the join, the fraction of the jth window in the join order (i.e. join window W_l, where $l = r_{i,j}$) that will be used for join processing is determined by the harvest fraction parameter $z_{i,j} \in (0, 1]$. There are $m \cdot (m-1)$ different harvest fractions. Their setting is strongly tied with the throttle fraction and the time correlations among the streams. We defer the details to Section 4.2.

- *Window rankings*; $s^v_{i,j}, i \in [1..m], j \in [1..m-1], v \in [1..n_{r_{i,j}}]$: For the ith direction of the join, we define an ordering over the logical basic windows of the jth window in the join order (i.e. join window W_l, where $l = r_{i,j}$), such that $s^v_{i,j}$ gives the index of the logical basic window that has rank v in this ordering. $B_{l,s^1_{i,j}}$ is the first logical basic window in this order, i.e. the one with rank 1. The ordering defined by $s^v_{i,j}$ values is strongly influenced by the time correlations among the streams (see Section 4.2 for details).

In summary, segments of the join window W_l, where $l = r_{i,j}$, that will be processed during the execution of the ith direction of the join is selected as follows. We first pick $B_{l,s^1_{i,j}}$, then $B_{l,s^2_{i,j}}$, and so on, until the total fraction of W_l processed reaches $z_{i,j}$. Any portion of window W_l that is not picked is not used during the execution of the ith direction of the join.

Figure 3 shows an example of window harvesting for a 3-way join, for the join direction R_1. In the example, we have $n_i = 5$ for $i \in [1..3]$. This means that we have 5 logical basic windows within each join window and as a result 6 basic windows per join window in practice. The join order for direction 1 is given as $R_1 = \{3, 2\}$. This means W_3 is the first window in the join order of R_1 (i.e. $r_{1,1} = 3$) and W_2 is the second (i.e. $r_{1,2} = 2$). We have $z_{1,1} = 0.6$. This means that $n_{r_{1,1}} \cdot z_{1,1} = 5 \cdot 0.6 = 3$ logical basic windows from $W_{r_{1,1}} = W_3$ are to be processed. Noting that we have $s^1_{1,1} = 4$, $s^2_{1,1} = 3$, and $s^3_{1,1} = 5$, the logical basic windows within W_3 that are going to be processed are selected as $3, 4$, and 5. They are marked in the figure with horizontal lines, their associated rankings written on top. The corresponding portions of the basic windows are also shaded in the figure. Note that there is a small shift between the logical basic windows and the actual basic windows (recall ϑ from Section 4.1.1). Along the similar lines, the logical basic windows 2 and 3 from W_2 are also marked in the fig-

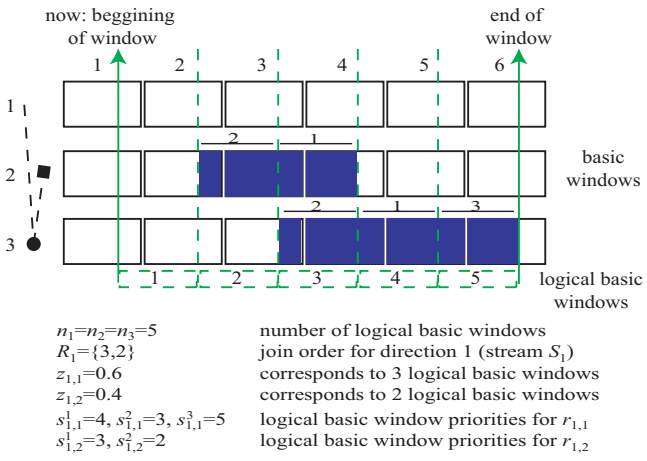

Figure 3: Example of window harvesting

ure, noting that $r_{1,2} = 2$, $z_{1,2} = 0.4$ corresponds to 2 logical basic windows, and we have $s^1_{1,2} = 3$, $s^2_{1,2} = 2$.

In the rest of this section, we describe the setting of window harvesting configuration parameters.

4.2 Configuration of Window Harvesting

Configuration of window harvesting involves setting the window ranking parameters and the harvest fraction parameters. This configuration is performed at the adaptation step, every Δ seconds.

4.2.1 Setting of Window Rankings

We set window ranking parameters $s^v_{i,j}$'s in two steps. First step is called *score assignment*. Concretely, for the ith direction of the join and the jth window in the join order R_i, that is W_l where $l = r_{i,j}$, we assign a *score* to each logical basic window within W_l. We denote the score of the kth logical basic window, which is $B_{l,k}$, by $p^k_{i,j}$. We define $p^k_{i,j}$ as the probability that an output tuple $(\ldots, t^{(i)}, \ldots, t^{(l)}, \ldots)$ has:

$$b \cdot (k-1) \leq T(t^{(i)}) - T(t^{(l)}) \leq b \cdot k$$

Here, $t^{(i)}$ denotes a tuple from S_i. This way, a logical basic window in W_l is scored based on the likelihood of having an output tuple whose encompassed tuples from S_i and S_l have an offset between their timestamps such that this offset is within the time range of the logical basic window.

The score values are calculated using the time correlations among the streams. For now, we will assume that the time correlations are given in the form of probability density functions (*pdf*s) denoted by $f_{i,j}$, where $i, j \in [1..m]$. Let us define $A_{i,j}$ as a random variable representing the difference $T(t^{(i)}) - T(t^{(j)})$ in the timestamps of tuples $t^{(i)}$ and $t^{(j)}$ encompassed in an output tuple of the join. Then $f_{i,j} : [-w_i, w_j] \rightarrow [0, \infty)$ is the probability density function for the random variable $A_{i,j}$. With this definition, we have $p^k_{i,j} = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. In practice, we develop a lightweight method for approximating a subset of these pdfs and calculating $p^k_{i,j}$'s from this subset efficiently. The details are given in Section 5 as part of the GrubJoin.

The second step of the setting of window ranking parameters is called *score ordering*. In this step, we sort the scores $\{p^k_{i,j} : k \in [1..n_{r_{i,j}}]\}$ in descending order and set $s^v_{i,j}$ to k, where v is the rank of $p^k_{i,j}$ in the sorted set of scores. If the time correlations among the streams change, then a new set of scores and thus a new assignment for the window rankings is needed. This is again handled by the reconfiguration performed at every adaptation step.

4.2.2 Setting of Harvest Fractions

Harvest fractions are set by taking into account the throttle fraction and the time correlations among the streams. First, we have to make sure that the CPU cost of performing the join agrees with the throttle fraction z. This means that the cost should be at most equal to z times the cost of performing the full join. Let $C(\{z_{i,j}\})$ denote the cost of performing the join for the given setting of the harvest fractions, and $C(\mathbf{1})$ denote the cost of performing the full join. We say that a particular setting of harvest fractions is feasible if and only if $z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\})$.

Second, among the feasible set of settings of the harvest fractions, we should prefer the one that results in the maximum output rate. Let $O(\{z_{i,j}\})$ denote the output rate of the join operator for the given setting of the harvest fractions. Then our objective is to maximize $O(\{z_{i,j}\})$. In short, we have an optimization problem:

Optimal Window Harvesting Problem:

$$\underset{\{z_{i,j}\}}{argmax} \quad O(\{z_{i,j}\})$$
$$\mathbf{s.t.} \quad z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\})$$

We now describe the formulations for functions C and O. Our formulations are similar to previous work [13, 2], with the exception that we integrate time correlations among the streams into the processing cost and output rate computations.

4.3 Formulation of $C(\{z_{i,j}\})$ and $O(\{z_{i,j}\})$

For the formulation of C, we will assume that the processing cost of performing the NLJ join is proportional to the number of tuple comparisons made per time unit. We do not include the cost of tuple insertion and removal in the following derivations, although they can be added with little effort.

The total cost C is equal to the sum of the costs of individual join directions, where the cost of performing the ith direction is λ_i times the number of tuple comparisons made for processing a single tuple from S_i. We denote the latter with C_i. Thus, we have:

$$C = \sum_{i=1}^{m} (\lambda_i \cdot C_i)$$

C_i is equal to the sum of the number of tuple comparisons made for processing each window in the join order R_i. The number of tuple comparisons performed for the jth window in the join order, that is $W_{r_{i,j}}$, is equal to the number of times $W_{r_{i,j}}$ is iterated over, denoted by $N_{i,j}$, times the number of tuples used from $W_{r_{i,j}}$. The latter is calculated as $z_{i,j} \cdot S_{i,j}$, where $S_{i,j} = \lambda_{r_{i,j}} \cdot w_{r_{i,j}}$ gives the number of tuples in $W_{r_{i,j}}$. We have:

$$C_i = \sum_{j=1}^{m-1} (z_{i,j} \cdot S_{i,j} \cdot N_{i,j})$$

$N_{i,j}$, which is the number of times $W_{r_{i,j}}$ is iterated over for evaluating the ith direction of the join, is equal to the number of partial join results we get by going through only the first $j-1$ windows in the join order R_i. We have $N_{i,1} = 1$ as a base case. $N_{i,2}$, that is the number of partial join results we get by going through $W_{r_{i,1}}$, is equal to $P_{i,1} \cdot \sigma_{i,r_{i,1}} \cdot S_{i,1}$, where $\sigma_{i,r_{i,1}}$ denotes the selectivity between W_i and $W_{r_{i,1}}$, and as before $S_{i,1}$ is the number of tuples in $W_{r_{i,1}}$. Here, $P_{i,1}$ is a *yield factor* that accounts for the fact that we only use $z_{i,j}$ fraction of $W_{r_{i,j}}$. If the pdfs capturing the time correlations among the streams are flat, then we

have $P_{i,j} = z_{i,j}$. We describe how $P_{i,j}$ is generalized to arbitrary time correlations shortly. By noting that for $j \geq 2$ we have $N_{i,j} = N_{i,j-1} \cdot P_{i,j-1} \cdot \sigma_{i,r_{i,j-1}} \cdot S_{i,j-1}$ as our recursion rule, we generalize our formulation as follows:

$$N_{i,j} = \prod_{k=1}^{j-1} \left(P_{i,k} \cdot \sigma_{i,r_{i,k}} \cdot S_{i,k} \right)$$

In the formulation of $P_{i,j}$, for brevity we will assume that $z_{i,j}$ is a multiple of $1/n_{r_{i,j}}$, i.e. an integral number of logical basic windows are selected from $W_{r_{i,j}}$ for processing. Then we have:

$$P_{i,j} = \sum_{k=1}^{z_{i,j} \cdot n_{r_{i,j}}} p_{i,j}^{s_{i,j}^k} \Big/ \sum_{k=1}^{n_{r_{i,j}}} p_{i,j}^k$$

To calculate $P_{i,j}$, we use a scaled version of $z_{i,j}$ which is the sum of the scores of the logical basic windows selected from $W_{r_{i,j}}$ divided by the sum of the scores from all logical basic windows in $W_{r_{i,j}}$. Note that $p_{i,j}^k$'s (logical basic window scores) are calculated from the time correlation pdfs as described earlier in Section 4.2.1. If $f_{i,j}$ is flat, then we have $p_{i,j}^k = 1/n_{r_{i,j}}, \forall k \in [1..n_{r_{i,j}}]$ and as a consequence $P_{i,j} = z_{i,j}$. Otherwise, we have $P_{i,j} > z_{i,j}$. This means that we are able to obtain $P_{i,j}$ fraction of the total number of matching tuples from $W_{r_{i,j}}$ by iterating over only $z_{i,j} < P_{i,j}$ fraction of $W_{r_{i,j}}$. This is a result of selecting the logical basic windows that are more valuable for producing join output. This is accomplished by utilizing the window rankings during the selection process. Recall that these rankings ($s_{i,j}^v$'s) are calculated from logical basic window scores.

We easily formulate O using $N_{i,j}$'s. Recalling that $N_{i,j}$ is equal to the number of partial join results we get by going through only the first $j-1$ windows in the join order R_i, we conclude that $N_{i,m}$ is the number of output tuples we get by fully executing the ith join direction. Since O is the total output rate of the join, we have:

$$O = \sum_{i=1}^{m} \lambda_i \cdot N_{i,m}$$

4.4 Bruteforce Solution

One way to solve the optimal window harvesting problem is to enumerate all possible harvest fraction settings assuming that the harvest fractions are set to result in selecting an integral number logical basic windows, i.e. $\forall_{\substack{i \in [1..m] \\ j \in [1..m-1]}}, z_{i,j} \cdot n_{r_{i,j}} \in \mathbb{N}$. Although straightforward to implement, this bruteforce approach results in considering $\prod_{i=1}^{m} n_i^{m-1}$ possible configurations. If we have $\forall i \in [1..m], n_i = n$, then we can simplify this as $\mathcal{O}(n^{m^2})$. As we will show in the experimental section, this is computationally very expensive due to the long time required to solve the optimization problem with enumeration, and makes it almost impossible to perform frequent adaptation. In the next section we will discuss an efficient heuristic that can find near-optimal solutions quickly, with much smaller computational complexity.

4.4.1 Example of Optimal Configuration

Figure 4 shows an example scenario illustrating the setting of window harvesting parameters optimally. In this scenario we have a 3-way join with $\lambda_1 = 300$, $\lambda_2 = 100$, $\lambda_3 = 150$, $w_1 = w_2 = w_3 = 10$, $b = 2$, and $z = 0.5$. The topmost graph on the left in Figure 4 shows the selectivities, whereas the two graphs next to it show the time correlation pdfs, $f_{2,1}$ and $f_{3,1}$. By looking at $f_{2,1}$, we can see that there is a time lag between the streams S_1 and S_2, since most of the matching tuples from these

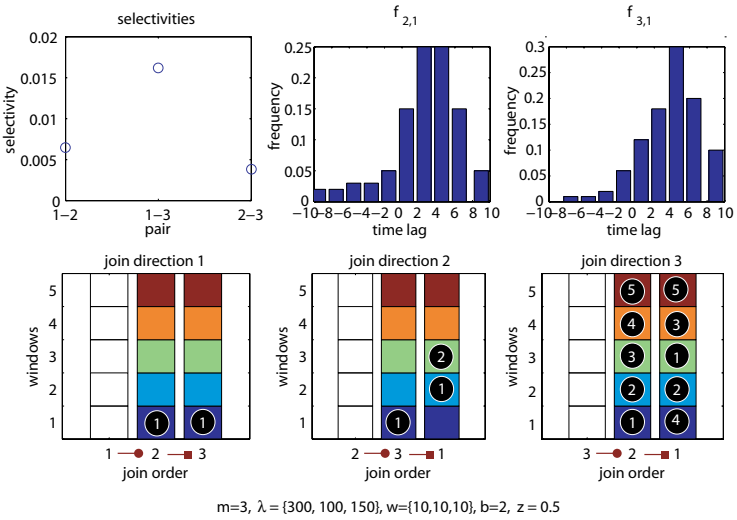

Figure 4: Optimal window harvesting example

two streams has a timestamp difference of about 4 seconds, S_2 tuple being lagged. Moreover, the probability that two tuples from S_1 and S_2 match decreases as the difference in their timestamps deviates from the 4 second time lag. By looking at $f_{3,1}$, we can say that the streams S_1 and S_3 are also unaligned, with S_3 lagging behind by around 5 seconds. In other words, most of the S_3 tuples match with S_1 tuples that are around 5 seconds older. By comparing $f_{2,1}$ and $f_{3,1}$, we can also deduce that S_3 is slightly lagging behind S_2, by around 1 seconds. As a result, our intuition tells us that the third join direction is more valuable then the others, since the tuples from other streams that are expected to match with an S_3 tuple are already within the join windows when an S_3 tuple is fetched. In this example, the join orders are configured as follows: $R_1 = \{2, 3\}, R_2 = \{3, 1\}, R_3 = \{2, 1\}$. This decision is based on the low selectivity first heuristic [20], as it will be discussed in the next section. The resulting window harvesting configuration, obtained by solving the optimal window harvesting problem by using the bruteforce approach, is shown in the lower row of Figure 4. The logical basic windows selected for processing are marked with dark circles and the selections are shown for each join direction. We observe that in the resulting configuration we have $z_{3,1} = z_{3,2} = 1$, since all the logical basic windows are selected for processing in R_3. This is inline with our intuition that the third direction of the join is more valuable than the others.[2]

5. GRUBJOIN

GrubJoin is a multi-way windowed stream join operator with built-in window-harvesting. It uses two main methods to make window harvesting work in practice. First, it employs a heuristic method to set the harvest fractions, and second it uses approximation techniques to learn the time correlations among the streams and to set the logical basic window scores based on that. In this section we describe the details of these two methods.

5.1 Heuristic Setting of Harvest Fractions

The heuristic method we use for setting the harvest fractions is greedy in nature. It starts by setting $z_{i,j} = 0, \forall i, j$. At each greedy step it considers a set of settings for the harvest fractions, called the *candidate set*, and picks the one with the highest *evaluation*

[2]See a demo at http://members.fortunecity.com/grubjoin/

metric as the new setting of the harvest fractions. Any setting in the candidate set must be a forward step in increasing the $z_{i,j}$ values, i.e. we must have $\forall i,j, z_{i,j} \geq z_{i,j}^{old}$, where $\{z_{i,j}^{old}\}$ is the setting of the harvest fractions that was picked at the end of the previous greedy step. The process terminates once a step with an empty candidate set is reached. We introduce three different evaluation metrics for deciding on the best configuration within the candidate set. In what follows, we first describe the candidate set generation and then introduce the three alternative evaluation metrics.

5.1.1 Candidate Set Generation

The candidate set is generated as follows. For the ith direction of the join and the jth window within the join order R_i, we add a new setting into the candidate set by increasing $z_{i,j}$ by $d_{i,j}$. In the rest of the paper we take $d_{i,j}$ as $1/n_{r_{i,j}}$. This corresponds to increasing the number of logical basic windows selected for processing by one. This results in $m \cdot (m-1)$ different settings, which is also the maximum size of the candidate set. The candidate set is then *filtered* to remove the settings which are infeasible, i.e. do not satisfy the processing constraint of the optimal window harvesting problem dictated by the throttle fraction z. Once a setting in which $z_{u,v}$ is incremented is found to be infeasible, then the harvest fraction $z_{u,v}$ is *frozen* and no further settings in which $z_{u,v}$ is incremented are considered in the future steps of the algorithm.

There is one small complication to the above described way of generating candidate sets. Concretely, when we have $\forall j, z_{i,j} = 0$ for the ith join direction at the start of a greedy step, then it makes no sense to create a candidate setting in which only one harvest fraction is non-zero for the ith join direction. This is because no join output can be produced from a join direction if there is one or more windows in the join order for which the harvest fraction is set to zero. As a result, we say that a join direction i is not *initialized* if and only if there is a j such that $z_{i,j} = 0$. If at the start of a greedy step, we have a join direction that is not initialized, say ith direction, then instead of creating $m-1$ candidate settings for the ith direction, we generate only one setting in which all the harvest fractions for the ith direction are incremented, i.e. $\forall j, z_{i,j} = d_{i,j}$.

Computational Complexity: In the worst case, the greedy algorithm will have $(m-1) \cdot \sum_{i=1}^{m} n_i$ steps, since at the end of each step at least one harvest fraction is incremented for a selected join direction and window within that direction. Taking into account that the candidate set can have a maximum size of $m \cdot (m-1)$ for each step, the total number of settings considered during the execution of the greedy heuristic is bounded by $m \cdot (m-1)^2 \cdot \sum_{i=1}^{m} n_i$. If we have $\forall i \in [1..m], n_i = n$, then we can simplify this as $\mathcal{O}(n \cdot m^4)$. This is much better than the $\mathcal{O}(n^{m^2})$ complexity of the exhaustive algorithm, and as we will show in the next section it has satisfactory running time performance.

5.1.2 Evaluation Metrics

The evaluation metric used for picking the best setting among the candidate settings has a tremendous impact on the optimality of the heuristic. We introduce three alternative evaluation metrics and experimentally compare their optimality in the next section. These evaluation metrics are:

• *Best Output*: The best output metric picks the candidate setting that results in the highest join output $O(\{z_{i,j}\})$.

• *Best Output Per Cost*: The best output per cost metric picks the candidate setting that results in the highest join output to join cost ratio $O(\{z_{i,j}\})/C(\{z_{i,j}\})$.

• *Best Delta Output Per Delta Cost*: Let $\{z_{i,j}^{old}\}$ denote the setting of the harvest fractions from the last step. Then the best delta output per delta cost metric picks the setting that results in the highest additional output to additional cost ratio $\frac{O(\{z_{i,j}\})-O(\{z_{i,j}^{old}\})}{C(\{z_{i,j}\})-C(\{z_{i,j}^{old}\})}$.

```
GREEDYPICK(z)
(1)  cO ← cC ← 0 {current cost and output}
(2)  ∀ 1≤i≤m , I_i ← false {initialization indicators}
(3)  ∀ 1≤i≤m, 1≤j≤m−1 , F_{i,j} ← false {frozen fraction indicators}
(4)  ∀ 1≤i≤m, 1≤j≤m−1 , z_{i,j} ← 0 {fraction parameters}
(5)  while true
(6)      bS ← 0 {best score for this step}
(7)      u ← v ← −1 {direction and window indices}
(8)      for i ← 1 to m {for each direction}
(9)          if I_i = true {if already initialized}
(10)             for j ← 1 to m − 1 {for each window in join order}
(11)                 if z_{i,j} = 1 or F_{i,j} = true {z_{i,j} is maxed or frozen}
(12)                     continue{move to next setting}
(13)                 z' ← z_{i,j} {store old value}
(14)                 z_{i,j} ← MIN(1, z_{i,j} + d_{i,j}) {increment}
(15)                 S ← EVAL(z, {z_{i,j}}, cO, cC)
(16)                 z_{i,j} ← z' {reset to old value}
(17)                 if S > bS {update best solution}
(18)                     bS ← S; u ← i; v ← j
(19)                 else if S < 0 {infeasible setting}
(20)                     F_{i,j} ← true {froze z_{i,j}}
(21)         else {if not initialized}
(22)             ∀ 1≤j≤m−1 , z_{i,j} ← d_{i,j} {increment all}
(23)             S ← EVAL(z, {z_{i,j}}, cO, cC)
(24)             ∀ 1≤j≤m−1 , z_{i,j} ← 0 {reset all}
(25)             if S > bS {update best solution}
(26)                 bS ← S; u ← i
(27)     if u = −1 {no feasible configurations found}
(28)         break{further increment not possible}
(29)     if I_u = false {if not initialized}
(30)         I_u ← true {update initialization indicator}
(31)         ∀ 1≤j≤m−1 , z_{u,j} ← d_{i,j} {increment all}
(32)     else z_{u,v} = z_{u,v} + d_{i,j} {increment}
(33)     cC = C({z_{i,j}}) {update current cost}
(34)     cO = O({z_{i,j}}) {update current output}
(35) return {z_{i,j}} {Final result}
```

```
EVAL(z, {z_{i,j}}, cO, cC)
(1)  S ← −1 {metric score of the solution}
(2)  if C({z_{i,j}}) > r · C(1) {if not feasible}
(3)      return S {return negative metric score}
(4)  switch(heuristic_type)
(5)      case BestOutput:
(6)          S ← O({z_{i,j}}); break
(7)      case BestOutputPerCost:
(8)          S ← O({z_{i,j}}) / C({z_{i,j}}); break
(9)      case BestDeltaOutputPerDeltaCost:
(10)         S ← (O({z_{i,j}}) − cO) / (C({z_{i,j}}) − cC); break
(11) return S {return the metric score}
```

Figure 5: Greedy Heuristic for setting the harvest fractions

Figure 5 gives the pseudo code for the heuristic setting of the harvest fractions. In the pseudo code the candidate sets are not explicitly maintained. Instead, they are iterated over on-the-fly and the candidate setting that results in the best evaluation metric is used as the new setting of the harvest fractions.

5.1.3 Illustration of the Greedy Heuristic

Figure 6 depicts an example illustrating the inner workings of the greedy heuristic for a 3-way join. The example starts with a setting in which $z_{i,j} = 0.2, \forall i,j$ and shows the following greedy steps of the heuristic. The harvest fraction settings are shown as 3-by-2 matrices in the figure. Similarly, 3-by-2 matrices are used (on the right side of the figure) to show the frozen harvest fractions. Initially none of the harvest fractions are frozen. In the first step a candidate set with six settings is created. In each setting one of the six harvest fractions is incremented by 0.1. As shown in the

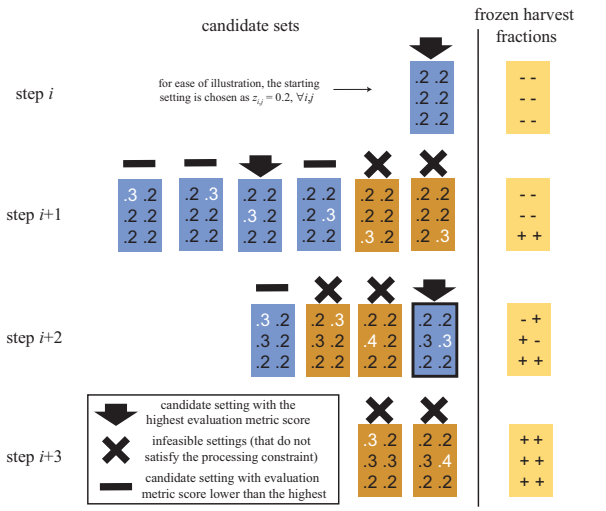

Figure 6: Illustration of the greedy heuristic

figure, out of these six settings the two are found to be infeasible, and are marked with a cross. These two settings are the ones in which $z_{3,1}$ and $z_{3,2}$ were incremented, and thus these two harvest fractions are frozen at their last values. Among the remaining four settings the one in which $z_{2,1}$ is increased is found to give the highest evaluation metric score. This setting is marked with an arrow in the figure, and forms the base setting for the next greedy step. The remaining three settings, marked with a line in the figure, are simply discarded. In the second step only four new settings are created, since two of the harvest fractions were frozen. As shown in the figure, among these four new settings two are found to be infeasible and thus two more harvest fractions are frozen. The setting marked with the arrow is found to have the best evaluation metric score and forms the basis setting for the next step. However, both of the two settings created for the next step are found to be infeasible and thus the last setting from the second step is determined as the final setting. It is marked with a frame in the figure.

5.2 Learning Time Correlations

The time correlations among the streams can be learned by monitoring the output of the join operator. Recall that the time correlations are captured by the pdfs $f_{i,j}$, where $i, j \in [1..m]$. $f_{i,j}$ is defined as the pdf of the difference $T(t^{(i)}) - T(t^{(j)})$ in the timestamps of the tuples $t^{(i)} \in S_i$ and $t^{(j)} \in S_j$ encompassed in an output tuple of the join. We can approximate $f_{i,j}$ by building a histogram on the difference $T(t^{(i)}) - T(t^{(j)})$ by analyzing the output tuples produced by the join algorithm.

This straightforward method of approximating the time correlations has two important shortcomings. First and foremost, since window harvesting uses only certain portions of the join windows, changing time correlations cannot be captured. Second, for each output tuple of the join we have to update $\mathcal{O}(m^2)$ number of histograms to approximate all pdfs, which hinders the performance. We tackle the first problem by using *window shredding*, and the second one through the use of sampling and *per stream histograms*. We now describe these two techniques.

5.2.1 Window Shredding

For a randomly sampled subset of tuples, we do not perform the join using window harvesting, but instead we use *window shredding*. We denote our *sampling parameter* by ω. On the average, for only ω fraction of the tuples we perform window shredding. ω is usually small (< 0.1). Window shredding is performed by executing the join fully, except that the first window in the join order of a join direction is processed only partially based on the throttle fraction z. The tuples to be used from such windows are selected so that they are roughly evenly distributed within the window's time range. This way, we get rid of the bias introduced in the output due to window harvesting, and can safely use the output generated from window shredding for building histograms to capture the time correlations. Moreover, since window shredding only processes z fraction of the first windows in the join orders, it respects the processing constraint of the optimal window harvesting problem.

5.2.2 Per Stream Histograms

Although the histograms used for approximating the time correlation pdfs are updated only for the output tuples generated from window shredding, the need for maintaining $m \cdot (m-1)$ histograms is still excessive and unnecessary. We propose to maintain only m histograms, one for each stream. The histogram associated with W_i is denoted by $\mathcal{L}_i$ and it is an approximation to the pdf $f_{i,1}$, i.e. the probability distribution for the random variable $A_{i,1}$ (introduced in Section 4.2.1).

Maintaining only m histograms that are updated only for the output tuples generated from window shredding introduces very little overhead, but necessitates developing a new method to calculate logical basic window scores ($p_{i,j}^k$'s) from these m histograms. Recall that we had $p_{i,j}^k = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. Since we do not maintain histograms for all pdfs ($f_{i,j}$'s), this formulation should be updated. We now describe the new method we use for calculating logical basic window scores.

We start with introducing some notations. We will assume that the histograms are equi-width histograms, although extension to other types are possible. $\mathcal{L}_i$ has a valid time range of $[-w_i, w_1]$, which is the input domain of $f_{i,1}$. Let $\mathcal{L}_i(I)$ denote the frequency for the time range I, and $\mathcal{L}_i[k]$ denote the frequency for the kth bucket in $\mathcal{L}_i$. Let $\mathcal{L}_i[k^*]$ and $\mathcal{L}_i[k_*]$ denote the higher and lower points of the kth bucket's time range, respectively. Finally, let $|\mathcal{L}_i|$ denote the number of buckets in $\mathcal{L}_i$.

From the definition of $p_{i,j}^k$, we have:

$$p_{i,j}^k = P\{A_{i,l} \in b \cdot [k-1, k]\}, \text{ where } r_{i,j} = l$$

For the case of $i = 1$, nothing that $A_{i,j} = -A_{j,i}$, we have:

$$\begin{aligned} p_{1,j}^k &= P\{A_{l,1} \in b \cdot [-k, -k+1]\} \\ &= \int_{x=-b\cdot k}^{-b\cdot(k-1)} f_{l,1}(x)\, dx \end{aligned}$$

We can approximate this using $\mathcal{L}_l$, as follows:

$$p_{1,j}^k \approx \mathcal{L}_l(b \cdot [-k, -k+1]) \quad (1)$$

For the case of $i \neq 1$, we will use the trick $A_{i,l} = A_{i,1} - A_{l,1}$:

$$\begin{aligned} p_{i,j}^k &= P\{(A_{i,1} - A_{l,1}) \in b \cdot [k-1, k]\} \\ &= P\{A_{i,1} \in b \cdot [k-1, k] + A_{l,1}\} \end{aligned}$$

Making the simplifying assumption that $A_{l,1}$ and $A_{i,1}$ are independent, we get:

$$\begin{aligned} p_{i,j}^k &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot P\{A_{i,1} \in b \cdot [k-1, k] + x\}\, dx \\ &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot \int_{y=b\cdot(k-1)+x}^{b\cdot k+x} f_{i,1}(y)\, dy\, dx \end{aligned}$$

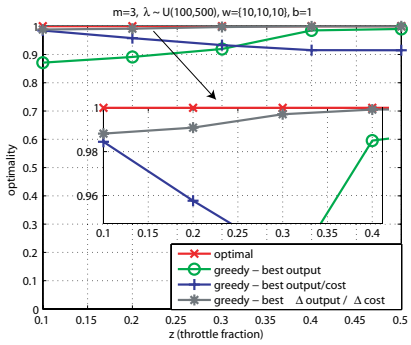

Figure 7: Effect of different evaluation metrics on optimality of greedy heuristic

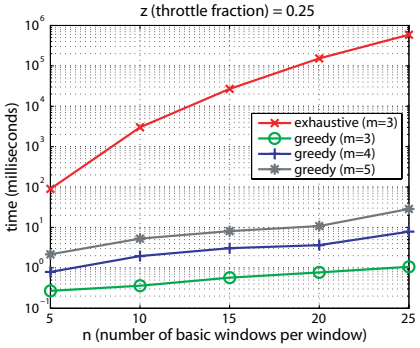

Figure 8: Running time performance with respect to m and number of basic windows

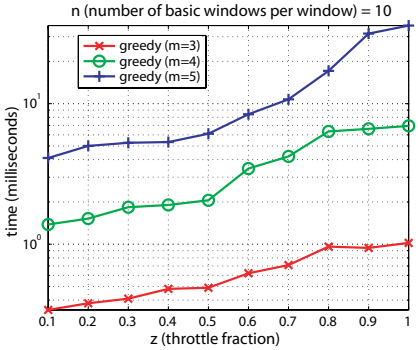

Figure 9: Running time performance with respect to m and throttle fraction z

We can approximate this using $\mathcal{L}_l$ and $\mathcal{L}_i$, as follows:

$$p_{i,j}^k \approx \sum_{v=1}^{|\mathcal{L}_l|} \left(\mathcal{L}_l[v] \cdot \mathcal{L}_i(b \cdot [k-1,k] + \frac{\mathcal{L}_l[v^*] + \mathcal{L}_l[v_*]}{2}) \right) \quad (2)$$

Equations (1) and (2) are used together to calculate the logical basic window scores by only using the m histograms we maintain. In summary, we only need to capture the pdfs $f_{i,1}, \forall i \in [1..m]$ to calculate $p_{i,j}^k$ values. This is achieved by maintaining $\mathcal{L}_i$ for approximating $f_{i,1}$. $\mathcal{L}_i$'s are updated only for output tuples generated from window shredding. Moreover, window shredding is performed only for a sampled subset of input tuples defined by the sampling parameter ω. The logical basic window scores are calculated from $\mathcal{L}_i$'s during the adaptation step (every Δ seconds). This whole process results in very little overhead during majority of the time frame of the join execution. Most of the computations are performed during the adaptation step.

5.3 Join Orders and Selectivities

The GrubJoin algorithm uses the MJoin [20] approach for setting the join orders $R_i, \forall i \in [1..m]$. This setting is based on the low selectivity first heuristic. Concretely, let U_i be the sorted set $\{\sigma_{i,j} : 1 \leq j \neq i \leq m\}$, in ascending order. Then we set $r_{i,j}$ to k, where $\sigma_{i,k}$ is the jth item in the set U_i. This technique assumes that all possible join orderings are possible, as it is in a star shaped join graph. In practice, the possible join orders should be pruned based on the join graph and then the heuristic should be applied.

Although the low selectivity first heuristic has been shown to be effective, there is no guarantee of optimality. In this work, we choose to exclude join order selection from our optimal window harvesting configuration problem, and treat it as an independent issue. We require that the join orders are set before the window harvesting parameters are to be determined. This helps cutting down the search space of the problem significantly. Using a well established heuristic for order selection and solving the window harvesting configuration problem separately is an effective technique that makes it possible to execute adaptation step much faster. This enables more frequent adaptation.

6. EXPERIMENTAL RESULTS

The GrubJoin algorithm has been implemented within our operator throttling based load shedding framework and has been successfully demonstrated as part of a large-scale stream processing prototype at IBM Watson Research. Here, we report three sets of experimental results to demonstrate the effectiveness of our approach. The first set of experiments evaluate the optimality and the runtime performance of the proposed heuristic algorithms used to set the harvest fractions. The second set of experiments use synthetically generated streams to demonstrate the superiority of window harvesting to tuple dropping, and to show the scalability of our approach with respect to various parameters, such as the number of join streams and the incoming stream rates. The third set of experiments show some anecdotal results from a real-world video join application. All experiments presented in this paper are performed on an IBM PC with 512MB main memory and 2.4Ghz Intel Pentium4 processor, using Java with Sun JDK 1.5.

6.1 Setting of Harvest Fractions

An important measure for judging the effectiveness of the three alternative metrics used in the candidate set evaluation phase of the greedy heuristic, is the optimality of the resulting setting of the harvest fractions with respect to the output rate of the join, compared to the best achievable obtained by setting the harvest fractions using the exhaustive search algorithm. The graphs in Figure 7 show optimality as a function throttle fraction z for the three evaluation metrics, namely $BestOutput$, $BestOutputPerCost$, and $BestDeltaOutputPerDeltaCost$. An optimality value of $\phi \in [0,1]$ means that the setting of the harvest fractions obtained from the heuristic yields a join output rate of ϕ times the best achievable, i.e. $O(\{z_{i,j}\}) = \phi \cdot O(\{z_{i,j}^*\})$ where $\{z_{i,j}^*\}$ is the optimal setting of the harvest fractions obtained from the exhaustive search algorithm and $\{z_{i,j}\}$ is the setting obtained from the heuristic. For this experiment we have $m = 3$, $w_1 = w_2 = w_3 = 10$, and $b = 1$. All results are averages of 500 runs. For each run, a random stream rate is assigned to each of the three streams using a uniform distribution with range $[100, 500]$. Similarly, selectivities are randomly assigned. We observe from Figure 7 that $BestOutputPerCost$ performs well only for very small z values (< 0.2), whereas $BestOutput$ performs well only for large z values ($z \geq 0.4$). $BestDeltaOutputPerDeltaCost$ is superior to other two alternatives and performs optimally for $z \geq 0.4$ and within 0.98 of the optimal elsewhere. We conclude that $BestDeltaOutputPerDeltaCost$ provides a good approximation to the optimal setting of harvest fractions. We next study the advantage of heuristic methods in terms of running time performance, compared to the exhaustive algorithm.

The graphs in Figure 8 plot the time taken to set the harvest fractions (in milliseconds) as a function of the number of logical basic windows per join window (n), for exhaustive and greedy approaches. The results are shown for 3-way, 4-way, and 5-way joins with the greedy approach and for only 3-way join with the exhaus-

tive approach. Throttle fraction z is set to 0.25 in this experiment. Note that the y-axis is in logarithmic scale. As expected, the exhaustive approach takes several orders of magnitude more time than the greedy approach. Moreover, the time taken for the greedy approach increases with increasing n and m, in compliance with its complexity of $\mathcal{O}(n \cdot m^4)$. However, what is important to observe here is the absolute values. For instance, for a 3-way join the exhaustive algorithm takes around 3 seconds for $n = 10$ and around 30 seconds for $n = 20$. Both of these values are simply unacceptable for performing fine grained adaptation. On the other hand, for $n \leq 20$ the greedy approach performs the setting of harvest fractions within 10 milliseconds for $m = 5$ and much faster for $m \leq 4$.

The graphs in Figure 9 plot the time taken to set the harvest fractions as a function of throttle fraction z, for greedy approach with $m = 3$, 4, and 5. Note that z affects the total number of greedy steps, thus the running time. The best case is when we have $z \approx 0$ and the search terminates after the first step. The worst case occurs when we have $z = 1$, resulting in $\approx n \cdot m \cdot (m-1)$ steps. We can see this effect from Figure 9 by observing that the running time performance worsens as z gets closer to 1. Although the degradation in performance for large z is expected due to increased number of greedy steps, it can be avoided by reversing the working logic of the greedy heuristic. Concretely, instead of starting from $z_{i,j} = 0, \forall i, j$ and increasing the harvest fractions gradually, we can start from $z_{i,j} = 1, \forall i, j$ and decrease the harvest fractions gradually. We call this version of the greedy algorithm *greedy reverse*. Note that greedy reverse is expected to run fast when z is large, but its performance will degrade when z is small. The solution is to switch between the two algorithms based on the value of z. We call this version of the algorithm *greedy double-sided*. It uses the original greedy algorithm when $z \leq 0.5^{(m-1)/2}$ and greedy reverse otherwise. The graphs in Figure 10 plot the time taken to set the harvest fractions as a function of throttle fraction z, for $m = 3$ with three variations of the greedy algorithm. It is clear from the figure that greedy double-sided makes the switch from greedy to greedy reverse when z goes beyond 0.5 and gets best of the both worlds, i.e. performs good for both small and large values of the throttle fraction z.

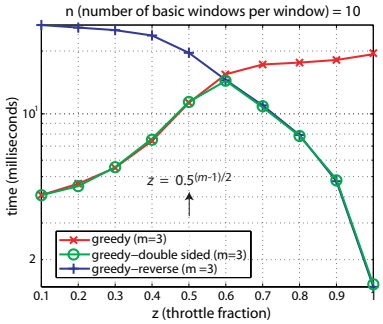

Figure 10: Running time of greedy algorithms w.r.t. z

6.2 Results on Join Output Rate

In this section, we report results on the effectiveness of GrubJoin with respect to join output rate, under heavy system load due to high rates of the incoming input streams. We compare GrubJoin with a stream throttling based approach called *RandomDrop*. In the case of RandomDrop, excessive load is shed by placing drop operators in front of input stream buffers, where the parameters of the drop operators are set based on the input stream rates using the static optimization framework of [2]. We report results on 3-way, 4-way, and 5-way joins. When not explicitly stated, the join refers to a 3-way join. The window size is set to $w_i = 20, \forall i$ and b is set to 2, resulting in 10 logical basic windows per join window. The sampling parameter ω is set to 0.1 for all experiments. The results reported in this section are from averages of several runs. Unless stated otherwise, each run is 1 minutes, and the initial 20 seconds are used for warm-up. The default value of the adaptation period Δ is 5 seconds for the GrubJoin algorithm, although we experiment with other Δ values in some of the experiments.

The join type we employ in the experiments reported in this subsection is ϵ-join. A set of tuples are considered to be matching iff their values (assuming single-valued numerical attributes) are within ϵ distance of each other. ϵ is taken as 1 in the experiments. We model stream S_i as a stochastic process $\mathbf{X_i} = \{X_i(\varphi)\}$. $X_i(\varphi)$ is the random variable representing the value of the tuple $t \in S_i$ with timestamp $T(t) = \varphi$. A tuple simply consists of a single numerical attribute with the domain $\mathcal{D} = [0, D]$ and an associated timestamp. We define $X_i(t)$ as follows:

$$X_i(\varphi) = (D/\eta) \cdot (\varphi + \tau_i) + \kappa_i \cdot \mathcal{N}(0,1) \mod D$$

In other words, $\mathbf{X_i}$ is a linearly increasing process (with wrap-around period η) that has a random Gaussian component. There are two important parameters that make this model useful for studying GrubJoin. First, the parameter κ_i, named as *deviation parameter*, enables us to adjust the amount of time correlations among the streams. If we have $\kappa_i = 0, \forall i$, then the values for the time-aligned portions of the streams will be exactly the same, i.e. the streams are identical with possible lags between them based on the setting of τ_i's. If κ_i values are large, then the streams are mostly random, so we do not have any time correlation left. Second, the parameter τ (named as *lag parameter*) enables us to introduce lags between the streams. We can set $\tau_i = 0, \forall i$ to have aligned streams. Alternatively, we can set τ_i to any value within the range $(0, \eta]$ to create non-aligned streams. We set $D = 1000$, $\eta = 50$, and vary the time lag parameters (τ_i's) and the deviation parameters (κ_i's) to generate a rich set of scenarios. Note that GrubJoin is expected to provide additional benefits when the time correlations among the streams are strong and the streams are non-aligned.

Varying λ, Input Rates

The graphs in Figure 11 show the output rate of the join as a function of the input stream rates, for GrubJoin and RandomDrop. For each approach, we report results for both aligned and non-aligned scenarios. In the aligned case, we have $\tau_i = 0, \forall i$ and in the non-aligned case we have $\tau_1 = 0, \tau_2 = 5$, and $\tau_3 = 15$. The deviation parameters are set as $\kappa_1 = \kappa_2 = 2$ and $\kappa_3 = 50$. As a result, there is strong time correlation between S_1 and S_2, whereas S_3 is more random. We make three major observation from Figure 11. First, we see that GrubJoin and Random Drop perform the same for small values of the input rates, since there is no need for load shedding until the rates reach 100 tuples/seconds. Second, we see that GrubJoin is vastly superior to RandomDrop when the input stream rates are high. Moreover, the improvement in the output rate becomes more prominent for increasing input rates, i.e. when there is a greater need for load shedding. Third, GrubJoin provides up to 65% better output rate for the aligned case and up to 150% improvement for the non-aligned case. This is because the lag-awareness nature of GrubJoin gives it an additional upper hand for sustaining a high output rate when the streams are non-aligned.

Varying Time Correlations

The graphs in Figure 12 study the effect of varying the amount of time correlations among the streams on the output rate of the join, with GrubJoin and RandomDrop for the non-aligned case. Recall that the deviation parameter κ is used to alter the amount of time correlations. It can be increased to remove the time correlations. In this experiment κ_3 is altered to study the change in output rate.

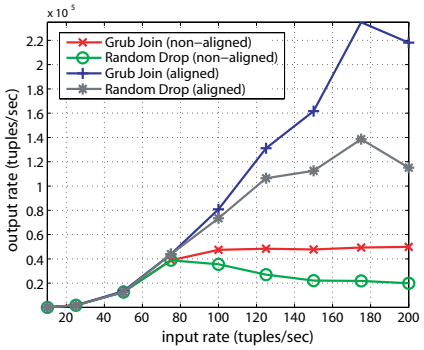

Figure 11: Effect of varying the input rates on the output rate w/wo time-lags

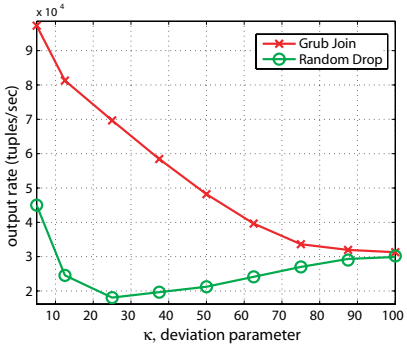

Figure 12: Effect of varying the amount of time correlations on the output rate

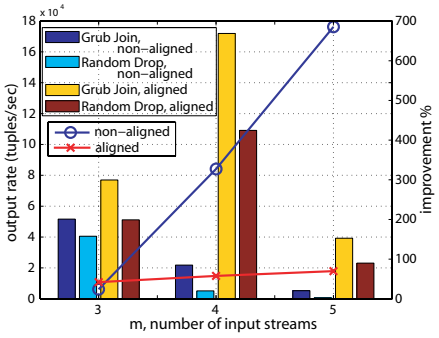

Figure 13: Effect of the # of input streams on the improvement provided by GrubJoin

The other settings are same with the previous experiment, except that the input rates are fixed at 200 tuples/second. We plot the output rate as a function of κ_3 in Figure 12. We observe that the join output rate for GrubJoin and Random Drop are very close when the time correlations are almost totally removed. This is observed by looking at the right end of the x-axis. However, for the majority of the deviation parameter's range, GrubJoin outperforms RandomDrop. The improvement provided by GrubJoin is 250% when $\kappa_3 = 25$, 150% when $\kappa_3 = 50$, and $\%25$ when $\kappa_3 = 75$. It is also worth describing the behavior of RandomDrop in this experiment. Note that as κ gets larger, RandomDrop start to suffer less from its inability to exploit time correlations by using only the usefull segments of the join windows for processing. On the other hand, when κ gest smaller, the selectivity of the join increases as a side effect and in general the output rate increases. These two contrasting factors result in a bimodal graph for RandomDrop.

Varying m, # of Input Streams

We study the effect of m (number of input streams) on the improvement provided by GrubJoin, in Figure 13. The m values are listed on the x-axis, whereas the corresponding output rates are shown in bars using the left y-axis. The improvement in the output rate (in terms of percentage) is shown using the right y-axis. Results are shown for both aligned and non-aligned scenarios. The input rates are set to 100 tuples/second for this experiment. We observe that, compared to RandomDrop, GrubJoin provides an improvement in output rate that is linearly increasing with the number of input streams. Moreover, this improvement is more prominent for non-aligned scenarios and reaches up to 700% when we have a 5-way join. This shows the importance of performing intelligent load shedding for multi-way windowed stream joins. Naturally, joins with more input streams are costlier to evaluate. For such joins, effective load shedding techniques play an even more crucial role in keeping the output rate high.

Overhead of Adaptation

In order to adapt to the changes in the input stream rates, the GrubJoin algorithm re-adjusts the window rankings and harvest fractions every Δ seconds. We now experiment with a scenario where input stream rates change as a function of time. We study the effect of using different Δ values on the output rate of the join. Recall that the default value for Δ was 5 seconds. In this scenario the stream rates start from 100 tuples/second, change to 150tuples/second after 8 seconds, and change to 50tuples/second after another 8 seconds. The graphs in Figure 14 plot the output rate of GrubJoin as a function of Δ, for different m values. Remember that larger values of m increases the running time of the heuristic used for setting the harvest fractions, thus is expected to have a profound effect on how frequent we can perform the adaptation. The Δ range used in this experiment is $[0.5, 8]$. We observe from Figure 14 that the best output rate is achieved with the smallest Δ value 0.5 for $m = 3$. This is because for $m = 3$, adaptation step is very cheap in terms of computational cost. We see that the best output rate is achieved for $\Delta = 1$ for $m = 4$ and for $\Delta = 3$ for $m = 5$. The $\mathcal{O}(n \cdot m^4)$ complexity of the adaptation step is a major factor for this change in the ideal setting of the adaptation period for larger m. In general, a default value of $\Delta = 5$ seems to be too conservative for stream rates that show frequent fluctuations. In order to get better performance, the adaptation period can be shortened. The exact value of Δ to use depends on the number of input streams, m.

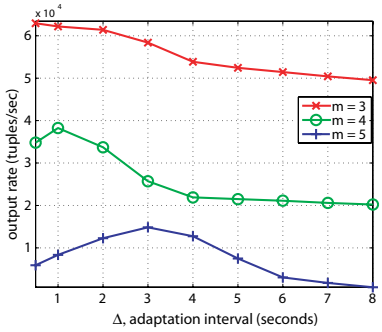

Figure 14: Effect of adaptation period on output rate

Tuple Dropping Behavior

In Section 3.2, we have mentioned that the operator throttling framework can lead to dropping tuples during times of transition, when the throttle fraction is not yet set to its ideal value. This is especially true when the input buffers are small. In the experiments reported in this section we have used very small buffers with size 10 tuples. However, as stated before, the tuple drops can be avoided by increasing the buffer size, at the cost of introducing delay. The graph in Figure 15 plots the average tuple drop rates of input buffers as a function of buffer size and input stream rates. The throttle fraction z is set to 1, 20 seconds before the average

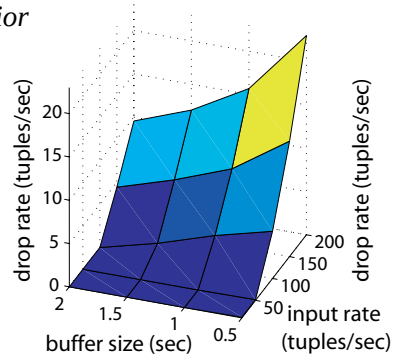

Figure 15: Tuple dropping behavior of operator throttling

drop rates are measured. The adaptation interval is set to its default value, i.e. $\Delta = 5$. As seen from the figure, 1 second buffers can cut the drop rate around 30% and 2 seconds buffers around 50% for input stream rates of around 200 tuples/second. However, in the experiments reported in this paper we chose not to use such large buffers, as they will introduce delays in the output tuples.

6.3 Video Join: A Case Study

7. DISCUSSIONS

Memory Load Shedding: This paper focuses on CPU load shedding for multi-way windowed stream joins. However, memory is also an important resource that may become a limiting factor when the join windows can not hold all the unexpired tuples due to limited memory capacity. The only way to handle limited memory scenarios is to develop *tuple admission* policies for join windows. Tuple admission policies decide which tuples should be inserted into join windows and which tuples should be removed from the join windows when there is no more space left to accommodate a newly admitted tuple. A straightforward memory conserving tuple admission policy for GrubJoin is to allow every tuple into join windows and to remove tuples from the logical basic windows that are not selected for processing such that there are no selected logical basic windows with larger indicies within the same join windows. More formally, the tuples within the logical basic windows listed in the following list can be dropped:

$$\left\{B_{i,j} : \neg\exists u, v, k \text{ s.t. } \left(r_{u,v} = i \wedge k \in [1..z_{u,v} \cdot n_i] \wedge s^k_{u,v} \geq j\right)\right\}$$

Indexed Join Processing: We have so far assumed that the join is performed in a NLJ fashion. However, special types of joins can be accelerated by appropriate index structures. For instance ϵ-joins can be accelerated through sorted trees and equi-joins can be accelerated through hash tables. As long as the cost of finding matching tuples within a join window is proportional (not necessarily linearly) to the fraction of the window used, our solution can be extended to work with indexed joins by pluging in the appropriate cost model. Note that these indexes are to be build on top of basic windows, which is an additional advantage, since tuple insertion and deletion costs are very significant for indexed joins. We have shown in our technical report [8] that, our ideas can be successfully implemented for set-based joins [15] using inverted indexes. One case that will not significantly benefit from load shedding is the equi-join scenario. In an equi-join, the time taken to find matching tuples within a join window is constant with hashtables and is independent of the window size. Thus, most of the time is spent during output tuple generation. As a result, the design space to develop intelligent CPU load shedding techniques is very small.

8. RELATED WORK

The related work in the literature on load shedding in stream join operators can be classified along four major dimensions. The first dimension defines the metric that is to be optimized when shedding load. Our work aims at maximizing the output rate of the join, also known as the MAX-subset metric [7]. Although output rate has been the predominantly used metric for join load shedding optimization [2, 8, 7, 16, 21], other metrics have also been introduced in the literature, such as the Archive-metric proposed in [7], and the sampled output rate metric introduced in [16].

The second dimension defines the constrained resource that necessitates load shedding. CPU and memory are the two major limiting resources in join processing. Thus, in the context of stream joins, works on memory load shedding [16, 7, 21] and CPU load shedding [2, 8] have received significant interest. In the case of user-defined join windows, the memory is expected to be less of an issue. Our experience shows that for multi-way joins, CPU becomes a limiting factor before the memory does. As a result, our work focuses on CPU load shedding. However, as discussed in Section 7, our framework can also be used for saving memory.

The third dimension defines the stream characteristic that is exploited for the purpose of optimizing the load shedding process. Stream rates, window sizes, and selectivities among the streams are the commonly used stream characteristics that are used for the purpose of load shedding optimization [2, 13]. However, these works do not incorporate tuple semantics into the decision process. In *semantic* load shedding, the load shedding decisions are influenced by the values of the tuples. In frequency-based semantic load shedding, tuples whose values frequently appear in the join windows are considered as more important [7, 21]. However, this approach only works for equi-joins. In time correlation-based semantic load shedding, also called age-based load shedding [16], a tuple's profitability in terms of producing join output depends on the difference between its timestamp and the timestamp of the tuple it is matched against [8, 16]. Our work takes this latter approach.

The fourth dimension defines the fundamental technique that is employed for shedding load. In the limited memory scenarios the problem is a caching one [2] and thus tuple admission/replacement is the most commonly used technique for shedding memory load [16, 7, 21]. On the other hand, CPU load shedding can be achieved by either dropping tuples from the input streams (i.e. stream throttling) [2] or by only processing a subset of join windows [8]. As we show in this paper, our window harvesting technique is superior to tuple dropping and prefers to perform the join partially, as dictated by our operator throttling framework.

To the best of our knowledge, this is the first work to address the adaptive CPU load shedding problem for multi-way stream joins. The most relevant works in the literature are the tuple dropping based optimization framework of [2], which supports multiple streams but is not adaptive, and the partial processing based load shedding framework of [8] which is adaptive but only works for two-way joins. The age-based load shedding framework of [16] is also relevant, as our work and [16] share the time correlation assumption. However, the memory load shedding techniques used in [16] are not applicable to the CPU load shedding problem and like [8], [16] is designed for two-way joins. Finally, [6] deals with the CPU load shedding problem in the context of stream joins, however the focus is on the special case in which one of the relations resides on the disk and the other one is streaming in real-time.

9. CONCLUSION

We have introduced a new load shedding framework for multi-way windowed stream joins, named operator throttling, that adjusts the amount of load shedding to be performed by setting a throttle fraction and leaves the load shedding decisions to the in-operator load shedder. We have developed window harvesting as an in-operator load shedding technique, that encourages to keep stream tuples within the join windows and sheds excessive load by processing only certain segments of the join windows, that are more useful for producing join output. Window harvesting can learn and exploit existing time correlations among the streams to prioritize segments of the join windows, in order to maximize the output rate of the join. We have designed several heuristic and approximation based techniques to make window harvesting effective in practice. Putting all these together, we have described an efficient implementation of a multi-way windowed stream join operator called GrubJoin, which has built-in window harvesting based load shedding ca-

pability and integrates with our operator throttling framework. We have conducted several experimental studies to show that GrubJoin is vastly superior to tuple dropping based alternatives when time correlations exist among the input streams, and is equally effective as tuple dropping in the absence of such correlations.

10. 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, 2003.

[2] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.

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

[4] 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.

[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] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *VLDB*, 2004.

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

[8] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. Technical report, 2005.

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

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

[11] 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.

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

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

[14] C. Pu and L. Singaravelu. Fine-grain adaptive compression in dynamically variable networks. In *IEEE ICDCS*, 2005.

[15] S.Helmer and G.Moerkotte. Evaluation of main memory join algorithms for joins with subset join predicates. In *VLDB*, 1997.

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

[17] Streambase systems. http://www.streambase.com/, May 2005.

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

[19] N. Tatbul and S. Zdonik. No false positives: Window-aware load shedding for data streams. Technical Report CS-05-06, Brown University, 2005.

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

[21] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. In *ACM SIGMOD*, 2005.