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

Buğra Gedik, *Member, IEEE,* Kun-Lung Wu, *Fellow, IEEE,*
Philip S. Yu, *Fellow, IEEE,* Ling Liu, *Senior Member, IEEE*

***Abstract*— Tuple dropping, though commonly used for load shedding in most data stream operations, is generally inadequate for multi-way, windowed stream joins. The join output rate can be unnecessarily reduced because tuple dropping fails to exploit the time correlations likely to exist among interrelated streams. In this paper, we introduce *GrubJoin* – an adaptive, multi-way, windowed stream join that effectively performs time correlation-aware CPU load shedding. GrubJoin maximizes the output rate by achieving near-optimal *window harvesting*, which picks only the most profitable segments of individual windows for the join. Due mainly to the combinatorial explosion of possible multi-way join sequences involving different window segments, GrubJoin faces unique challenges that do not exist for binary joins, such as determining the optimal window harvesting configuration in a time efficient manner and learning the time correlations among the streams without introducing overhead. 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. Our experimental results show that GrubJoin is vastly superior to tuple dropping when time correlations exist and is equally effective when time correlations are nonexistent.**

***Index Terms*— Stream Joins, Query processing, Load Shedding**

I. INTRODUCTION

In today's highly networked world, businesses often rely on time-critical tasks that require analyzing data from on-line sources and generating responses 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, sensor readings in environmental monitoring, 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 are processed, moving away from the traditional "store and then process" model of database management systems (DBMS's) to "on-the-fly processing" model of emerging data stream management systems (DSMS's). This shift has recently created a strong interest in research on DSMS-related topics, in both academia [1], [2], [3] and industry [4], [5].

In DSMS's, CPU load shedding is needed in maintaining high system throughput and timely response when the available CPU resource is not sufficient to handle the processing demands of the continual queries installed in the system, under the current rates of the input streams. Without load shedding, the mismatch between the available CPU and the query service 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 continual stream queries, in an effort to match the service rate of a DSMS to its input rate, at the cost of producing minimally degraded output.

Windowed stream joins are one of the most common, yet costliest when compared with selections or projections, operations in DSMS's. M-way, windowed stream joins are key operators used by many applications to correlate events in multiple streams [6]. For example, let us look at the following two stream join applications:

Example 1 [7] - *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 sources.

Example 2 [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 (here we have $m = 3$).

Time correlations often exist among tuples in interrelated streams, because causal events manifest themselves in these streams at different, but correlated, times. With time correlations, for pairs of matching tuples from two streams, there exists a non-flat match probability distribution, which is a function of the difference between their timestamps. For instance, in Example 2, 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 increases. The streams can also be *nonaligned*, either due to delays in the delivery path, such as network and processing delays, or due to the time of event generation effect inherent in the application. As an illustration to the nonaligned case, in Example 1, similar tuples appearing in different video streams or similar readings found in different sensor streams will have a *lag* between their timestamps,

• *B. Gedik, K-L. Wu, and P. S. Yu are with the IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532. E-mail: {bgedik,klwu,psyu}@us.ibm.com.*

• *L. Liu is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332. E-mail: lingliu@cc.gatech.edu.*

due to the time it takes for an object to pass through all cameras or all sensors

However, the lags or time correlations between tuples of different streams generally are unpredictable at the application design time and can vary significantly during runtime. This is further exacerbated by the unpredictable nature of the stream rates, which impacts the processing delays that contribute to the time lags. Therefore, it is not possible to deal with unpredictable time correlations by simply incorporating a fixed time window in the join predicates. More importantly, in order to accommodate the dynamic nature of the time correlations, the join window sizes tend to be large in user specifications, which often cause the system to be overloaded. As a result, it is important to develop a time correlation-aware load shedding framework for m-way, windowed stream joins, particularly in the face of bursty stream rates.

So far, the predominantly used approach to CPU load shedding in stream joins has been tuple dropping [9], [10]. 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 not effective in shedding CPU load for multi-way, windowed stream joins. The output rate can be unnecessarily degraded because tuple dropping does not recognize, hence fails to exploit, time correlations that are likely to exist among interrelated streams.

In this paper, we present *GrubJoin*[1]: an adaptive, multi-way, windowed stream join that effectively 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 quantity of the output produced may be decreased when load shedding is performed.

To maximize the output rate while shedding CPU load, window harvesting picks only the most profitable segments of individual join windows for the join operations while ignoring the less valuable ones, similar to farmers harvesting fruits, like strawberries, by picking only the ripest while leaving the less ripe untouched. 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 segments of different join windows, GrubJoin faces a set of challenges in performing window harvesting. These challenges are unique to multi-way, windowed stream joins and do not exist for two-way, windowed stream joins. In particular, there are three major challenges:

– First, mechanisms are needed to configure window harvesting parameters so that the throttle fraction imposed by operator throttling is respected. We should also be able to assess the optimality of these mechanisms in terms of the 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 be adapted.

In general, these challenges also apply to the binary joins. However, it is the m-way joins that necessitate the development of heuristic algorithms for window harvesting configuration and approximation techniques for learning time correlations, due to the high cost of these steps for increasing m.

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 the time correlations among the streams.

GrubJoin has been implemented and shown to be very effective in DAC (disaster assistance claim monitoring) [6], a large-scale reference application running on System S [5], a distributed stream processing middleware under development at the IBM T. J. Watson Research Center since 2003. In DAC, the join window sizes of GrubJoin must be relatively large to accommodate the unknown time correlations, which are not only workload dependent but also impacted by the upstream operator performance and processing logic. Otherwise, there would not be any join result at all when a tuple from one stream expires before the tuples, from another stream, with which it is to be joined even arrive. However, large window sizes can easily cause system overload. GrubJoin offers an effective solution to this problem. The join window sizes can be as large as needed to accommodate unpredictable time correlations, and the system overload, if it exists, will be effectively handled via adaptive time correlation-aware load shedding.

To the best of our knowledge, this is the first comprehensive work[2] 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 recognize and take advantage of the time correlation effect in join processing. In the context of two-way stream joins with limited memory, the *age-based* load shedding framework of [12] pointed out the importance of the time correlation effect and exploited it to make tuple replacement decisions. The work presented in [13] showed significant performance gains in approximate sliding window join processing when temporal correlations and reference locality are taken into account. Furthermore, in the context of traditional joins, the database literature includes join operators, such as Drag-Join [14], that capitalized on the time of data creation effect in data warehouses, which is very similar to the time correlation effect in stream joins. Moreover, similar time correlation assump-

[1] As an intransitive verb, grub means "to search laboriously by digging". It relates to the way that the most profitable segments of individual join windows are picked and processed with window harvesting in order to maximize the join output.

[2] An earlier version of this work appeared in a conference [11], which focuses on the fundamental concepts of window harvesting and operator throttling. This journal version provides the complete details of GrubJoin.

tions are used to develop load shedding techniques for two-way stream joins in [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.

The rest of this paper is organized as follows. Section II introduces stream and join models and presents the basics of multi-way stream join processing. Section III describes our operator throttling framework. Section IV describes window harvesting and formalizes the configuration of window harvesting parameters as an optimization problem. Section V describes GrubJoin, including the heuristics and approximations it uses for reconfiguration and learning purposes. Section VI presents experimental results and Section VII provides discussions. Section VIII gives related work and Section IX concludes the paper.

II. 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.

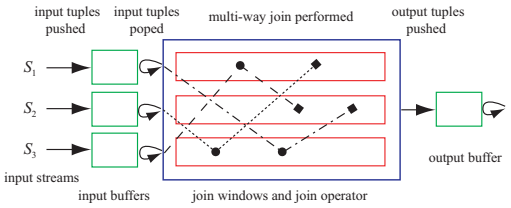

Fig. 1. Multi-way, windowed stream join processing, join directions, and join orders for each direction

direction index i	order index j = 1	2
1	$R_1 = \{$ 3	2 $\}$
2	$R_2 = \{$ 3	1 $\}$
3	$R_3 = \{$ 1	2 $\}$

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 $T \geq T(t) \geq T - w_i$. 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 fetches the tuples 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 [15]. 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*. The 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 discuss join order selection in Section V. Figure 1 illustrates join directions and orders for a 3-way join. Once the join order for each direction is decided, the processing is carried out in an NLJ (nested-loop) fashion. Since we do not focus on any particular type of join condition, NLJ is a natural choice.

Table I serves as a reference table for the commonly used notation in the paper.

III. Operator Throttling

Operator throttling is a load shedding framework for stream operators. It 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, 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 z 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 [16], or a data compression operator can decrease its compression ratio based on the throttle fraction [17].

A. 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 input 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

$S_i, i \in [1^m$	ith input stream, out of m
W_i	sliding window defined on S_i
w_i	size of W_i in time units
$B_{i,j}, j \in [1^{n_i}$	jth basic window in W_i, out of n_i
b	basic window size in time units
$T, T(t)$	current time and tuple t's timestamp
λ_i	tuple arrival rate of stream S_i
α_i	tuple consumption rate for stream S_i
β, γ	join performance and boost factor
R_i	join order for a tuple from S_i
$r_{i,j}, j \in [1^{m-1}$	jth window index in order R_i

z	overall throttle fraction
$z_{i,j}$	harvest fraction for $W_{r_{i,j}}$, for S_i tuples
$s^v_{i,j}$	index of the basic window in W_l, $l = r_{i,j}$, with rank v for S_i tuples
$p^k_{i,j}$	score of the kth basic window in W_l, $l = r_{i,j}$, (i.e., $B_{l,k}$) for S_i tuples
$A_{i,j}$	random variable representing timestamp difference of S_i and S_j tuples in join output
$f_{i,j}$	probability distribution function for $A_{i,j}$
$\mathcal{L}_i$	histogram associated with W_i
ω	window shredding sampling parameter

TABLE I

NOTATION REFERENCE TABLE — LIST OF COMMONLY USED NOTATION

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 the 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 $\gamma \cdot z^{old}$, where γ is called the *boost factor* and we have $\gamma > 1$. 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.

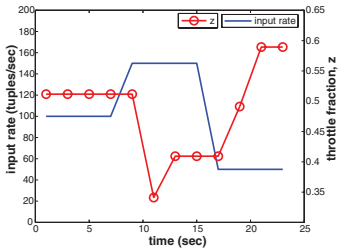

Fig. 2. Throttle fraction adaptation example.

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 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 new 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 IV-B.

B. 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. The appropriate buffer size to use is dependent on the delay requirements of the application at hand. Thus, we do not automatically adjust the buffer size. However, we study the impact of buffer size on tuple dropping behavior in Section VI-B.7.

IV. 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 the most profitable segments 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 most 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 it as an optimization problem.

A. Fundamentals

Window harvesting involves organizing join windows into a set of *basic windows* and for each join direction, selecting the most valuable segments of the windows for executing the join.

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 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. It is important to note that, a logical basic window always stores tuples belonging to a fixed time interval *relative* to the current time. This small distinction between logical and real basic windows become handy when selecting the most profitable segments of the join windows to process. Basic window related concepts are illustrated in Figure 3.

There are two major advantages of using basic windows. First, basic windows make expired tuple management more efficient [18]. 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. In general, small basic windows are more advantageous in better capturing and exploiting the time correlations. On the other hand, too small basic windows will cause overhead in join processing as well as in window harvesting configuration. This trade-off is studied in Section VI-B.4.

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, namely *harvest fractions* and *window rankings*. Before formally describing these parameters, we first give an example to provide intuition to their definitions. Consider a tuple $t^{(i)}$ arriving on stream S_i that is to be compared against the tuples in windows other than W_i, following the order defined by R_i. Let us consider the step where we are to compare $t^{(i)}$ against the window which is the jth one in the join order R_i. In other words, we are to process W_l, where $l = r_{i,j}$. There are two important decisions to make here: (1) How much of W_l do we process? (harvest fractions are used to answer this) and (2) Which basic windows within W_l do we process? (window rankings are used to answer this)

• *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. The settings of these fractions are strongly tied with the throttle fraction and the time correlations among the streams. The details will be presented in Section IV-B.

• *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.

In summary, the most profitable 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 $z_{i,j}$. Other segments of W_l that are not picked are ignored and not used during the execution of the join.

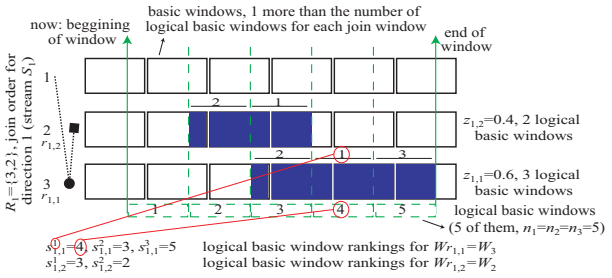

Fig. 3. Example of window harvesting.

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. In this example, harvest fractions are set to result in picking an integral number of logical basic windows for join processing. 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, with 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 IV-A.1). Along the similar lines, the logical basic windows 2 and 3 from W_2 are also marked in the figure, noting that $r_{1,2} = 2$, $z_{1,2} = 0.4$ corresponds to 2 logical basic windows, and we have $s^1_{1,2} = 3$, $s^2_{1,2} = 2$.

B. Configuration of Window Harvesting

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

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 $(\ldots, t^{(i)}, \ldots, t^{(l)}, \ldots)$ satisfy the following:

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

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 V-B.2.

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.

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 iff $z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\})$.

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:

$$\begin{array}{ll} \underset{\{z_{i,j}\}}{argmax} & O(\{z_{i,j}\}) \\ \textbf{s.t.} & z \cdot C(\mathbf{1}) \geq C(\{z_{i,j}\}). \end{array}$$

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

The formulations of the functions C and O are similar to previous work [19], [9], with the exception that we integrate time correlations among the streams into the processing cost and output rate computations.

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:

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

$N_{i,j}$, 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:

$$P_{i,j} = \sum_{k=1}^{z_{i,j}\cdot n_{r_{i,j}}} p^{s^k_{i,j}}_{i,j} \Big/ \sum_{k=1}^{n_{r_{i,j}}} p^k_{i,j}$$

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 $p^k_{i,j}$'s (logical basic window scores) are calculated from the time correlation pdfs as described earlier in Section IV-B.1. If $f_{i,j}$ is flat, then we have $p^k_{i,j} = 1/n_{r_{i,j}}, \forall k \in [1..n_{r_{i,j}}]$ 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 ($s^v_{i,j}$'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:

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

D. Brute-force 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 brute-force 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, making 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.

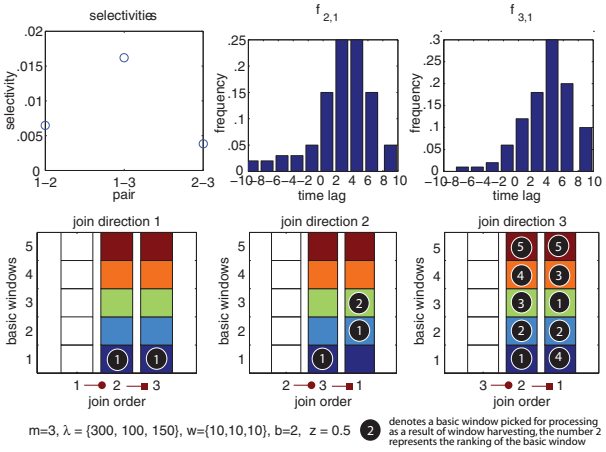

Fig. 4. Optimal window harvesting example.

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 stream rates $\lambda_1 = 300$, $\lambda_2 = 100$, $\lambda_3 = 150$, join window sizes $w_1 = w_2 = w_3 = 10$, basic window size $b = 2$, and throttle fraction $z = 0.5$. 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 two streams have 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 nonaligned, 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 second. As a result, our intuition tells us that the third join direction is more valuable than 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 [15], 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 brute-force 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.[3]

V. 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. We now describe these methods.

A. 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 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 $\forall i, j, z_{i,j} \geq z^{old}_{i,j}$, where $\{z^{old}_{i,j}\}$ is the setting of the harvest fractions that was picked at the end of the previous 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 three alternative evaluation metrics.

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

[3] See a demo at http://www-static.cc.gatech.edu/projects/disl/SensorCQ/optimizer.html

$z_{u,v}$ is *frozen* and no further settings in which $z_{u,v}$ is incremented are considered in the future steps.

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.

2) Evaluation Metrics: The evaluation metric used for picking the best setting among the candidate settings significantly impacts 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, i.e., $O(\{z_{i,j}\})$.

• *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, i.e., $O(\{z_{i,j}\})/C(\{z_{i,j}\})$.

• *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, i.e., $\frac{O(\{z_{i,j}\}) - O(\{z_{i,j}^{old}\})}{C(\{z_{i,j}\}) - C(\{z_{i,j}^{old}\})}$.

Figure 6 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.

3) Illustration of the Greedy Heuristic: Figure 5 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, out of these six settings the last two are found to be infeasible,

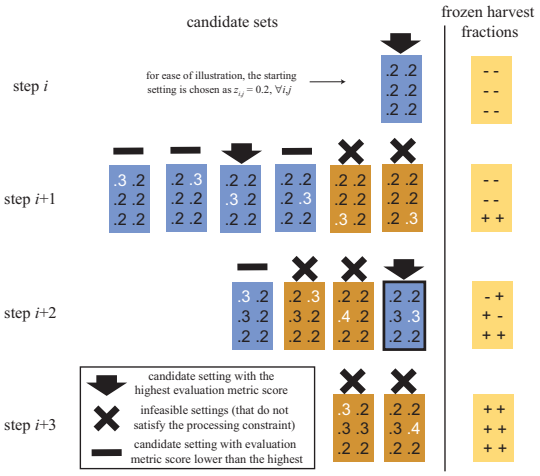

Fig. 5. Illustration of the greedy heuristic

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.

B. 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*. The rest of this section describes these two techniques.

1) Window Shredding: For a randomly sampled subset of the incoming 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

```
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}
```

Fig. 6. Greedy Heuristic for setting the harvest fractions.

the incoming 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 dictated by the throttle fraction. Note that the aim of window shredding is to capture a random sample of the join output in terms of the timestamps of the matching tuples, not in terms of the attribute values in the join result [20].

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 IV-B.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 ($p^k_{i,j}$'s) from these m histograms. Recall that we had $p^k_{i,j} = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. 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.

From the definition of $p^k_{i,j}$, we have:

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

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

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

Using $\mathcal{L}_i(I)$ to denote the frequency for the time range I in histogram $\mathcal{L}_i$, we can approximate Equation 1 as follows:

$$p^k_{1,j} \approx \mathcal{L}_l(b \cdot [-k, -k+1]). \quad (2)$$

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

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

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

$$\begin{aligned} p^k_{i,j} &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot P\{A_{i,1} \in b \cdot [k-1, k] + x\}\, dx \\ &= \int_{x=-w_l}^{w_1} f_{l,1}(x) \cdot \int_{y=b\cdot(k-1)+x}^{b\cdot k+x} f_{i,1}(y)\, dy\, dx. \end{aligned} \quad (3)$$

At this point, we will assume that the histograms are equi-width histograms, although extension to other types are possible. The valid time range of $\mathcal{L}_i$, which is $[-w_i, w_1]$ (the input domain of $f_{i,1}$), is divided into $|\mathcal{L}_i|$ number of histogram buckets. We use $\mathcal{L}_i[k]$ to denote the frequency for the kth bucket in $\mathcal{L}_i$. We use $\mathcal{L}_i[k^*]$ and $\mathcal{L}_i[k_*]$ to denote the higher and lower points of the kth bucket's time range, respectively. Finally, we can approximate Equation 3 as follows:

$$p^k_{i,j} \approx \sum_{v=1}^{|\mathcal{L}_l|} \left(\mathcal{L}_l[v] \cdot \mathcal{L}_i(b \cdot [k-1, k] + \frac{\mathcal{L}_l[v^*] + \mathcal{L}_l[v_*]}{2}) \right). \quad (4)$$

Equations (2) and (4) 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 $p^k_{i,j}$ 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.

C. Join Orders and Selectivities

The problem of optimal join ordering is NP-complete [21]. As a result, GrubJoin uses the MJoin [15] approach for setting the join orders. This setting is based on the low selectivity first heuristic. For brevity, we assumed 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.

VI. 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 System S [5] reference application, namely DAC [6]. Here, we report two sets of experimental results to demonstrate the effectiveness of our approach. We also briefly describe the use of GrubJoin in the context of DAC. 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. All experiments presented in this paper, except the DAC related ones, are performed on an IBM PC with 512MB main memory and 2.4Ghz Intel P4 processor, using Java with Sun JDK 1.5. The GrubJoins employed in DAC are written in C++ and run on 3.4Ghz Intel Xeons.

A. 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 of throttle fraction z for the three evaluation metrics, namely *Best Output* (BO), *Best Output Per Cost* ($BOpC$), and *Best Delta Output Per Delta Cost* ($BDOpDC$). 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 $BOpC$ performs well only for very small z values (< 0.2), whereas BO performs well only for large z values ($z \geq 0.4$). $BDOpDC$ 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 $BDOpDC$ 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 3-way join with the exhaustive approach. The throttle fraction z is set to 0.25. Note that the y-axis is in logarithmic scale. As expected, the exhaustive approach takes several orders of magnitude more time than the greedy one. 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 $m \leq 4$.

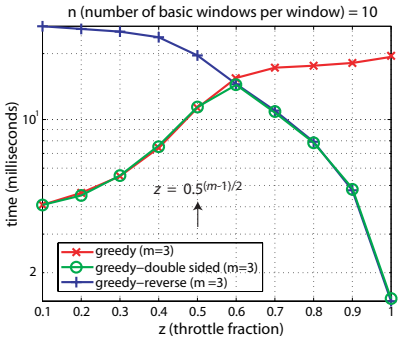

Fig. 10. Running time of greedy algorithms w.r.t. z.

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

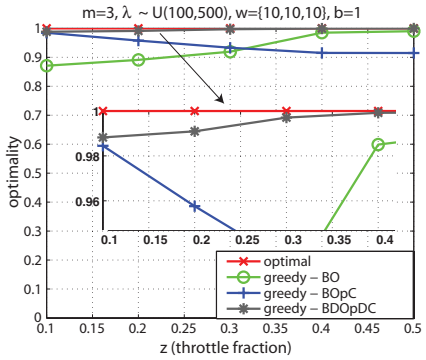

Fig. 7. Effect of different evaluation metrics on optimality of greedy heuristic.

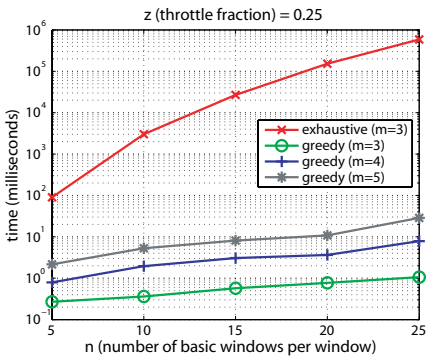

Fig. 8. Running time performance w.r.t m and number of basic windows.

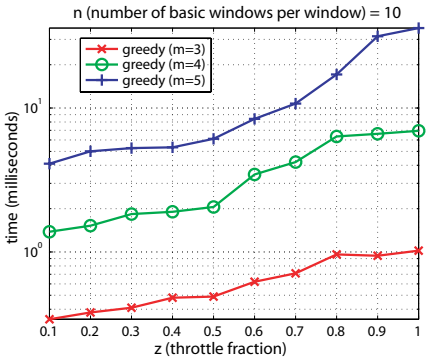

Fig. 9. Running time performance w.r.t m and throttle fraction z.

$z \leq 0.5^{(m-1)/2}$ 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: it performs good for both small and large values of the throttle fraction z.

B. 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 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 [9]. 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 study the impact of Δ on the performance of the join.

The join type 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 nonaligned 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 nonaligned.

1) 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 nonaligned scenarios. In the aligned case, we have $\tau_i = 0, \forall i$, and in the nonaligned 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 RandomDrop 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[4]. 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 nonaligned 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 nonaligned.

2) 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 nonaligned case. Recall that the deviation parameter κ is used to alter the strength 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. The other settings are the same as the previous experiment, except that the input

[4]The slight decrease in the output rates of the GrubJoins, observed from the right hand side of the figure, is due to the simulation setup, wherein the workload generation takes increasingly more processing time with increasing input rates, and the GrubJoin backs off due to its load adaptive nature.

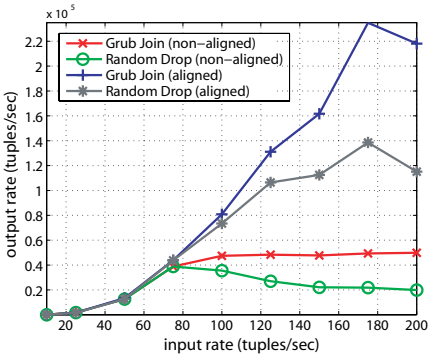

Fig. 11. Effect of varying the input rates on the output rate w/wo time-lags.

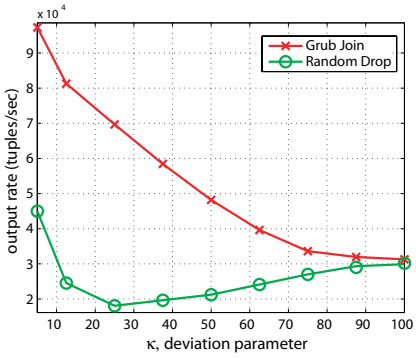

Fig. 12. Effect of varying the amount of time correlations on the output rate.

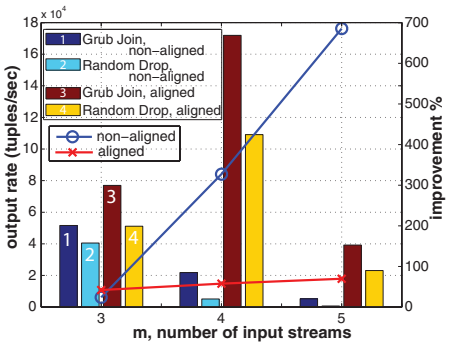

Fig. 13. Effect of the # of input streams on the improvement provided by GrubJoin.

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 $\kappa_3 = 75$. Note that, as κ gets larger, RandomDrop starts to suffer less from its inability to exploit time correlations. On the other hand, when κ gets smaller, the selectivity of the join increases as a side effect and in general the output rate increases. These contrasting factors result in a bimodal graph for RandomDrop.

3) 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 nonaligned 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 nonaligned 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.

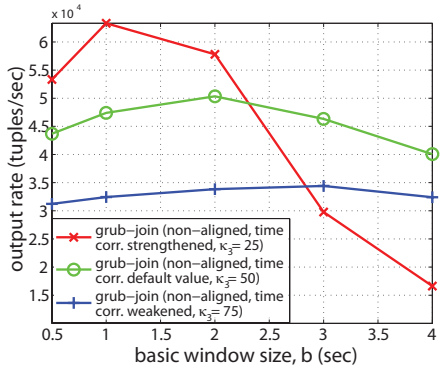

Fig. 14. Effect of basic window size on output rate.

4) Varying b, Basic Window Size: As we have mentioned earlier, small basic windows are preferable when the time correlations are strong, in which cases it is advantageous to precisely locate the profitable sections of the join windows for processing. However, small basic windows increase the total number of basic windows within a join window and thus make configuration of window harvesting costly. In order to study this effect, in Figure 14 we plot the output rate of the join as a function of the basic window size for different levels of time correlations among the streams for a 3-way join. We can see from the figure that decreasing the basic window size improves the output rate only to a certain extent and further decreasing the basic window size hurts the performance. The interesting observation here is that, the basic window value for which the best output rate is achieved varies based on the strength of the time correlations, and this optimal value increases with decreasing time correlations. This is intuitive, since with decreasing time correlations there is not much gain from small basic windows and the overhead starts to dominate. The good news is that the impact of basic window size on the output rate of the join significantly lessens when the time correlations are weakening (see the line for $\kappa_3 = 75$, which is flatter than others). As a result, it is still preferable to pick small basic window sizes. However, since the cost of setting the harvest fractions is dependent on the number of basic windows, rather than their size, it is advisable not to exceed 20 basic windows per join window based on our results in Section VI-A. The default setting of $b = 2$ used for most of the experiments in this section is a conservative choice, resulting in 10 basic windows.

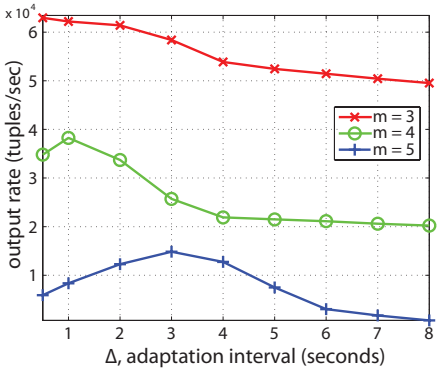

Fig. 15. Effect of adaptation period on output rate.

5) Overhead of Adaptation: In order to adapt to the changes in the input stream rates, GrubJoin 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 15 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 15 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.

6) Cost of Join Conditions: One of the motivating scenarios for CPU load shedding is the costly join conditions. We expect that the need for load shedding will become more salient with the increasing cost of the join conditions and thus GrubJoin will result in more profound improvement over tuple dropping schemes. To study the effect of join condition cost on the relative performance of GrubJoin over RandomDrop, we took the highest input stream rate at which the GrubJoin and RandomDrop perform around the same for the non-aligned scenario depicted in Figure 11 and used this rate (which is 75 tuples/sec) with different join condition costs to find out the relative improvement in output rate. We achieve different join condition costs by using a *cost multiplier*. A value of x for the cost multiplier means that the join condition is evaluated x times for each comparison made during join processing to emulate the costlier join condition. The results are presented in Table II.

cost multiplier	1	2	4	6	8	10
improvement	6%	10%	15%	22%	27%	35%

TABLE II
IMPACT OF THE JOIN CONDITION COST ON THE PERFORMANCE OF GRUBJOIN COMPARED TO RANDOMDROP

As expected, the relative improvement provided by GrubJoin increases with increasing cost multiplier. An increase of 35% is observed for a cost multiplier of 10. It is interesting to note that the increase in stream rates, as it can be observed from Figure 11, has a more pronounced impact compared to the cost of the join condition. This can be attributed to the fact that increasing stream rates not only increases the number of tuples to be processed per time unit, but it also increases the number of tuples stored within time based join windows, further increasing the cost of join processing and strain the CPU resources.

7) Tuple Dropping Behavior: In Section III-B, 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.

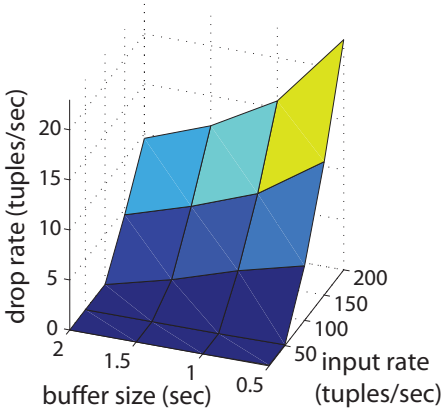

Fig. 16. Tuple dropping behavior of operator throttling

The graph in Figure 16 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 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.

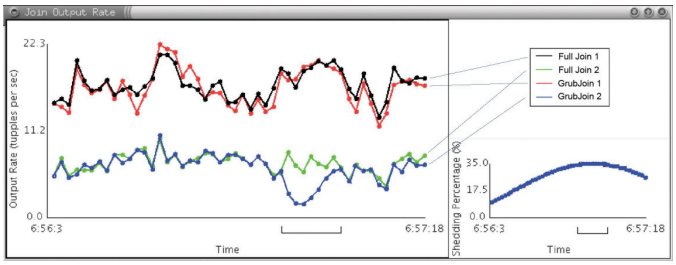

Fig. 17. GrubJoin performance in DAC

C. GrubJoin and DAC

We employ GrubJoin in DAC – a disaster assistance claim monitoring application that serves as a reference application for System S [5]. There are two join operations in DAC that use large windows, close to an hour in size. The semantics of these joins and the details of the DAC workload generation can be found in [6]. In summary, the first join correlates an unfairly processed-claims stream to a problematic-agents stream, whereas the second join correlates a suspicious-claims stream to a problematic-agents stream. We intentionally specify a larger window size for the first join, whose results are more critical for the DAC application. Both joins exhibit strong time correlations.

Here we present a partial account of the GrubJoin performance in DAC. In particular, we compare the output rates of the two GrubJoins to the output rates of the corresponding full joins. To be

able to run the full joins without any drops, we have sufficiently reduced the input stream rates of all joins. To assess GrubJoin performance under heavy load, we have imposed artificial load shedding to the GrubJoins and kept the full-joins intact. The results are reported in Figure 17, which is a snapshot taken from the DAC GUI. The small graph on the right shows the percentage of load shedding applied to the GrubJoins. Observe that, when the load shedding percentage is below 35, the GrubJoin output rates are almost exactly the same as those of the corresponding full joins. When the load shedding percentage reaches 35, then second GrubJoin's output rate drops below that of the corresponding full join. However, the same percentage of load shedding does not reduce the first GrubJoin's output rate below that of the corresponding full join. This is because the first joins use larger window sizes, yet exhibit similar time correlations as the second joins. It is clear from these results that GrubJoin can maintain high output rates under overload conditions, and as a result the application designers can specify conservative (i.e., large) windows to safely accommodate time correlations that are unpredictable at the application design time.

VII. Discussions and Ongoing Work

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 built on top of basic windows. Since tuple insertion and deletion costs are significant for indexed joins, it is more advantageous to maintain indexes on individual basic windows, which are much smaller in size compared with the entire join window. However, there is one case where the benefit of load shedding may be less compelling: equi-join. 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. Most of the execution time is spent on generating output tuples. As a result, the design space for intelligent CPU load shedding techniques is not as large.

Video Joins: We are continuing to study the impact of GrubJoin for load shedding on the quality of the join output based on real-life streams. One such study involves using GrubJoin to perform join operations on two annotated video streams: one from CNN and the other from ABC News. Each tuple in these video streams is a vector of 512 attributes produced by a high-level feature extraction algorithm. We performed a cosine similarity join on these streams to detect correlated events, using aligned streams with 1 hour join windows. The detected events were compared against those contained in the ground truth identified beforehand. The initial results show that the output of the GrubJoin has never missed any of the correlated events, whereas the output of a random-tuple-dropping-based join on average missed about 15% of the correlated events. We plan to continue with more studies using different kinds of real-life streams.

VIII. 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 is the metric to be optimized when shedding load. Our work aims at maximizing the output rate of the join, also known as the MAX-subset metric [22]. Although output rate has been the predominantly used metric for join load shedding optimization [9], [8], [22], [12], [23], other metrics have also been introduced in the literature, such as the Archive-metric proposed in [22], and the sampled output rate metric in [12].

The second dimension is 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 [12], [22], [23] and CPU load shedding [9], [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, our framework can also be used to save memory (see Section VII for more details).

The third dimension is the stream characteristic that is exploited for optimizing the load shedding process. Stream rates, window sizes, and selectivities among the streams are the commonly used characteristics that are used for load shedding optimization [9], [19]. 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 [22], [23]. However, this only works for equi-joins. In time correlation-based semantic load shedding, also called age-based load shedding [12], 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], [12]. We take this latter approach.

The fourth dimension is the fundamental technique that is employed for shedding load. In the limited memory scenarios the problem is a caching one [9] and thus tuple admission/replacement is the most commonly used technique for shedding memory load [12], [22], [23]. On the other hand, CPU load shedding can be achieved by either dropping tuples from the input streams (i.e., stream throttling) [9] 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. It performs 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 [9], 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 [12] is also relevant, as our work and [12] share the time correlation assumption. However, the memory load shedding techniques used in [12] are not applicable to the CPU load shedding problem, and like [8], [12] is designed for two-way joins. Finally, [24] 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 streamed in.

IX. Conclusion

We presented GrubJoin, an adaptive, multi-way, windowed stream join which performs time correlation-aware CPU load shedding. We developed the concept of window harvesting as an in-operator load shedding technique for GrubJoin. Window harvesting keeps the stream tuples within the join windows until they expire and sheds excessive CPU load by processing only the most profitable segments of the join windows, while ignoring the less valuable ones. Window harvesting exploits the time correlations to prioritize the segments of the join windows and maximize the output rate of the join. We developed several heuristic and approximation-based techniques to make window harvesting effective in practice for GrubJoin, which has built-in load shedding capability based on window harvesting that is integrated with an operator throttling framework. In contrast to stream throttling where load shedding is achieved by dropping tuples from the input streams, operator throttling 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. Our experimental studies show that GrubJoin is vastly superior to tuple dropping when time correlations exist among the input streams, and is equally effective in the absence of such correlations.

Acknowledgement:
The last author's research is partially supported by NSF CSR. CyberTrust and SGER.

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*, vol. 26, 2003.

[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] 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 *Conference on Innovative Data Systems Research, (CIDR)*, 2003.

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

[5] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core," in *ACM International Conference on Management of Data, (SIGMOD)*, 2006.

[6] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang, "Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S," IBM T. J. Watson Research, Tech. Rep. RC24199, 2007.

[7] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid, "Stream window join: Tracking moving objects in sensor-network databases," in *Scientific and Statistical Database Management Conference, (SSDBM)*, 2003.

[8] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "Adaptive load shedding for windowed stream joins," in *ACM Conference on Information and Knowledge Management, (CIKM)*, 2005.

[9] A. M. Ayad and J. F. Naughton, "Static optimization of conjunctive queries with sliding windows over infinite streams," in *ACM International Conference on Management of Data, (SIGMOD)*, 2004.

[10] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Very Large Databases Conference, (VLDB)*, 2003.

[11] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, "A load shedding framework and optimizations for m-way windowed stream joins," in *IEEE International Conference on Data Engineering, (ICDE)*, 2007.

[12] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *Very Large Data Bases Conference, (VLDB)*, 2004.

[13] F. Li, C. Chang, G. Kollios, and A. Bestavros, "Characterizing and exploiting reference locality in data stream applications," in *IEEE International Conference on Data Engineering, (ICDE)*, 2006.

[14] S. Helmer, T. Westmann, and G. Moerkotte, "Diag-Join: An opportunistic join algorithm for 1:N relationships," in *Very Large Databases Conference, (VLDB)*, 1998.

[15] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of m-way join queries over streaming information sources," in *Very Large Databases Conference, (VLDB)*, 2003.

[16] N. Tatbul and S. Zdonik, "A subset-based load shedding approach for aggregation queries over data streams," in *Very Large Databases Conference, (VLDB)*, 2006.

[17] C. Pu and L. Singaravelu, "Fine-grain adaptive compression in dynamically variable networks," in *IEEE International Conference on Distributed Computing Systems, (ICDCS)*, 2005.

[18] L. Golab, S. Garg, and M. T. Ozsu, "On indexing sliding windows over online data streams," in *International Conference on Extending Database Technology, (EDBT)*, 2004.

[19] J. Kang, J. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," in *IEEE International Conference on Data Engineering, (ICDE)*, 2003.

[20] S. Chaudhuri, R. Motwani, and V. Narasayya, "On random sampling over joins," in *ACM International Conference on Management of Data, (SIGMOD)*, 1999.

[21] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," vol. 9, no. 3, 1984.

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

[23] J. Xie, J. Yang, and Y. Chen, "On joining and caching stochastic streams," in *ACM International Conference on Management of Data, (SIGMOD)*, 2005.

[24] S. Chandrasekaran and M. J. Franklin, "Remembrance of streams past: Overload-sensitive management of archived streams," in *Very Large Databases Conference, (VLDB)*, 2004.

Buğra Gedik received the B.S. degree in Computer Science from the Bilkent University, Ankara, Turkey, and the Ph.D. degree in Computer Science from the College of Computing at the Georgia Institute of Technology, Atlanta, GA, USA. He is with the IBM Thomas J. Watson Research Center, currently a member of the Software Tools and Techniques Group. Dr. Gedik's research interests lie in data intensive distributed computing systems, spanning data stream processing, mobile and sensor-based data management, and data-centric overlay networks. His research focus is on developing system-level architectures and techniques to address scalability problems in distributed continual query systems and information monitoring applications. He is the recipient of the ICDCS 2003 best paper award. He has served as the program commitee co-Chair of CollaborateCom 2007 and as program committee member of several international conferences, such as ICDE, MDM, EDBT, and CollaborateCom. He was the co-chair of the SSPS'07 workshop and co-PC chair of the DEPSA'07 workshop, both on data stream processing systems.

Philip S. Yu received the B.S. Degree in E.E. from National Taiwan University, the M.S. and Ph.D. degrees in E.E. from Stanford University, and the M.B.A. degree from the New York University. He is with the IBM Thomas J. Watson Research Center and currently manager of the Software Tools and Techniques group. His research interests include data mining, Internet applications and technologies, database systems, multimedia systems, parallel and distributed processing, and performance modeling. Dr. Yu has published more than 500 papers in refereed journals and conferences. He holds or has applied for more than 300 US patents.

Dr. Yu is a Fellow of the ACM and a Fellow of the IEEE. He is associate editors of ACM Transactions on the Internet Technology and ACM Transactions on Knowledge Discovery from Data. He is on the steering committee of IEEE Conference on Data Mining and was a member of the IEEE Data Engineering steering committee. He was the Editor-in-Chief of IEEE Transactions on Knowledge and Data Engineering (2001-2004), an editor, advisory board member and also a guest co-editor of the special issue on mining of databases. He had also served as an associate editor of Knowledge and Information Systems. In addition to serving as program committee member on various conferences, he was the program chair or co-chairs of the IEEE Workshop of Scalable Stream Processing Systems (SSPS07), the IEEE Workshop on Mining Evolving and Streaming Data (2006), the 2006 joint conferences of the 8th IEEE Conference on E-Commerce Technology (CEC' 06) and the 3rd IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE' 06), the 11th IEEE Intl. Conference on Data Engineering, the 6th Pacific Area Conference on Knowledge Discovery and Data Mining, the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, the 2nd IEEE Intl. Workshop on Research Issues on Data Engineering: Transaction and Query Processing, the PAKDD Workshop on Knowledge Discovery from Advanced Databases, and the 2nd IEEE Intl. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems. He served as the general chair or co-chairs of the 2006 ACM Conference on Information and Knowledge Management, the 14th IEEE Intl. Conference on Data Engineering, and the 2nd IEEE Intl. Conference on Data Mining. He has received several IBM honors including 2 IBM Outstanding Innovation Awards, an Outstanding Technical Achievement Award, 2 Research Division Awards and the 88th plateau of Invention Achievement Awards. He received a Research Contributions Award from IEEE Intl. Conference on Data Mining in 2003 and also an IEEE Region 1 Award for "promoting and perpetuating numerous new electrical engineering concepts" in 1999. Dr. Yu is an IBM Master Inventor.

Kun-Lung Wu received the B.S. degree in E.E. from the National Taiwan University, Taipei, Taiwan, the M.S. and Ph.D. degrees in C.S. both from the University of Illinois at Urbana-Champaign. He is with the IBM Thomas J. Watson Research Center, currently a member of the Software Tools and Techniques Group. His recent research interests include data streams, continual queries, mobile computing, Internet technologies and applications, database systems and distributed computing. He has published extensively and holds many patents in these areas. Dr. Wu is a Fellow of the IEEE and a member of the ACM. He is the Program Co-Chair for the IEEE Joint Conference on e-Commerce Technology (CEC 2007) and Enterprise Computing, e-Commerce and e-Services (EEE 2007). He was an Associate Editor for the IEEE Trans. on Knowledge and Data Engineering, 2000-2004. He was the general chair for the 3rd International Workshop on E-Commerce and Web-Based Information Systems (WECWIS 2001). He has served as an organizing and program committee member on various conferences. He has received various IBM awards, including IBM Corporate Environmental Affair Excellence Award, Research Division Award, and several Invention Achievement Awards. He received a best paper award from IEEE EEE 2004. He is an IBM Master Inventor.

Ling Liu is an Associate Professor in the College of Computing at Georgia Institute of Technology. There she directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining performance, security, privacy, and data management issues in building large scale distributed computing systems. Dr. Liu and the DiSL research group have been working on various aspects of distributed data intensive systems, ranging from decentralized overlay networks, mobile computing and location based services, sensor network and event stream processing, to service oriented computing and architectures. She has published over 200 international journal and conference articles in the areas of Internet Computing systems, Internet data management, distributed systems, and information security. Her research group has produced a number of open source software systems, among which the most popular ones include WebCQ and XWRAPElite. Dr. Liu has received distinguished service awards from both the IEEE and the ACM and has played key leadership roles on program committee, steering committee, and organizing committees for several IEEE conferences, including IEEE International Conference on Data Engineering (ICDE), IEEE International Conference on Distributed Computing (ICDCS), International Conference on Web Services (ICWS), and International Conference on Collaborative Computing (CollaborateCom). Dr. Liu is currently on the editorial board of several international journals, including IEEE Transactions on Knowledge and Data Engineering, International Journal of Very Large Database systems (VLDBJ). Dr. Liu is the recipient of best paper award of WWW 2004 and best paper award of IEEE ICDCS 2003, a recipient of 2005 Pat Goldberg Memorial Best Paper Award, and a recipient of IBM faculty award in 2003, 2006. Dr. Liu's research is primarily sponsored by NSF, DARPA, DoE, and IBM.