A Load Shedding Framework and Optimizations for M-way Windowed Stream Joins

Buğra Gedik[♠◇] Kun-Lung Wu[♠] Philip S. Yu[♠] Ling Liu[◇]

[♠] Thomas J. Watson Research Center, IBM Research
[◇] CERCS, College of Computing, Georgia Tech
{bgedik,klwu,psyu}@us.ibm.com, lingliu@cc.gatech.edu

Abstract

Tuple dropping, though commonly used for load shedding in most stream operations, is inadequate for m-way, windowed stream joins. The join output rate can be overly reduced because it fails to exploit the time correlations likely to exist among interrelated streams. In this paper, we introduce GrubJoin*: an adaptive, m-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 window segments for the join. Due to combinatorial explosion of possible m-way join sequences involving window segments, m-way, windowed stream joins pose several unique challenges. We focus on addressing two of them: (1) How can we quickly determine the optimal window harvesting configuration for any m-way, windowed stream join? (2) How can we monitor and learn the time correlations among the streams with high accuracy and minimal 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.*

1 Introduction

In today's highly networked world, many applications rely on time-critical tasks that require analyzing data from online sources and generating responses in near real-time. Online data today are increasingly coming in the form of data streams, i.e., time-ordered series of events or readings. Examples include stock tickers in financial services, link statistics in networking and sensor readings in environmental monitoring. In these examples, dynamically changing, rapid data rates and stringent response time requirements force a paradigm shift in how the stream data are processed, moving away from traditional "store and then process" model of database management systems to "in-transit processing" model of data stream management systems (DSMSs). This shift has created a strong interest in DSMS-related research, in both academia [1, 4, 5] and industry [16, 13].

In a DSMS, CPU load shedding is critical in maintaining high system throughput and timely response when the available CPU is not sufficient to handle the processing 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. Such mismatch can also cause unbounded growth in system queues, further bogging down the system. In view of these problems, CPU load shedding can be broadly defined as an optimization mechanism to reduce the amount of processing for evaluating continual queries in an effort to match the service rate of a DSMS to its input rate, at the cost of producing a potentially degraded output.

Windowed stream joins are one of the most common, yet costly, operations in a DSMS [3, 10]. M-way, windowed stream joins are key operators used by many applications to correlate events in multiple streams coming from various sources. Consider the following two applications:

Example 1 [11] - *Tracking objects using multiple video (sensor) sources*: Assuming that scenes (readings) from m 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 m video (sensor) sources.

Example 2 [7] - *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 these three different sources (here $m = 3$).

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 decreases as the timestamp difference 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. The m-way video stream join in Example 1 gives an illustration of the nonaligned case, where similar tuples appearing in different video streams will have a *lag* between their timestamps, due to the time it takes for an object to pass through all cameras.

In view of the presence of time correlations among correlated streams and bursty, unpredictable stream rates, it is important to develop a time correlation-aware load shedding framework and optimization techniques for m-way, windowed stream joins. So far, tuple dropping [2, 17] has been predominantly used for CPU load shedding in stream joins. The rates of the input streams are sufficiently reduced via tuple dropping in order to sustain a stable system. However, tuple dropping is generally ineffective in shedding CPU load for m-way, windowed stream joins. The join output rate can be unnecessarily degraded because tuple dropping does not recognize, hence fails to exploit, time correlations that often exist among interrelated streams.

In an earlier work [7], we developed an adaptive load shedding scheme for two-way, windowed stream joins. It improves upon tuple dropping by using selective processing. The main idea is to decide which tuples in a window are more profitable for the join, if not all tuples can be processed due to limited CPU. The selective processing in [7] mainly depends on (1) maintaining statistics on individual window segments to learn time-based correlations between the two input streams, and (2) applying the learning results to dynamically revise the selective processing decisions.

Although our preliminary work in [7] established some foundation for load shedding in m-way, windowed stream joins, the combinatorial explosion of possible m-way join sequences involving different window segments poses several unique challenges that require more efficient learning algorithms and more scalable selective processing techniques. The techniques developed for two-way stream joins are too costly to apply directly for exploiting time correlations among multiple streams. In addition, it is inadequate to implement m-way stream joins by using a sequence of two-way stream joins. Such an implementation is memory intensive due to intermediate results. Moreover, time correlation detection on any two streams is insufficient to capture the global view of the time correlations, resulting in less accurate time correlation detection, leading to less effective load shedding.

In view of the above challenges, in this paper we present a general framework and a set of optimization techniques for performing time correlation-aware CPU load shedding for m-way, windowed stream joins. Our framework consists of three key functional components: *window harvesting*, *operator throttling*, and *window partitioning*. We develop *GrubJoin*[1] – an m-way, windowed stream join algorithm, which implements these three functional components. While shedding load, GrubJoin maximizes the output rate by achieving near-optimal window harvesting within an operator throttling scheme built on top of the window partitioning core.

GrubJoin divides each join window into multiple, small-sized segments of *basic windows* to efficiently implement time correlation-aware load shedding. Our operator throttling scheme performs load shedding within the stream operator, i.e., regulating the amount of work performed by the join. This requires altering the processing logic of the m-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. 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, we develop window harvesting optimization that picks only the most profitable basic windows of individual join windows for the join while ignoring the less valuable ones, similar to farmers harvesting fruits, like strawberries, by picking only the ripest while leaving the less ripe untouched.

One key challenge of GrubJoin is: How do we implement window harvesting efficiently in view of the combinatorial explosion of possible m-way join sequences? Another key challenge is: How do we learn the time correlations among all the $m^2/2$ pairs of streams with little overhead? We tackle the first challenge by developing fast greedy heuristics for making near-optimal window harvesting decisions, and the second by employing low-overhead approximation techniques that only maintain m cross-stream statistics to capture the time correlations among the $m^2/2$ pairs of streams. As a result, GrubJoin is very efficient in both window harvesting and time-correlation learning, enabling it to quickly react and timely adapt to fast-changing stream rates.

To the best of our knowledge, this is the first work on time correlation-aware CPU load shedding for m-way, windowed stream joins that are adaptive to the dynamically changing input stream rates. We would like to point out that the *age-based* load shedding framework in [15] is the first one that recognizes the time correlation effect on making tuple replacement decisions for two-way stream joins with limited memory. Furthermore, in the context of traditional joins, the database literature includes join operators, such as Drag-Join [12], that capitalized on the time of data creation effect in data warehouses, which is similar to the time correlation effect in stream joins.

2 Preliminaries

In this section, we present our window-based stream join model, introduce some notations, and describe the basics of m-way, windowed stream join processing.

We denote the ith input stream by S_i, where $i \in [1..m]$

[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.

and $m \geq 2$ is 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 tuples/sec) of a stream S_i by λ_i.

An m-way stream join operator has m join windows, as shown in the 3-way join example of Fig. 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 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 result tuples into the output buffer.

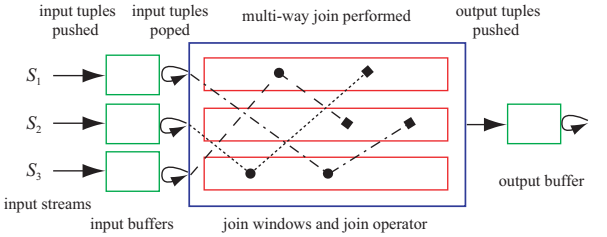

Figure 1: M-way, windowed stream join processing, join directions, join orders for each direction

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

The GrubJoin algorithm can be seen as a descendant of MJoin [19]. MJoins have been shown to be effective for fine-grained adaptation and are suitable for the environments where the stream rates are bursty. 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 join direction 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. We use the MJoin [19] approach for setting the join orders. This setting is based on the low-selectivity-first heuristic. Once the join orders are 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. Fig. 1 illustrates join directions and orders for a 3-way join.

3 Operator Throttling

Operator throttling is a load shedding framework for general stream operators, such as stream join, aggregation [18] and other operators. It regulates the amount of load shedding by 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 $(0, 1]$. The in-operator load shedding technique will 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/quantity of the output. In stream joins, the side-effect is a reduced output rate, i.e. a subset result.

Setting of the Throttle Fraction

The setting of the throttle fraction depends on the join performance 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 join operator under current system load, 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 load shedding to match the tuple consumption and arrival rates. 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*. This is aimed at increasing the throttle fraction z, assuming that additional processing resources are available. If not, the throttle fraction will be readjusted during the next adaptation step.

4 Window Harvesting

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. Window harvesting maximizes the join output rate by using the time correlations among the streams to decide which window segments are most valuable for output generation.

4.1 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 join execution.

4.1.1 Partitioning into 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. Note that a logical basic window always stores tuples belonging to a fixed time interval *relative* to the current time. This distinction between logical and real basic windows becomes handy when we are selecting the most profitable window segments for the join.

There are two advantages of using basic windows. First, basic windows make expired tuple management more efficient [9], 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.

4.1.2 Configuration Parameters

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

• *Harvest fractions*; $z_{i,j}, i \in [1..m], j \in [1..m-1]$: For the ith direction of the join, the fraction of the jth window in the join order (i.e., join window W_l, where $l = r_{i,j}$) that will be used for join processing is determined by the harvest fraction parameter $z_{i,j} \in (0, 1]$. There are $m \cdot (m-1)$ different harvest fractions. 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 4.2.

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

In summary, the most profitable segments of the join window W_l ($l = r_{i,j}$) 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 for the join execution.

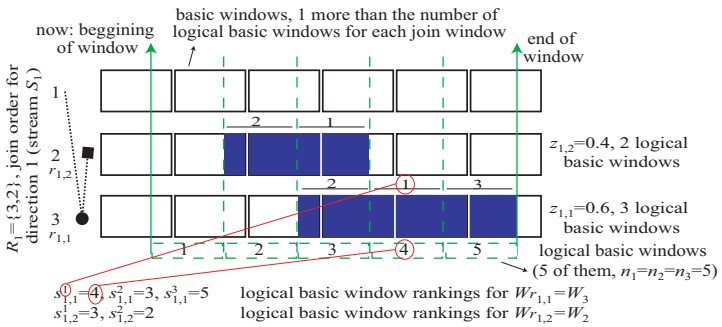

Figure 2: Example of window harvesting.

Fig. 2 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 such that an integral number of logical basic windows are picked 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 4.1.1). 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 $s^1_{1,2} = 3$, $s^2_{1,2} = 2$.

4.2 Configuration of Window Harvesting

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

4.2.1 Setting of Window Rankings

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

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

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

The score values are calculated using the time correlations among the streams. For now, we will assume that the time correlations are given in the form of probability density functions (*pdf*s) denoted by $f_{i,j}$, where $i, j \in [1..m]$. Let us define $A_{i,j}$ as a random variable representing the difference $T(t^{(i)}) - T(t^{(j)})$ in the timestamps of tuples $t^{(i)}$ and $t^{(j)}$ encompassed in an output tuple of the join. Then $f_{i,j} : [-w_i, w_j] \rightarrow [0, \infty)$ is the probability density function for the random variable $A_{i,j}$. With this definition, we have $p^k_{i,j} = \int_{b\cdot(k-1)}^{b\cdot k} f_{i,r_{i,j}}(x)dx$. In practice, we develop a lightweight method for approximating a subset of these pdfs and calculating $p^k_{i,j}$'s from this subset efficiently. The details are given in Section 5.2.2.

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 a new assignment for the window rankings is needed. This is handled by the reconfiguration performed at every adaptation step.

4.2.2 Setting of Harvest Fractions

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

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

The formulations of the functions C and O are given in our technical report [8]. Our formulations are similar to previous work [14, 2], with the exception that we integrate time correlations into the processing cost and output rate computations.

4.3 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 such that an integral number logical basic windows are selected, 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 problem with enumeration, making it impossible to perform frequent adaptation.

5 GrubJoin

GrubJoin is an m-way, windowed stream join operator with built-in window-harvesting. It uses two key methods to make window harvesting work efficiently in practice. First, it employs a fast heuristic method to set the harvest fractions, making possible frequent rate adaptation with little overhead (see Section 6.2.4). Second, it uses approximation and sampling techniques to learn the time correlations among the streams and to set the logical basic window scores based on that. These two methods make GrubJoin efficient, enabling it to not only outperform tuple dropping when time correlations exist among the streams, but also perform competitively when there are no time-correlations (see Section 6.2.2).

5.1 Heuristic Setting of Harvest Fractions

The heuristic method we use for setting the harvest fractions is greedy in nature. It starts by setting $z_{i,j} = 0, \forall i, j$. At each greedy step it considers a set of settings for the harvest fractions, called the *candidate set*, and picks the one with the highest *evaluation 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 the best configuration within the candidate set. In what follows, we first describe the candidate set generation and then introduce three alternative evaluation metrics.

5.1.1 Candidate Set Generation

For the ith direction of the join and the jth window within the join order R_i, we add a new setting into the candidate set by

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

There is one small complication to the above approach to generating candidate sets. When we have $\forall j, z_{i,j} = 0$ for the ith join direction at the start of a greedy step, it makes no sense to create a candidate setting in which only one harvest fraction is non-zero. This is because no join output can be produced from a join direction if there are 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., $\forall j, z_{i,j} = d_{i,j}$.

The computational complexity of the greedy algorithm is given by $m \cdot (m-1)^2 \cdot \sum_{i=1}^{m} n_i$ (see our technical reprt for details [8]). If we have $\forall i \in [1..m], n_i = n$, this can be simplified as $\mathcal{O}(n \cdot m^4)$. This is much better than the $\mathcal{O}(n^{m^2})$ complexity of the bute-force algorithm, and as we will show in the next section it has satisfactory running time performance.

5.1.2 Evaluation Metrics

We introduce three alternative evaluation metrics and experimentally compare their optimality in the Section 6.

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

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

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

5.2 Learning Time Correlations

The time correlations among the streams are 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.

This straightforward method of approximating the time correlations has two 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*. These are discussed next.

5.2.1 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 average, for only ω fraction of 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. 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.

5.2.2 Per Stream Histograms

The need to maintain $m \cdot (m-1)$ histograms is 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 $A_{i,1}$ that was introduced in Section 4.2.1.

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

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

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

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

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

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:

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

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

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

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

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

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:

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

Equations (2) and (4) are used to calculate the logical basic window scores by only using the m histograms we maintain.

5.2.3 Cost of Time Correlation Learning

In summary, we only need to capture m pdfs ($f_{i,1}, \forall i \in [1..m]$) to calculate the logical basic window scores ($p_{i,j}^k$). This is achieved by maintaining the histogram $\mathcal{L}_i$ for approximating the pdf $f_{i,1}$. The m histograms are updated only for output tuples generated from window shredding. Moreover, window shredding is performed only for a sampled subset of incoming tuples defined by the sampling parameter ω. The logical basic window scores are calculated using the m histograms during the adaptation step (every Δ seconds). This whole process generates very little overhead. If time-correlations do not exist, the logical basic window scores are close to each other and GrubJoin reduces to processing a random subset of the basic windows. Even in these extreme cases, GrubJoin is able to perform equally well as tuple dropping (see Section 6.2.2), thanks to the low overhead of window harvesting configuration and time correlation learning.

6 Experimental Results

The GrubJoin algorithm has been implemented and successfully demonstrated as part of System S [13], a large-scale stream processing prototype, at IBM Research. Here, we report two sets of experimental results to demonstrate the effectiveness of our approach. The first set evaluates the optimality and the runtime performance of the proposed heuristic algorithms used to set the harvest fractions. The second demonstrates the superiority of window harvesting to tuple dropping, shows the scalability of our approach with respect to various workload and system parameters, and illustrates that the overhead of adaptation is small.

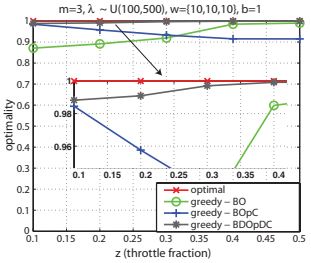

Figure 3: Effect of different evaluation metrics on optimality of greedy heuristic.

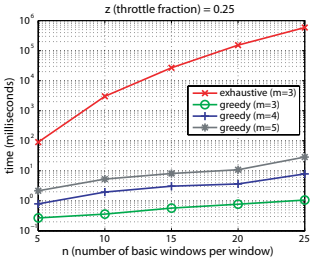

Figure 4: Running time performance w.r.t m and number of basic windows.

6.1 Setting of Harvest Fractions

To evaluate the effectiveness of the three alternative metrics for setting the harvest fractions, we measure the optimality of the resulting settings with respect to the join output rate, compared to the best achievable obtained by using the brute-force algorithm. The graphs in Fig. 3 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 brute-force 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 Fig. 3 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 brute-force algorithm.

The graphs in Fig. 4 plot the time taken to set the harvest fractions (in msecs) 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, 4, 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. 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 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 msecs for $m = 5$ and much faster for $m \leq 4$, enabling fine grained adaptation.

The graphs in Fig. 5 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 Fig. 5 by observing that the running time performance worsens as z gets closer to 1. Although the degradation in performance for large z is expected due to increased number of greedy steps, it can be avoided by reversing the working logic of the greedy heuristic. Concretely, instead of starting from $z_{i,j} = 0, \forall i, j$, and increasing the harvest fractions gradually, we can start from $z_{i,j} = 1, \forall i, j$, and decrease the harvest fractions gradually. We call this version of the greedy algorithm *greedy reverse*. Note that greedy reverse is expected to run fast when z is large, but its performance will degrade when z is small. The solution is to switch between the two algorithms based on the value of z. We call this version of the algorithm *greedy double-sided*. It uses the original greedy algorithm when $z \leq 0.5^{(m-1)/2}$ and greedy reverse otherwise. Performance results on greedy double-sided can be found in our technical report [8].

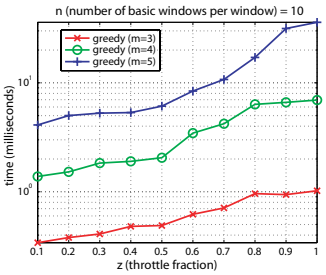

Figure 5: Running time performance wrt m, throttle fraction.

6.2 Results on Join Output Rate

In this section, we report results on the effectiveness of GrubJoin with respect to join output rate, under heavy system load due to high rates of the incoming input streams. We compare GrubJoin with the *RandomDrop* approach. 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 [2]. We report results on 3-way, 4-way, and 5-way joins. When not explicitly stated, the join refers to a 3-way join. The window size is set to $w_i = 20, \forall i$, and b is set to 2, resulting in 10 logical basic windows per join window. The sampling parameter ω is set to 0.1 for all experiments. The results reported in this section are from averages of several runs. Unless stated otherwise, each run is 1 minute, 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 in Section 6.2.4.

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 consists of a single numerical attribute with the domain $\mathcal{D} = [0, D]$ and an associated timestamp. We define $X_i(t)$ as:

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

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 time correlations.

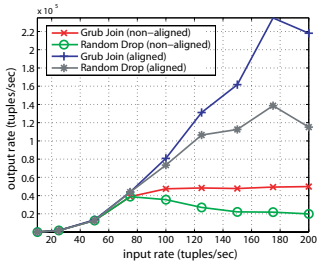

Figure 6: Effect of varying the input rates on the output rate w/wo time-lags.

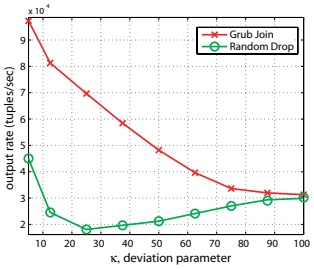

Figure 7: Effect of varying the amount of time correlations on the output rate.

6.2.1 Varying λ, Input Rates

The graphs in Fig. 6 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 Fig. 6. 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/sec. Second, we see that GrubJoin is vastly superior to RandomDrop when the input stream rates are high. Moreover, the improvement in the output rate becomes more prominent for increasing input rates, i.e., when there is a greater need for load shedding. Third, GrubJoin provides up to 65% better output rate for the aligned case and up to 150% improvement for the nonaligned case. This is

because the lag-awareness nature of GrubJoin gives it an additional upper hand in sustaining a high output rate when the streams are nonaligned.

6.2.2 Varying Time Correlations

The graphs in Fig. 7 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 with the previous experiment, except that the input rates are fixed at 200 tuples/sec. We plot the output rate as a function of κ_3 in Fig. 7. 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.

6.2.3 Varying m, # of Input Streams

We study the effect of m on the improvement provided by GrubJoin, in Fig. 8. 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/sec 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 m. 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 m-way joins. Naturally, joins with more input streams are costlier to evaluate. For such joins, effective load shedding techniques play a more crucial role in keeping the output rate high.

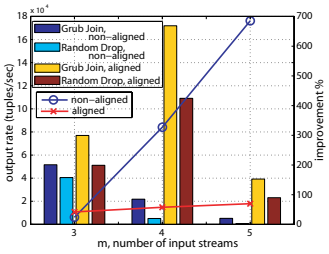

Figure 8: Effect of the # of input streams on the improvement provided by GrubJoin.

6.2.4 Adaptation Overhead

In order to adapt to the changes in the input stream rates, the GrubJoin algorithm re-adjusts the window rankings and harvest fractions every Δ seconds. We now experiment with a scenario where input stream rates change as a function of time. We study the effect of using different Δ values on the output rate of the join. In this scenario the stream rates start from 100 tuples/sec, change to 150tuples/sec after 8 seconds, and change to 50tuples/sec after another 8 seconds.

The graphs in Fig. 9 plot the output rate of GrubJoin as a function of Δ, for different m values. Remember that larger values of m increase the running time of the heuristic used for setting the harvest fractions, and thus have a profound effect on how frequent we can perform the adaptation. The Δ range used in this experiment is $[0.5, 8]$ seconds. We observe from Fig. 9 that the best output rate is achieved with the smallest Δ value of 0.5 for $m = 3$. This is because for $m = 3$ the 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 Δ for larger m.

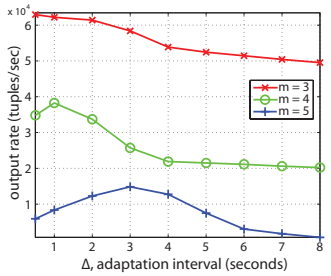

Figure 9: Effect of adaptation period on output rate.

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 low cost of window harvesting configuration in GrubJoin makes it possible to use smaller Δ values to perform more frequent adaptation. As a result, GrubJoin achieves additional gain in output rate when the input rates are fast changing and require frequent adaptation. The optimal value of Δ to use depends on the number of input streams.

7 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 [6]. Besides the output rate metric for join load shedding optimization [2, 7, 6, 15, 20], other metrics have also been introduced in the literature, such as the Archive-metric [6], and the sampled output rate metric [15].

The second dimension is the constrained resource that forces load shedding. CPU and memory are the two major limiting resources in join processing. In the context of stream joins, works on memory load shedding [15, 6, 20] and CPU load shedding [2, 7] 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 m-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 [8].

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 commonly used for load shedding optimization [2, 14]. 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 [6, 20]. However, this only works for equi-joins. In time correlation-based semantic load shedding, also called age-based load shedding [15], 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 [7, 15]. Our work takes this latter approach.

The fourth dimension is the technique used for shedding load. In the limited memory scenarios the problem is a caching one [2] and thus tuple admission/replacement is the most commonly used technique [15, 6, 20]. On the other hand, CPU load shedding can be achieved by either dropping tuples from the input streams [2] or by only processing a subset of the join windows [7]. As we show in this paper, our window harvesting technique is superior to tuple dropping and prefers to perform the join partially, as dictated by our operator throttling framework.

8 Conclusion

We presented GrubJoin, an adaptive, m-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 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 maximizes the output rate of the join. We developed several heuristic and approximation-based techniques to make window harvesting effective in practice for m-way, windowed stream joins. Our experimental studies show that GrubJoin is vastly superior to tuple dropping.

References

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

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

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

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

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[7] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *ACM CIKM*, 2005.

[8] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. GrubJoin: An adaptive multi-way windowed stream join with time correlation-aware cpu load shedding. Technical Report GIT-CERCS-05-19, Georgia Tech, 2005.

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

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

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

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

[13] 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 SIGMOD*, 2006.

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

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

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

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

[18] N. Tatbul and S. Zdonik. A subset-based load shedding approach for aggregation queries over data streams. In *VLDB*, 2006.

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

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