iJoin: Importance-aware Join Approximation over Data Streams

Dhananjay Kulkarni[1] and Chinya V. Ravishankar[2]

[1] Boston University, Metropolitan College,
Department of Computer Science, Boston MA 02215, USA,
kulkarni@bu.edu,
WWW home page: http://people.bu.edu/kulkarni/
[2] University of California - Riverside, Department of Computer Science,
Riverside CA 92521, USA,
ravi@cs.ucr.edu,
WWW home page: http://www.cs.ucr.edu/~ravi

Abstract. We consider approximate join processing over data streams when memory limitations cause incoming tuples to overflow the available space, precluding exact processing. Selective eviction of tuples (*load-shedding*) is needed, but is challenging since data distributions and arrival rates are unknown a priori. Also, in many real-world applications such as for the stock market and sensor-data, different items may have different *importance* levels. Current methods pay little attention to load-shedding when tuples bear such importance semantics, and perform poorly due to premature tuple drops and unproductive tuple retention. We propose a novel framework, called iJoin, which overcomes these drawbacks, and also provides tuples a fair chance in being part of the join result. Our load-shedding scheme for iJoin maximizes the total importance of join results, and allows reconfiguration of tuple-importance. We also show how to trade off load-shedding overhead and approximation-error. Our experiments show that iJoin has the best performance, and is practical.

1 Introduction

Various sources, such as news feeds[1], stock traders [2–4], sensors [5, 6], and online bidding systems [7] generate data continuously. Recent literature [8] has classified such data as *data streams* and there has been extensive research in querying and processing data streams. Since techniques used in traditional relational databases [9] are not directly applicable in the context of data streams, there has been a significant interest in building a Data Stream Management System [8, 10–15]. Only recently, there have been some vendors [16, 17] who have started developing a general-purpose DSMS.

There exist numerous challenges in storing and querying a data stream in a DSMS. Since data stream are continuous and unbounded, it is impractical to store the entire data during query processing. Generally, a time-based *sliding window* [18] is specified over the data stream and this defines the subset of the data stream processed by the query. A *continuous query* (CQ) [18] (or standing query) is executed over successive instances of the sliding window. Though sliding widows CQs provide a way to limit processing load, the DSMS may have to still deal with bursty data arrival and unpredictable data distributions. Most static techniques [19, 20] are impractical because they assume that input size, arrival rates, or input value distributions are constant - which is rarely the case. Thus, we need to address the dynamic nature of data streams while query processing.

Windowed Joins [20] are an important class of queries over data streams, because they help identify important correlations across multiple data streams. Joins are also *stateful* operations, which means that all tuples that arrive within the time window need to be stored, in order to compute the *exact* join result. In practice, the data stream arrival rates may fluctuate and overwhelm the available memory. In other instances, for example in sensor devices, the available memory itself might be very small in comparison to the volume of arriving tuples. Of course, when the time-window *slides*, some tuples *expire* and these expired tuples make way for future in-coming tuples. However, when the memory is already full and a new tuples arrives, there is no recourse but to 'drop' tuples according to some *load-shedding policy*. When the join is executed over the reduced window (after dropping tuples), the join result is a subset of the exact join. Such a join, with some missing output-tuples, is said to be *approximate*. Our work focusses on generating the best approximation (least error) for a join, by proposing a novel load-shedding technique to decide which tuples to drop.

Previous load-shedding schemes either focus on evicting tuples at random [21], or at maximizing the size [22] of the result set. Moreover, none of the work considers *tuple-importance*, which is a very important requirement in data stream applications. We introduce tuple-importance using an example and motivate the need to exploit tuple-importance during join processing.

Consider two sources, for example CNN and BBC, generating news-feed information. Apart from other details, each tuple consists of: `HTML link`, `keyword`, `category` (such as "sports" or "politics") and the `time` when the news item was reported. It is easy to formulate a join query over CNN and BBC data streams that does the following: **"Find all news articles reported in last 2 hours, which have the same keyword"**.

Moreover, depending on the time-of-day the user might place more importance to news belonging to a certain category. For example, a user may prefer "politics" early in the morning, but enjoy reading about "sports" later in the evening. As evident from the example, it is rarely the case that a user perceives each tuple as being equally important. This is an example of a *value-based* importance, since the value of a tuple-attribute defines its importance. As a real-world example, iGoogle [23] (the personalized version of the Google search page) seems to be doing a similar query while reporting (and personalizing) news from various sources.

Other data-stream applications also require, or may benefit from tuple-level importance. For example, in sensor network the values sensed in a certain area, such as a disaster zone, might be more important than the ones sensed by sensors in other regions. This is an example of a *source-based* importance, since the source (or the sensor) that generated the tuple, defines its importance.

Above examples motivate us to look at schemes that exploit tuple-level importance during load-shedding. We believe that the assumption that all tuples have the same importance level is too simplistic and rarely applicable in practice. Our work focuses on developing a load-shedding scheme for tuples having *different* importance levels. Moreover, we accommodate periodic changes to tuple-importance. Our goal will be to maximize the *total-importance* of the join result by exploiting tuple-level importance.

1.1 Motivation and Problem

We consider equi-joins, but our work is equally applicable to other types of join. Let w be the size of a time-based window. An *exact* join [20] requires all tuples to be in memory. If sufficient memory is available at time t, all tuples with time stamps i such that $t-w < i \leq t$, are buffered and matched with joining streams.

As we have noted, sufficient memory may not be available to store all input tuples, due to burstiness or high arrival rates. In fact, unpredictable arrival characteristics may make it impossible to estimate the optimal memory requirement. We can only hope to provide the best approximation (least error) for the join. We will measure the join quality in terms of the importance of the joining tuples, and determine which tuples to evict or store that will maximize the join quality.

We present an example to show that load-shedding algorithms that do not address tuple-importance are not likely to meet our objectives. Thus, schemes such as first-in-first-out (FIFO), random-drop [20], and semantic approximation [22] are sub-optimal for two reasons. First, some important tuples suffer *premature evictions* and fail to match with other tuples. Second, some *unproductive* tuples stay in memory too long without contributing much to the join result.

Example:

Table 1 shows streams R and S. Each element is shown as a *value:importance* pair. Consider a sliding-window equi-join over R and S, with a window of 8 time units. Let output tuple o have importance $\min\{imp(r), imp(s)\}$, where $r \in R$ and $s \in S$ are the matching tuples that produce output tuple o. The total importance of the join result is the sum of the importance of the output tuples.

Time	1	2	3	4	5	6	7	8
R	a:1	b:2	c:3	d:4	d:4	b:2	a:1	c:3
S	b:2	a:1	b:2	b:2	c:3	c:3	d:4	a:1

Table 1. Arrivals

Consider the output in the time range $2 \leq t \leq 8$. An *exact* join (Table 2(a)) produces 16 output tuples with a total importance of 36. Let us now assume that only 2 tuples per stream can be stored in memory. Thus, after time $t = 3$, we must evict a tuple each time a new tuple arrives. Table 2(b) illustrates the scheme where tuples are dropped in first-in-first-out (FIFO) order. Table 3(a) illustrates a scheme where tuples are randomly dropped. This scheme produces only 3 output tuples, with a total importance of 5. This example clearly shows that choosing the right tuples to drop is difficult. Table 3(b) shows an approach intended to maximize the output size. Clearly, even this strategy does not significantly improve the approximation.

t	**R**	**S**	**Output**	**Imp**
2-3	a,**b**	b,**a**	b,a	3
3-4	a,b,c	b,a,**b**	b	2
4-5	a,b,c,d	b,a,b,**b**	b	2
5-6	a,b,c,d,d	b,a,b,b,**c**	c	3
6-7	a,b,c,d,d,**b**	b,a,b,b,c,**c**	b,b,b,c	9
7-8	a,b,c,d,d,b,**a**	b,a,b,b,c,c,**d**	a,d,d	5
8-9	a,b,c,d,d,b,a,**c**	b,a,b,b,c,c,d,**a**	c,c,a,a	8

(a) Exact (total importance = 36)

t	**R**	**S**	**Output**	**Imp**
2-3	a ,**b**	b,**a**	b,a	3
3-4	b,c	a,**b**	b	2
4-5	c,d	b,b	-	0
5-6	d,d	b,c	-	0
6-7	d,b	c,c	-	0
7-8	b,a	c,d	-	0
8-9	a,c	d,**a**	a	1

(b) FIFO (total imp. = 6)

Table 2. Exact join and approximate join under FIFO

t	**R tuples**	**S tuples**	**Out**	**Imp**
2-3	a,**b**	b,**a**	b,a	3
3-4	a,c	a,b	-	0
4-5	a,d	b,b	-	0
5-6	d,d	b,c	-	0
6-7	d,**b**	b,c	b	2
7-8	b,a	b,d	-	0
8-9	b,c	b,a	-	0

(a) RAND (total imp. = 5)

t	**R tuples**	**S tuples**	**Output**	**Imp**
2-3	a ,**b**	b,**a**	b,a	3
3-4	a,b	a,**b**	b	2
4-5	a,b	b,**b**	b	2
5-6	a,b	b,b	-	0
6-7	b,**b**	b,b	b,b	4
7-8	b,b	b,d	-	0
8-9	b,c	b,a	-	0

(b) SIZE (total imp. = 11)

Table 3. Approximate join processing using RAND and SIZE

Let $Str(v, t)$ denote a tuple with value v arriving in stream Str at time t. The total output importance suffers for two reasons. First, tuple $R(c, 3)$ is dropped prematurely, though it would have matched $S(c, 5)$ at $t = 6$. Second, $R(a, 1)$ produces no output tuple after $t = 2$, but occupies memory until $t = 5$, forcing the dropping of other tuples, which may have contributed more to the join. Without a priori knowledge of stream characteristics, it is difficult to determine if dropped tuples will result in future output tuples. We call $R(c, 3)$ a *premature* tuple and $R(a, 1)$ an *unproductive* tuple.

None of the previous schemes exploit tuple-level importance or correlations among tuple values. Our scheme both limits premature tuple drops and eliminates unproductive tuples. Our work deals with the question of *which* tuples to drop, to minimize the error during join approximation.

1.2 Approach

We present *iJoin*, a novel join-approximation technique, which tries to maximize join quality by dynamically adjusting tuple priorities, and performing load-shedding in a controlled manner. We assign each tuple a *residence priority RP*, using some novel techniques to estimate a tuple's worth.

When a tuple arrives, it is assigned an *initial* RP based on a user-specified function (say, the value of a certain attribute). The tuple's RP value is kept updated, based on its importance, the number of tuples it has matched, and its residence time in memory. Eviction is based on RP values.

We limit premature drops by requiring that tuples *mature* in memory for a threshold duration before becoming candidates for eviction. We also apply a penalty to *unproductive* tuples that have remained in memory for too long, without contributing towards output, and eventually evict them. We present three schemes for recomputing RPs: conservative, adaptive, and aggressive. Our approach also ensures that *fairness* is a used as metric in comparing join-approximation methods. Our work is 80%-85% fair in presence of limited memory, and gives the best join-approximation. We believe that our work is first to provide such a balance and yet very applicable in practice.

2 Related Work

Load-shedding [24] is generally used to evict tuples under memory size limitations, and has been studied in the context of sliding window versions of join queries [25, 20, 26, 22], aggregate queries [27], as well as in distributed environments [28]. Our work fits in the context of load-shedding and is aimed at *approximate* results.

Approximate join processing has been studied from different perspectives. When memory is full, [29–31, 20] propose randomly evicting a tuple from the join-memory. [22] argues that this scheme is likely to produce sub-optimal results, and proposes various heuristics to maximize the output *size*. Neither random-drop [20] nor semantic load-shedding [22] consider tuple-level importance, and hence perform poorly when applied for maximizing the *total importance* of the result. Load-shedding in Aurora [21] aims at reflecting the input tuple-distribution in the join output. However, none of the QoS-driven schemes in [21] consider user-specified importance in computing the approximate join.

Continuous queries are well-studied, but most work [32–34] relates to minimizing the memory utilization or improving the query latency. Eddy [35] provides a very general framework that performs a per-tuple optimization, but does not address load-shedding, or approximate joins when tuples bear user-specified importance. In contrast, our work does not focus on minimizing query latency, but on memory limitations during query execution.

Stream summarization is another alternative to addressing the case of limited memory. These schemes are orthogonal to our work and have been studied for approximating results when all tuples do not fit in memory. XJoin [29] and MJoin [30] produce exact results. In XJoin, some tuples are flushed to the disk when the data stream overwhelm the main memory. The flushed tuples (or backlog of tuples) are processed at a later time, when the arrival rates drop, and CPU is available to process this backlog. MJoin improves this scheme by decomposing multiple join operators. Our work deals only with the FastCPU [22] case, where all incoming tuples can processed. The problem is only when all the tuples cannot be buffered for processing to complete. Moreover, we are not interested in allowing evicted tuples to re-enter the system. Our work focuses on developing a on-line scheme that can dynamically adapt to the stream arrival characteristics.

3 Problem Formulation

We distinguish between *join attributes* and *importance attributes*. A join-attribute *jattr* is an attribute over which the join condition is specified. An importance-attribute *iattr* is an attribute whose values define the *importance* of the tuples. In the example presented in Section 1, the *jattr*=`keyword` and *iattr*=`category`. Join- and importance-attributes may overlap, but we use the distinction to highlight the purpose of value-based importance specification we described in Section 1.1. Our work is also applicable to source-based importance, but henceforth in the discussion we only consider value-based importance.

3.1 Tuple Importance

Various *importance functions* can be defined over the importance attributes. Since joins are ultimately at the user's (or application's) behest, we argue that it is best to allow the user (or application) to define tuple-importance semantics. For ordered attributes, for example, the user may provide a histogram of importance values. For categorical attributes, the user may specify a lookup table of categories, and their respective importance values.

Formally, importance-function F_i returns the importance of the tuple as a function of the importance-attributes of the tuple. As Equation 1 shows, a tuple $r(t)$ has importance $r(t).imp$, where F_i is the importance-function stated above and $a_1, a_2, \ldots$ are the importance-attribute.

$$\mathtt{r(t).imp} = \mathtt{F_i(r(t).a_1, r(t).a_2, \ldots)} \tag{1}$$

3.2 Join Processing Model

We consider sliding-window joins. Let the window size be w. At time t, the stream window consists of tuples $r(i) \in R$ and $s(i) \in S$ with time-stamps such that $t-w+1 < i \leq t$. When a new tuple $r(j)$ arrives at time i, it is matched with all tuples $s(i)$ in the window with $i \leq j$. If possible, $r(i)$ is buffered. Similarly, when new S tuples arrives, they are joined with the tuples in R's window.

After the join is executed over a particular window, we slide the time-window by Δt time units. Tuples that fall outside the new window are said to be *expired*, and are hence flushed out of memory. This join processing model is popularly known as a sliding-window join [20] and used in most recent literature.

3.3 Join Output Quality

When a tuple $r(j)$ joins with a tuple $s(i)$, the output tuple $o(j)$ acquires importance based on $r(j).imp$ and $s(i).imp$. Various assignments are possible, but for simplicity we define importance of an output tuple as shown in Equation 2. The output tuple also bears user-specified importance, as it is derived from importance of input-tuples, whose importance is defined by as in Equation 1.

$$\texttt{o(j).imp} = \min\{\texttt{r(j).imp}, \texttt{s(i).imp}\} \tag{2}$$

Let $\Omega_q = \{o(i_1), \ldots, o(i_n)\}$ be the output tuples of a join query q. We define join output quality as in Equation 3. This is called the **total importance** IMP(q) of outputs produced by query q.

$$\texttt{IMP(q)} = \sum_{\texttt{o(i)} \in \Omega} o(i).imp \tag{3}$$

3.4 The Case of Limited Memory

Let M be the total amount of buffer memory available, and for simplicity, let it be equally divided between R and S. If each tuple occupies one unit of memory, a maximum of $M/2$ tuples from each stream can be buffered. Let the tuple $r(t)$ arrive when R's buffer is full. If any of the $M/2$ buffered R tuples were to expire due to shifting of the sliding window, then $r(t)$ can occupy the vacated memory. Otherwise, the DSMS has to make either of the following two decisions.

- The DSMS can drop $r(t)$ and process the tuples already in the join buffer. This method is unfair to $r(t)$, since it is summarily and prematurely dropped.
- The DSMS can evict a buffered tuple, clearing space for $r(t)$. This requires some load-shedding policy.

Load-shedding is challenging in stream environments, because there is no foreknowledge of arrival characteristics. Any estimation is likely to be sub-optimal if the data streams are erratic. In our work, we present a load-shedding scheme to maximize the quality of an approximate join.

3.5 Dealing with Fairness

Fairness is a measure of how 'fair' the algorithm performed with respect to retaining (or evicting) tuples from join memory. When memory is unlimited, none of the tuples face eviction and hence fairness is irrelevant. However, when memory is a constraint, tuple eviction needs to be guided by fairness due to several reasons. For example, premature tuple eviction is unfair as these tuples do not get enough time to prove their 'worth'. Likewise, fairness is also affected when unproductive tuples enjoy long time in residence.

Addressing tuple-importance and fairness simultaneously is challenging. Generally, since tuple characteristics are not know apriori, we cannot accurately predict tuple correlations which help estimate the tuple 'worth'. The best we can do is to give all tuples near equal time to reside in memory. None of the previous work addresses fairness during load-shedding, so our contribution is novel. We aim at developing a load-shedding policy that not only addresses fairness, but also provides a means to trade-off approximation quality with the level of fairness.

As shown in Equation 4, we will use Jain's fairness index [36] as a measure of fairness, where tuple lifetime L_i is the difference between time that tuple i gets evicted, and it's arrival time. Fairness ranges from $1/n$ (worst case) to 1 (best case), where n is the total number of incoming tuples. As we will see through our experiments, a relatively fair heuristic is likely to have better join quality, than an approach that retains a baised set of tuples.

$$\texttt{fairness} = \frac{(\sum L_i)^2}{(n \times \sum L_i^2)} \tag{4}$$

3.6 Problem Statement

Given the available memory M and a sliding-window join query $\langle \alpha, c, w \rangle$, where $\alpha = \{S_1, \ldots, S_n\}$ is the set of streams (with importance semantics), c is the join condition, and w is the time-window, compute the approximate join such that **total importance** of the join output is maximized.

4 Approach

In this section, we present our approach to perform approximate join processing. As a prelude to iJoin, we study a simple static join approximation scheme that tries to exploit tuple importance. If the memory is full when a new tuple arrives, the tuple with the lowest importance is dropped. Tuples with the highest importance are *greedily* retained. Table 4 shows how this greedy scheme would perform for the scenario in Section 1.1.

Time	R	S	O	Imp
2-3	a,**b**	b,**a**	b,a	3
3-4	b,c	b,**b**	b	2
4-5	c,d	b,b	-	0
5-6	d,d	b,c	-	0
6-7	d,d	c,c	-	0
7-8	d,d	c,**d**	d,d	8
8-9	d,d	c,d	-	0

Table 4. Approx Join using GREEDY (Total imp=13)

As with any static scheme, this scheme causes premature drops when a tuple arrives with importance lower than those of currently buffered tuples. For example, $R(b, 6)$ is dropped on arrival at $t = 6$. In contrast, some high-importance tuples remain in memory for disproportionately long duration, but make no (or

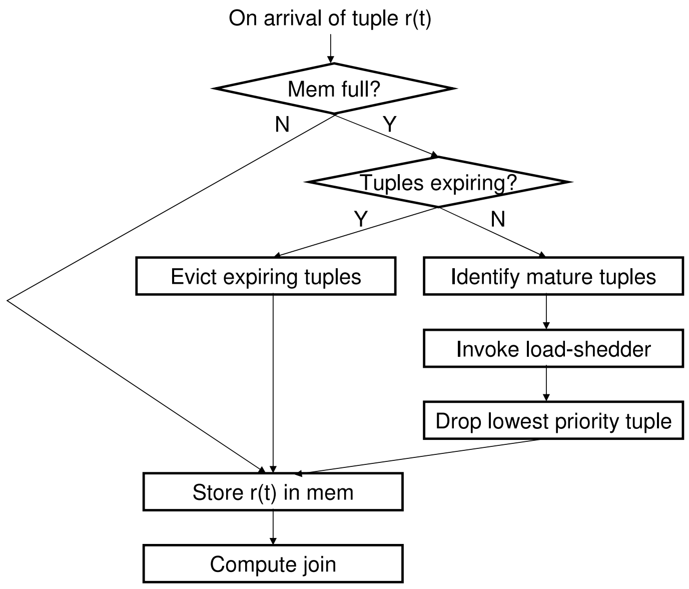


Fig. 1. Flow diagram for iJoin approximation

little) contribution to join results. For example, tuples with value d (importance=4) occupy R's buffer for all duration after $t > 5$. Next we present our dynamic scheme, called iJoin, that addresses the above drawbacks.

4.1 iJoin Overview

Our framework is outlined in Figure 1. When a tuple $r(i) \in R$ arrives at time i, it is buffered if there is room, and joined with tuples in the window for stream S. As shown in Algorithm 1, various tuple-related information (or metadata) is also updated during join operation. We discuss how this metadata is used in determining tuple priorities in Section 4.3. If the buffer is full, we drop some tuples based on the load-shedding strategy discussed in Section 4.7. Once the unwanted tuples are dropped, we accommodate $r(i)$ and proceed with the join as in Algorithm 1. Since dropped tuples never return, we must use existing statistics to determine which tuples to hold on to.

4.2 Tuple Metadata

We maintain some metadata with each tuple to help estimate the join statistics. We use these statistics to determine a tuples 'worth' and hence which tuple to drop from set of candidates tuples already in join memory.

The following metadata is stored with each tuple.

- t_a**:** The arrival time of the tuple, which is set on tuple arrival.

Algorithm 1 Join operation

```
Require: window size $w$, join condition $c$, $r(i), \gamma = \{s(j)\}$ such that $i - w \leq j \leq i, w$
Ensure: the output set $\Omega$
 1: for all $s(j) \in \gamma$ do
 2:   if $isMatch(s(j), r(i), c) = TRUE$ then
 3:     $o(i) = \{s(j), r(i)\}$
 4:     $o(i).imp = \min\{s(j).imp, r(i).imp\}$
 5:     $\Omega \leftarrow \Omega \cup \{o(i)\}$
 6:     $s(j).matches \leftarrow s(j).matches + 1$
 7:     $s(j).prevmatch \leftarrow i$
 8:   end if
 9: end for
10: if $\Omega \neq \emptyset$ then
11:   $r(i).matches \leftarrow |\Omega|$
12:   $r(i).prevmatch \leftarrow i$
13: end if
```

– **imp:** This is the tuple importance determined according to Equation 1. It is updated only if the user redefines the importance function.
– **matches:** The number of tuples that have joined with the tuple, so far. This value is updated each time the tuple matches another tuple.
– **prevmatch:** The timestamp of most recent matching tuple. This is updated every time a tuple matches (or joins) with a tuple.

Algorithm 1 shows how the metadata is updated when $r(i)$ finds a match to produce the output tuple o(i). The `isMatch()` function in Algorithm 1 returns either TRUE or FALSE depending on the join condition $c \in \{<, \leq, =, \neq, >, \geq\}$.

4.3 Tuple Priority

A tuple's *residence priority (RP)* indicates how valuable the tuple is to the join process, relative to other tuples in memory. As we will see later, tuples with the lowest `RP` get dropped from memory. On arrival, each tuple $r(i)$ is assigned a default `RP`, called $\mathtt{RP_{init}(i)}$. This value can either be specified by the user, or be system parameter configured on a per-tuple basis. We must construct a function F_p to determine the tuple priority at some time $t' > t_a$. Since we want to maximize the total importance of a query output, F_p (see Equation 5) is a function of the tuple importance, the number of matches, as well as the age of the tuple in memory. Residence priority at time t' is computed using Equation 6.

$$\mathtt{F_p}(\mathtt{imp}, \mathtt{matches}, \mathtt{age}) = \frac{\mathtt{imp} \times \mathtt{matches}}{\mathtt{age}} \qquad (5)$$

$$\mathtt{P}(\mathtt{r(i)}, \mathtt{t'}) = \mathtt{F_p}(\mathtt{r(i).imp}, \mathtt{r(i).matches}, (\mathtt{t'} - \mathtt{r(i).t_a})) \qquad (6)$$

4.4 Local Priority

The `matches` value used in Equation 5 is a *global* indication of how many tuples matched it during its lifetime. We have found that this can bias the estimation of tuple's worth. The *local* priority of a tuple, defined below, works better. We estimate the number of matches using Equation 7, where d is a constant used in an exponential decay function [37], $\mathtt{match}_{[\mathtt{t_1},\mathtt{t_2}]}$ is the number of tuples that matched in the time range $[t_1, t_2]$, and t' is the current time. We use this in Equation 5 to obtain the local priority of the tuple.

$$\mathtt{matches}' = \sum_{\mathtt{t}=\mathtt{t_a}}^{\mathtt{t}'} \{\mathtt{match}_{[\mathtt{t},\mathtt{t}-1]} \times \mathtt{e}^{-\mathtt{d}(\mathtt{t}'-\mathtt{t})}\} \qquad (7)$$

4.5 Establishing Tuple Maturity

To reduce premature drops, we place a residence threshold on the tuples before they are candidates for load shedding. We consider a tuple to be **mature** if it's age is greater than a certain threshold τ, which is a tunable system parameter, whose value can be set higher (or lower), in accordance with greater (or lower) memory availability. We will only drop mature tuples, so residence times are longer. We will see the benefits of this policy in the Section 5

4.6 Penalizing Unproductive Tuples

A tuple is *unproductive* if it does not produce a join output for a long time. Our goal is to identify such tuples, and penalize it for occupying memory, without producing output tuples. When such unproductive tuples are penalized, they will quickly lose their residence priority and will be eventually evicted.

We first record the timestamp of the most recent match. For example, if a output tuple was produced at time $\mathtt{t}$, which means when a arriving tuple $\mathtt{s}(\mathtt{t})$ matched with $\mathtt{r}(\mathtt{i})$, we record this time by assigning $\mathtt{r}(\mathtt{i}).\mathtt{prevmatch} = \mathtt{t}$ and $\mathtt{s}(\mathtt{t}).\mathtt{prevmatch} = \mathtt{t}$. When we recompute the tuple priorities, as in Algorithm 2, we do the following. First, we identify which tuples are unproductive. A tuple is considered unproductive at time $\mathtt{t}'$ if $\mathtt{t}' - \mathtt{r}(\mathtt{i}).\mathtt{prevmatch} \geq \Delta$, where Δ is a tunable parameter. If a tuple is identified as being unproductive, we reduce its priority by a penalty δ computed using Equation 8, where c is some constant.

$$\mathtt{Penalty}(\delta) = \mathtt{c} \times (\mathtt{t}' - \mathtt{r}(\mathtt{i}).\mathtt{prevmatch}) \qquad (8)$$

A tuple suffers a penalty in proportion to the time it is unproductive. An unproductive tuple can redeem itself if there is a match before it is evicted.

4.7 Load-shedding Scheme for iJoin

We perform load shedding based on residence priority, as shown in Algorithm 2, using the tuple's `age, matches, imp`, and `prevmatch` information. We first

Algorithm 2 Load-shedding invoked at time (t)

```
Require: $\beta$={r(i)} such that t-w $\leq$ i $\leq$ t, Maturity threshold $\tau$, Unproductivity
    threshold $\mathit{\Delta}$, Penalty $\delta$, $k$ .
 1: for all r(i) $\in \beta$ do
 2:    Apply Condition Maturity ($\tau$) to determine if r(i) is MATURE
 3:    if r(i) is NOT MATURE then
 4:       $\beta \leftarrow \beta$ - {r(i)}
 5:    end if
 6: end for
 7: for all r(i) $\in \beta$ do
 8:    Determine the tuple-priority P(r(i),t)
 9:    Apply Condition Unproductivity ($\mathit{\Delta}$) to determine if r(i) is UNPRODUCTIVE
10:    if r(i) is UNPRODUCTIVE then
11:       P(r(i), t) $\leftarrow$ P(r(i),t) - $\delta$
12:    end if
13: end for
14: for all r(i) $\in \beta$ do
15:    Sort tuple by tuple priority P(r(i), t)
16: end for
17: Drop r(j) such that P(r(j),t) = min (P(r,t)) $\forall$ r $\in \beta$
```

identify mature tuples as eviction candidates. Next, we examine the `prevmatch` timestamp to check for unproductive tuples, apply a penalty, and compute all new residence priorities. We then sort the tuples in descending order of their priority, and drop the tuples with the lowest tuple priority. Well-known data structures such as priority queues can be used here for implementation.

4.8 Recomputing Tuple Priorities

Recomputing residence priorities can be expensive. In Algorithm 2, for example, we recalculate the priorities each time load shedding is triggered. We propose three schemes to control how often we recompute residence priorities, which represent different tradeoffs between maintaining accurate priority estimates and join approximation error.
(1) Successive: In this scheme, we compute tuple priority each time we need to evict a tuple from memory. This scheme has high overhead, but provides the best estimates of the ‘worth’ of tuples currently in memory.
(2) k-Successive: In this scheme, we compute tuple priorities after every k load shedding decisions, reducing overhead by a factor of k, over the successive scheme. If tuples have relatively stable distributions, k could be set to a higher value. If the stream is expected to be erratic, it is better to use a lower k.
(3) Adaptive: In this scheme, we monitor the *relative difference* between consecutive join-approximation executions. We recompute residence priorities only if the difference in join quality falls below a certain threshold ϵ. The choice of ϵ depends on how much overhead is acceptable. Lowering ϵ increases overhead.

Parameter	**Value**
arrival rate	100–200 tuples per sec
tuple domain	1–100 categorical values
imp domain	1–100 decimal values
join memory M	10 tuples
window size w	25 secs
maturity threshold τ	2 secs
unproductivity threshold Δ	3 secs
recomputation policy	successive

Table 5. Default parameter settings

5 Experiments

We studied the performance of our scheme using various experiments. Since no real data sets were available with importance specified, we used a synthetic dataset closely resembling real-world settings. The tuples had categorical attributes, and arrived at the rates between 100–200 tuples per second. Unless specified, we used the default values shown in Table 5, where domain size is the number of distinct values in the join attribute and the importance attribute. The importance of each tuple was mapped to the range 1–100. We allowed the importance function to be redefined no more than twice during execution.

Other parameters were kept configurable, to study their influences on iJoin performance. We were specifically interested in how the available-memory M, window size w and the data distribution affected the query output. Unless specified, we set the maturity threshold $\tau = 2$ secs, and the unproductivity threshold $\Delta = 5$ secs. For obvious reasons, we always maintained $\tau \geq \Delta \geq M \geq w$.

We used an equi-join query over 2 streams as a test query. We allowed the query to run for 100 seconds on a Quad Xeon 550MHz. Note that our work addresses the FastCPU [22] case, and hence the simulations are consistent across various other processors as well. Unless specified, we used the successive scheme for recomputing the residence priority. We measured the following:

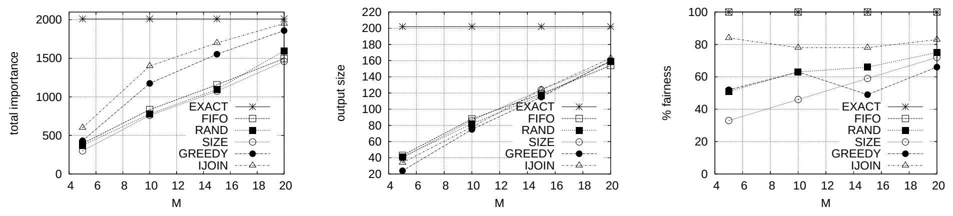
- **total importance:** This is join-output quality measure (Equation 3).
- **output size:** The number of output tuples generated by the query.
- **fairness:** A measure of fairness as stated in 4.

We compared our work with the following load-shedding schemes.

- **EXACT:** The optimal scheme where memory is unlimited.
- **FIFO:** A FIFO scheme, with queue size M.
- **RAND:** Random load-shedding scheme used in [20].
- **SIZE:** A scheme similar to [22], that maximizes output size.
- **GREEDY:** The greedy version of IJOIN we described in Section 4.

5.1 Effect of Memory Size

We varied the available join memory from 5 to 20 tuples. Figure 2 shows that IJOIN has the best performance in meeting our approximation objective of maximizing total importance. EXACT, of course out-performs all schemes due to availability of unlimited memory. Though the SIZE scheme is designed to maximize the output-set, we noticed that our heuristics did better, and also showed higher join quality. This might be due to the SIZE's limited ability to detect correlation among joining streams in our dataset. IJOIN also does the best with respect to fairness. Our scheme is developed to provide equal, or almost equal opportunity to each tuple to find a matching tuple, and our experiments show that IJOIN was more than 80% fair in doing so. FIFO obviously is 100% fair, but has the lowest join-output quality.



(a) Join output quality (b) Number of outputs (c) Fairness Index

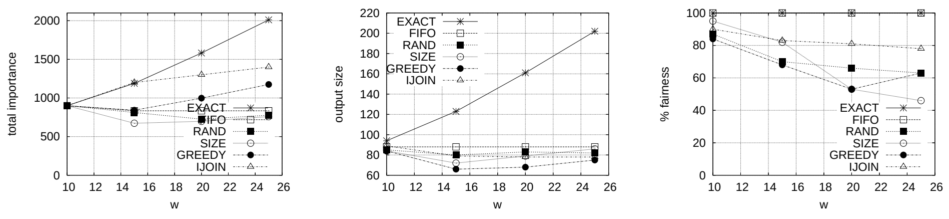
Fig. 2. Performance when available memory is varied from 5 to 20

5.2 Effect of Window Size

Figure 3 shows the performance of various schemes when the window size varied from 10 to 25 seconds. Intuitively, a larger window places higher memory constraints on the join operation, as the available join memory was constant in this experiment. Only the IJOIN and GREEDY schemes seem to improve join quality when the window size grows. This is due to IJOIN's ability to find better correlations, and use recent estimates to determine the more valuable tuples. A larger window-size provides IJOIN a larger set for tuples to find correlations, and hence, the residence priorities. For this dataset, the output size hardly seems to increase for any scheme. IJOIN is between 80%-85% fair in retaining tuples in memory. In contrast, SIZE experiences a drop from 90% to 42%.

5.3 Effect of Domain Size of Join Attribute

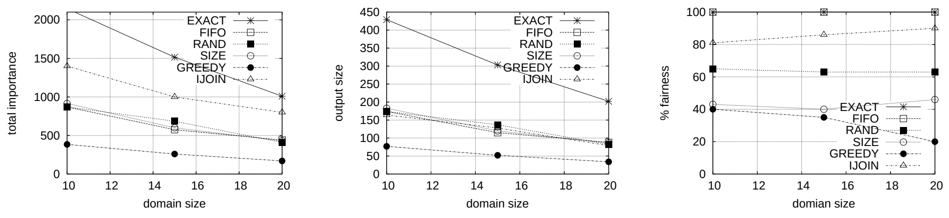
In this set of experiments, we varied the domain from 10–20 distinct values. Since we wanted to see the effect of domain-size on the output quality, we let all tuples



(a) Join output quality (b) Number of outputs (c) Fairness Index

Fig. 3. Performance when query window is varied from 10 to 25

have the same importance. We noticed that domain size has large impact on our quality measure. Intuitively, when the domain size increases, the probability of the same value appearing in the window decreases. As shown in Figure 4, this affects the number of matching tuples, so we see a linear drop in the output size and total importance. IJOIN still outperforms all other approximation schemes, and is consistently fair between 80%-85%. Interestingly, GREEDY has the worst performance in this setting, because all tuples have the same importance-level. Moreover, this leads to only a selected few tuples occupying the memory. For example, we say that GREEDY is only 20% fair when domain size is 20.

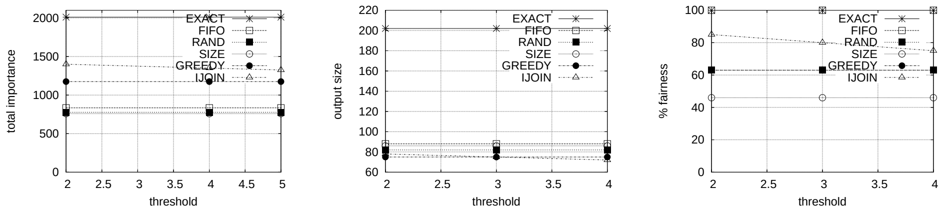


(a) Join output quality (b) Number of outputs (c) Fairness Index

Fig. 4. Performance when domain-size is varied from 10 to 20

5.4 Effect of Thresholds

We studied how join approximation is affected by the unproductivity threshold Δ. Since this parameter is specific to IJOIN, other schemes show no change when the threshold is increased from 2 to 5 seconds. As shown in Figure 5 when the threshold is more coarse (higher value) IJOIN quality drops. This is because when the threshold is coarse, a tuple a can remain longer without contributing to any result. We recommend that the scheme use a finer threshold (low value) or apply higher penalty δ (see Algorithm 2) when tuples are identified as unproductive. Though we used threshold ranging between 2-5, IJOIN had better performance than all other schemes.



(a) Join output quality (b) Number of outputs (c) Fairness Index

Fig. 5. Performance when unproductivity threshold is increased from 2 to 5 seconds

5.5 Performance of Re-computation Schemes

In this experiment we study how we can trade-off computational overhead with approximation quality. As shown in Table 6, the successive provides the best approximation. The successive scheme is also the most 'fair' heuristics at about 80%. It is worth noting however that the adaptive scheme has a lower overhead than successive, with only 600 instances when tuple priorities needed to be re-computed. The k-successive scheme is a balance between the two schemes, with a low overhead and is about 75% fair. We propose that that successive scheme be used in erratic environments, where data distribution and tuple correlations are likely to be volatile. Adaptive scheme performs best when data distributions are relatively stable.

	Successive	**k-Successive**	**Adaptive**
Total importance	1400	1200	1300
Output size (#tuples)	80	75	80
Fairness	81%	75%	80%
Overhead (#invocations)	high (1000)	low (100)	medium (600)

Table 6. Performance of re-computation schemes

6 Conclusions

We have presented a novel join-approximation framework, called *iJoin*, that addresses tuple-importance. Our load-shedding policy in iJoin is based on various tuple properties, such as recent value-correlations, time spent in memory, and tuple-importance. We determine which tuples are too early to drop, as well as tuples that have spent too much unproductive time in memory. Our techniques show that limiting premature drops, and penalizing unproductive tuples is an effective strategy in maximizing quality of join-result. Compared to previous work, our approach not only suffers least approximation error, but also is fair with respect to retaining tuples in memory. We believe that addressing fairness is important goal when memory is limited, because there is no foreknowledge of the 'worth' of future incoming tuples. We have also outlined 3 schemes that help recompute tuple-priorities. The successive scheme is the best to use if data is erratic, or tuple-priority are likely to fluctuate frequently. The adaptive scheme has lower overhead, and provides the best trade-off between join quality and the overhead of load-shedding.

References

1. Yahoo news, http://news.yahoo.com/rss, 2007.
2. Nyse euronext, http://www.nyse.com/, 2007.
3. Etrade financial, http://etrade.com, 2007.
4. Scottrade homepage, http://scottrade.com, 2007.
5. Tao project, http://www.pmel.noaa.gov/tao/, 2007.
6. Pacific tsunami warning center, http://www.prh.noaa.gov/pr/ptwc/, 2007.
7. eBay. ebay homepage, http://pages.ebay.com, 2007.
8. Sirish Chandrasekaran and et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of CIDR*, 2003.
9. Morton M. Astrahan and et al. System r: Relational approach to database management. *TODS*, 1(2):97–137, 1976.
10. Don Carney and et al. Monitoring streams - A new class of data management applications. Technical Report CS-02-04, Department of Computer Science, Brown University, February 2002.
11. Rajeev Motwani and et al. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of CIDR*, 2003.
12. Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proceedings of ACM SIGMOD*, pages 379–390, 2000.
13. Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of USENIX Annual Technical Conference*, pages 13–24, 1998.
14. Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of ICDE*, 2002.
15. Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of ACM SIGMOD*, pages 430–441, 1994.
16. StreamBase Stream Processing Engine. Streambase systems inc. homepage, http://www.streambase.com/, 2007.

17. Coral8, inc. homepage, http://www.coral8.com, 2007.
18. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of Principles of database systems*, 2002.
19. Surajit Chaudhuri. An overview of query optimization in relational systems. In *Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.
20. J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of ICDE*, 2003.
21. Nesime Tatbul. Qos-driven load shedding on data streams. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, 2002.
22. Abhinandan Das. Semantic approximation of data stream joins. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):44–59, 2005. Member-Johannes Gehrke and Member-Mirek Riedewald.
23. igoogle webpage, http://www.google.com/ig, 2007.
24. N. Tatbul, U. Cetintemel, S. Zdonik, M. Chemiack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of VLDB*, 2003.
25. B. Gedik, K. Wu, P. Yu, and L. Liu. Adaptive load shedding for windowed stream joins, 2005.
26. U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins, 2004.
27. N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *International Conference on Very Large Data Bases (VLDB'06)*, Seoul, Korea, September 2006.
28. N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *International Conference on Very Large Data Bases (VLDB'07)*, Vienna, Austria, September 2007.
29. Tolga Urhan and Michael J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
30. Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-join queries over streaming information sources. In *Proceedings of VLDB*, 2003.
31. Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of SIGMOD*, 2002.
32. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB Journal*, 13(4):333–353, 2004.
33. J. Widom and R. Motwani. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of CIDR*, 2003.
34. Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *vldb'2003: Proceedings of the 29th international conference on Very large data bases*, pages 838–849. VLDB Endowment, 2003.
35. Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD*, volume 29, pages 261–272. ACM, 2000.
36. R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. DEC Research Report TR-301, Digital Equipment Corporation, Maynard, MA, USA, September 1984.
37. Mathworld, http://mathworld.wolfram.com, 2007.