GreedyDual-Join
Locality-Aware Buffer Management for Approximate Join Processing Over Data Streams

Feifei Li, Ching Chang, Azer Bestavros, and George Kollios
Computer Science Department,
Boston University
Boston, MA, USA
{lifeifei, jching, best, gkollios}@cs.bu.edu

Technical Report BUCS-TR-2004-028

Abstract

We investigate adaptive buffer management techniques for approximate evaluation of sliding window joins over multiple data streams. In many applications, data stream processing systems have limited memory or have to deal with very high speed data streams. In both cases, computing the exact results of joins between these streams may not be feasible, mainly because the buffers used to compute the joins contain much smaller number of tuples than the tuples contained in the sliding windows. Therefore, a stream buffer management policy is needed in that case. We show that the buffer replacement policy is an important determinant of the quality of the produced results. To that end, we propose GreedyDual-Join (GDJ) an adaptive and locality-aware buffering technique for managing these buffers. GDJ exploits the temporal correlations (at both long and short time scales), which we found to be prevalent in many real data streams. We note that our algorithm is readily applicable to multiple data streams and multiple joins and requires almost no additional system resources. We report results of an experimental study using both synthetic and real-world data sets. Our results demonstrate the superiority and flexibility of our approach when contrasted to other recently proposed techniques.

1 Introduction

Stream database systems have attracted quite a bit of interest recently due to the mushrooming number of applications that require online data management on very fast changing data. Example applications that motivate the need for stream database systems include network monitoring, sensor networks, financial applications, *etc.* [4]. The main difference between a traditional data base management system and a system that manages data streams (DSMS) is the assumption of the former that each relation is stored on disk and that each relation has a finite size. In a DSMS, instead of relations we have unbounded data streams and relational operations are applied over these streams. However, since the data streams are potentially unbounded, the storage that is required to evaluate complex relational operations, such as joins, is also unbounded [2]. There are two approaches to address the above issue. One is to allow for approximations that can guarantee high-quality results. The other, is to define new versions of the relational operations based on sliding windows. In that case, the operation is applied only to the most recent window of the data stream. Note that imposing sliding windows on data streams is a natural method to get good quality approximate results and in many applications, (*e.g.*, monitoring), it makes more sense to consider only the most recent data [4].

One very important operation in a DSMS is *the sliding window join*, which can be defined as follows: given two streams R and S and a sliding window W, a new tuple r in R that arrives at time t, can be joined with any tuple s in S that arrived between t-W and t. Furthermore, r can be joined with any tuple in S that will come between t and t+W. Beyond that, *e.g.*, at time t+W, r expires and can be safely removed from memory since it cannot produce any more results. The window can be defined either as time based, tuple based, or landmark based, and each stream may have a different sliding window. If the system can store and process the entire sets of the two windows in main memory, then the exact answer to the above operation can be produced. However, there are many cases where the full contents of the sliding windows cannot be stored in memory. One reason can be that the *memory size* is smaller than the size of the window and only a subset of the window elements can be stored in main memory. Another reason could be that the arrival rate of stream tuples is very high to the point that it exceeds the *processing capacity* of the system—i.e., the ability of the system to process all tuples in the window in a timely manner. In either of these cases, it is desirable that the use of the available buffer space be optimized so as to produce the best possible approximate result of the sliding window join operation [11].

Clearly, what constitutes a "good approximation" depends on the application at hand. Examples may include (1) producing the largest subset of the exact answer, or (2) producing an unbiased sample of the join results. Existing work focused mainly on the first problem [11, 22, 35], with the exception of the work in [32] which addressed both problems. In this paper, while we concern ourselves mostly with the first of these approximation goals, we also discuss and evaluate the suitability of our buffer management technique for the second goal as well.

1.1 Problem Statement

We consider a system that manages a set of n continuous data streams $S = \{S_1, S_2, \dots, S_n\}$. A continuous data stream is a potentially unbounded sequence of tuples $\{t_1, t_2, t_3, \dots\}$. Each tuple may have multiple attributes $\{a_1, a_2, \dots, a_m\}$ as well as a timestamp i indicating the time that such a tuple arrived into the system. Each attribute a_i takes values from a discrete domain D_i. We denote a tuple from stream S_j that arrives at time i, as t_i^j. Consider a workload $Q = \{Q_1, Q_2, \dots, Q_m\}$ of ad-hoc continuous queries that are running in such a system. For simplicity, in this paper, we only consider equi-joins, noting that our techniques can be easily extended to general *theta* joins. We assume that each Q_f is a sliding window equi-join query on two streams S_l and S_k over a common attribute a_p. Tuple t_l^i is joined with tuple t_k^j, if $t_l^i[a_p] = t_k^j[a_p]$ and $|i - j| < W_f$. The value W_f represents the sliding window specified for this join.

As we hinted before, we assume that the total memory available is smaller than the total memory needed to compute the exact result of the query workload. Specifically, if the size of the *i-th* stream window is W_i and the memory that we have in our disposal is M, then $M < \sum_{i=1}^{n} W_i$. Thus, it is impossible to guarantee exact results, and we must rely on approximate solutions instead. To do so, we adopt the *MAX-subset* criterion proposed in [11], which aims to maximize the number of tuples produced by the system. Furthermore, we discuss the problem of producing a representative (unbiased) random sample, and we show how our technique can be tuned to cater to that goal as well.

We consider the buffer management problem for the architecture shown in Figure 1. This problem can be decomposed into two sub-problems. The first is the memory allocation problem: given a set of streams S and a memory size M, we have to partition the memory into $|S|$ buffers, one for each stream. The second is the buffer replacement policy: upon the arrival of a new tuple to a full buffer, we have to choose one of the tuples in the buffer to evict.

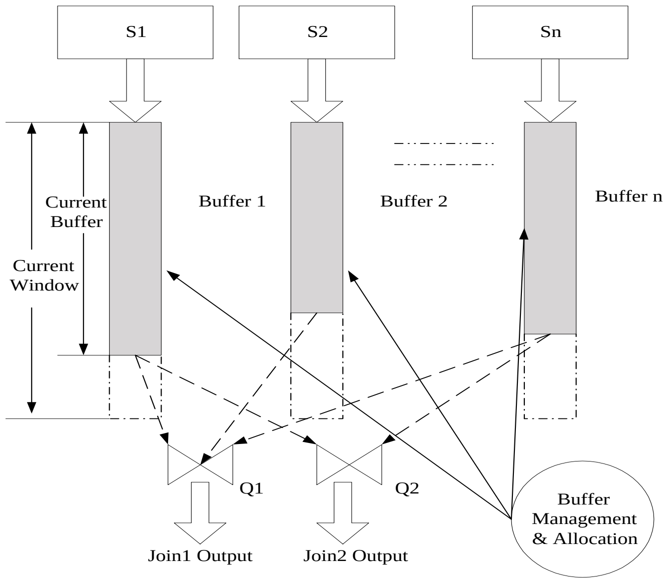


Figure 1: Stream Join Processing Architecture

1.2 Contributions

We cast the problem of buffer management for approximate join processing as a generalized cache replacement problem. Unlike traditional cache replacement techniques, the performance of the replacement strategy for a given cache (*e.g.*, the buffer of stream R) depends on the content of one or more *other* caches (*e.g.*, the buffer for stream S), which in turns depends on the contents of that cache. This interdependence suggests that a successful replacement technique (for all caches) must ensure a “convergent” behavior. In this paper, we argue that previously suggested techniques do not necessarily promote (or capitalize on) such convergence; we present an alternative buffer management technique that does. Our approach—which could be thought of as a generalization of the well-known Greedy-Dual (GD) algorithm [36]—enables the reference stream of one cache to affect the cache-ability of entries in other caches, thus engendering said convergent behavior. Among its many features, our approach capitalizes effectively on temporal locality properties in the streams, allows an integrated buffer management for all streams, and is readily amenable to approximate join processing involving multiple joins on multiple attributes in multiple streams. In support of our claims, we present extensive simulation results using synthetic as well as real traces.

2 Reference Locality in Data Streams

Locality of reference properties are important determinants of the performance of caching mechanisms. Denning and Schwartz [12] established the fundamental properties that characterize temporal locality in memory access patterns. The presence of temporal locality has been documented for many other types of data streams, including web reference streams [1, 3, 7].

2.1 Sources and Characteristics

There are two significantly different sources of temporal locality in data streams in general: (1) popularity over long time scales, often captured by Zipf’s law, and (2) popularity over shorter time scales, often captured by various correlation coefficients. Among others, two methods are well adopted for the quantification of temporal locality, namely the stack distance model and the inter-request time distribution model.

In [28], Mattson et al introduced the concept of stack distances as a means for analyzing the behavior of demand-paged memory systems and for evaluating the performance of memory management schemes.

Definition 1 *For a given data stream,* ***stack distance*** *refers to the nubmer of unique references separating consecutive requests to the same object.*

In [1], Almeida *et al* used the marginal distribution of stack distance strings to characterize temporal locality. While the stack distance model provides means for characterizing the degree of temporal locality that exists in a request stream, it is not able to delineate the causes of such locality—namely whether it is due to popularity over long or short time scales [27].

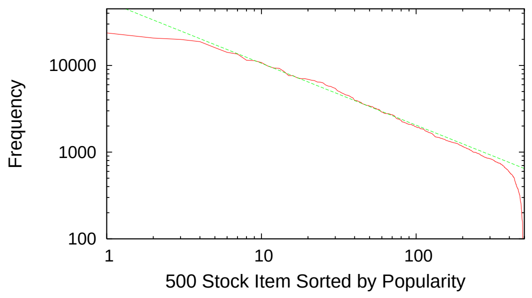
The second method in characterizing temporal locality is the use of the distribution of inter-request times. In [19], Jin and Bestavros showed that this distribution is predominantly determined by the large skew (*e.g.*, power law) governing the long-term popularity of objects [3, 7]. This inherent relationship tends to disguise the existence of short-term temporal correlations (*i.e.*, popularity over shorter time scales).

One way of quantifying the causes of reference locality is to compare the distribution of stack distances or inter-request times for for a reference stream with that of a randomly permuted version of that same trace. The random permutation of the original trace will destroy any short-term temporal correlations, but will not affect the popularity profile of the constituent references in the trace: A significant change in either distributions would indicate the prevelance of short-term correlations.

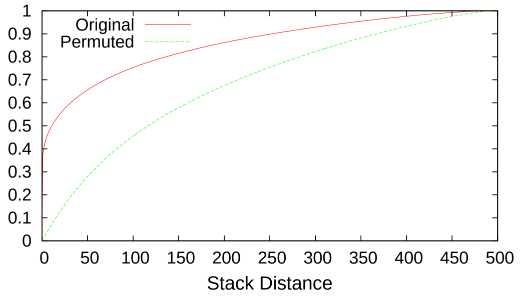
In [19], Jin and Bestavros used this methodology to quantify (using newly proposed metrics) the relative strengths of both sources of temporal locality in a variety of web traces. Their results revealed measurable differences in strength for these two sources of locality, which they attributed to the location (within the network) from which the stream was obtained. For streams at nodes closer to the "source" (*i.e.*, end users), temporal locality is due mostly to popularity over short time scales, whereas for streams at nodes closer to the "destination" of the stream (*i.e.*, the origin server), temporal locality is due mostly to popularity over long time scales. This difference can be explained by the higher levels of flow aggregations at proxies closer to the destination, which tends to "dilute" popularity over shorter time scales, and "emphasize" popularity over longer time scales.

While the above discussion has centered mostly on web request streams, it should be evident that the same phenomena apply to many other data stream systems. For instance, consider streams of stock market data used to answer queries [37]. Temporal locality in such a stream could be a reflection of the market capitalization of various companies, or a reflection of breaking news that impact stock prices of specific companies (which may or may not be otherwise "popular"). Clearly, the former would result in locality due to popularity over long time scales, whereas the latter would result in locality due to popularity over short time scales.

To validate this hypothesis, we obtained traces from mutiple days of real stock transaction data from [17] and characterized the popularity of each stock as well as the corresponding average stack distance for that stock. Figure 2.1 shows the highly-skewed popularity profile of stocks in the trace for one of the days in our trace. The relationship between frequency and rank (a straight line with slope $b = 0.712559 + / - 0.003563 (0.5\%)$ on a log-log scale) is consistent with a Zipf-like distribution. Figure 2.1 shows the CDF of the stack distance for the stock trading trace, and for a randomly permuted version of that trace. Clearly, the two distributions are quite different, with the original trace exhibiting much stronger reference locality properties. This indicates that a significant source of reference locality is due to correlations over short time scales–correlations that are not captured well (or at all) using long-term frequency measures.



(a) Zipf-Like Popularity Profile for Stocks in a Real Stock Trading Trace.



(b) CDF of Stack Distance for Original and Randomly Permuted Stock Trading Trace.

Another example of popularity over long versus short time scales concerns Origin-Destination pairs (*a.k.a.*, OD flows) observed in network packet streams. In [24], large traces of OD flows in two major networks (US Abilene and Sprint-Europe) revealed that traffic intensity at any router is merely the superposition of OD flows with three different characteristics: those with periodically changing intensities, where the period is over a fairly long (diurnal) time scale, those with bursting intensities over a fairly short time scale (*e.g.*, subseconds as shown in [31]), and those with random intensities. Yet another example of data stream systems with data exhibiting locality over multiple time scales include sensor networks. Here, one would expect that data values that are "popular" locally (*i.e.* in the neighborhood of where they have been collected) may not be so over an entire sensor field. This would give rise to different relative strengths for the two sources of temporal locality at different locations (depending on the levels of agreggation, for example).

2.2 Caching Implications

To be effective, a cache replacement strategy must be able to delineate between the above-mentioned sources of temporal locality (and adjust its behavior accordingly). LFU is an example cache replacement strategy that capitalizes on long-term skew in popularity (*i.e.* frequency). LFU always evicts the object with the lowest reference count. LFU is online-optimal under a purely Independent Reference Model[1] [10]. LRU is an example cache replacement strategy that capitalizes on shorter-term residency. LRU is the most widely used cache replacement algorithm, as it captures recency (which indirectly captures frequency as we will see below) and was shown to be superior to other policies, *e.g.* FIFO and Random. Several studies have considered both recency and frequency information [25, 30]. For example, the LRU-K [30] algorithm maintains the last K reference times to each object to compute the average reference rate.

In the remainder of this section, we contrast the benefits of using a caching algorithm that leverages *both* sources of temporal locality (such as GDJ, the technique we propose later in this paper) to one which only leverages long-term popularity (such as the frequency-based technique considered in prior work [11]).

As explained earlier, the marginal distribution of the stack distance for a request stream captures the temporal locality present in that stream. That is, if D is a random variable corre-

[1]IRM assumes that a request stream consists of a sequence of independent, identically-distributed random variables.

sponding to stack distance (with distribution function F_D) and $H(C)$ is the hit rate of a cache that can hold C entries then

$$P[D < C] = F_D(C) = H(C)$$

Thus, knowledge of the distribution function F_D provides enough information to predict the performance of a cache of any size for the given trace.

For a cache of size C, the difference between the hit rates achievable by a locality-aware approach versus that achieved by a purely frequency-based approach can be estimated by contrasting the values of $F_D(C)$ for the distributions of stack distance of the original trace and that of a randomly permuted version thereof. From Figure 2.1, we can see that for a cache sized at 10% of the working set size, a locality-based approach could potentially be twice as effective as a frequency-based approach. Naturally, the differences between these approaches is exacerbated when the cache size is small.

2.3 Dealing with Non-Uniform Values of Cached Entries

In many applications, including the one considered in this paper, entries in the cache may not be equally valuable. For a web caching application, the miss penalty for one entry may be much higher than another due to differences in bandwidth between the cache and the corresponding origin servers of these entries. For join processing, some tuples may be more successful in producing join results than others. The basic frequency/recency-aware and hybrid techniques mentioned earlier do not consider variable-cost objects. For instance, in LRU the most recently accessed object has (by definition) the highest value. The GreedyDual(GD) algorithm [36] can be viewed as a generalization of LRU which takes into consideration both the "cost" (or value) of an object as well as recency information. GD was shown to be online optimal in terms of its competitive ratio. Using GD, the object with the lowest "value" (H) is replaced, and the value of all objects in the cache is reduced by that value. When an object is accessed, its value is "restored" to its original value. Effectively, GD implements an aging technique whereby the value of an object slowly decreases until it is evicted (unless its value is replenished as a result of being accessed). In [21], Jin and Bestavros proposed a generalization of GD called GreedyDual* (GD*). GD* uses the metrics proposed in [19] to *tune* GD's aging algorithm (possibly in an on-line fashion) to capitalize *optimally* to the relative strengths of the two sources of temporal locality discussed earlier.

3 Model-Based Approaches

In this section we discuss the most important previous approaches to computing sliding window stream joins with limited memory. The simplest approach is to use random load shedding and drop tuples with a rate proportional to the rate of the stream over the size of the buffer [22]. However, this approach does not optimize the Max-subset criterion and has been shown to perform poorly in almost all cases [11]. A better approach is to assume a model for the stream and drop tuples selectively using that model. This approach is called *semantic* or *informed* load shedding. Next we discuss two model-based approaches and point out their strengths and weaknesses.

3.1 Frequency-Based Stream Model

The frequency-based model was implicitly used in [11] and formally defined in [32]. Under this model, the probability that the join attribute of a new tuple s has a certain value v is independent of the arrival time of s and is always $f(v)$. This is equivalent to the IRM model [10] mentioned in the previous section. Based on this model, an online algorithm called *PROB Heuristic* was proposed in [11]. This algorithm could be viewed as an adaptation of the LFU for sliding window stream join. *PROB Heuristic* assumes a single join between two streams S and R, and assumes that each stream has a separate buffer B_S and B_R respectively. Furthermore, it assumes that the probability of value $f(v)$, $\forall v \in \mathcal{D}$ is known for both streams. A simple approach to approximate that is to build a histogram of observed (past) tuple values [13, 15] and use that as a model for future arrivals. When the buffer for S is full and a new tuple arrives, the tuple that has the value with the smallest probability to appear in the other stream, e.g. R, is dropped from B_S. A symmetric technique is used for a new tuple in R.

The advantage of this model is its simplicity. Furthermore, the model can be used to perform sampling over the join results and buffer allocation in a simple way [32]. However, as we mentioned earlier in the paper, the LFU approach leverages long term popularity, but fails to capitalize on possibly strong temporal correlations over shorter time scales.

3.2 Age-Based Stream Model

A completely different approach to model stream arrivals was proposed recently in [32]. In this model, called the *age-based* model, the expected join multiplicity of a tuple is independent of the value of the join attribute and depends *only* on the arrival time. In particular, the number of join results produced by a tuple depends on the "age" of the tuple. Using this idea, a different load shedding technique called *AGE* was proposed. First, it is assumed that the system knows the age curve of each tuple and that the tuples in each stream follow the *same* age curve. Let, t be the time that a tuple s of S arrived in the system and $p_S(k)$ be the number of join results produced by s between time instants t+k-1 and t+k (that is, when the age of s is between k-1 and k). The $p_S(k)$ depends only on k and not on s. Defining $C_S(k) = \sum_{j=1}^{k} p_S(k)$, we can compute the value k_S^{opt} ($k \leq W$) that maximizes the ratio $\frac{C_S(k)}{k}$. The algorithm to drop tuples uses only this value. If the size of the buffer in stream S is larger than k_S^{opt}, e.g. $B_S > k_S^{opt}$, then a simple FIFO technique is used to manage the buffer. On the other hand, if $B_S < k_S^{opt}$, a simple staggered approach is adopted. The first B_S tuples are kept in the buffer for k_S^{opt} time instants each. When a tuple in the buffer gets "older" than k_S^{opt}, it is removed from there and the next arriving tuple takes its place. If the buffer is full and all tuples in the buffer have age less than k_S^{opt}, all incoming tuples are dropped.

The above model is appropriate for only a specific set of applications (i.e. on-line auctions) that follow the proposed model of arrivals. Even in that case, the assumption that all tuples follow the same aging process can be very restrictive. Indeed, in many real data streams the above model does not hold, or if it is used, it may perform poorly (as we will show in our experiments later in the paper). The reason is that the method assumes that all tuples have the same age curve and therefore the AGE algorithm keeps only a single value (k_S^{opt}) for each stream (in [32] the average value for $p_S(k)$ is used to estimate k_S^{opt}). However, in practice, tuples may have completely different k_S^{opt} values and an average value can be an equally bad estimate for all the tuples. This turned out to be true *in all* the stream data we used in this paper–both real and synthetic. For instance, Figure 2 shows the distribution of k^{opt} computed for each value in the stock data stream. The large variance in that distribution suggests that any "representative" value for k^{opt} will only work for a small percentage of the tuples.

To summarize, techniques that rely either explicitly or implicitly on a particular model—namely an assumption about the marginal utility of keeping a particular tuple based on its historical popularity or some assumed aging process—will perform in accordance with the accuracy of such a model as a predictor of real utility. Given the fundamental relationship between the efficiency of cache management and reference locality properties, we argue that the most natural models are those that capture locality of reference properties, and that the most efficient

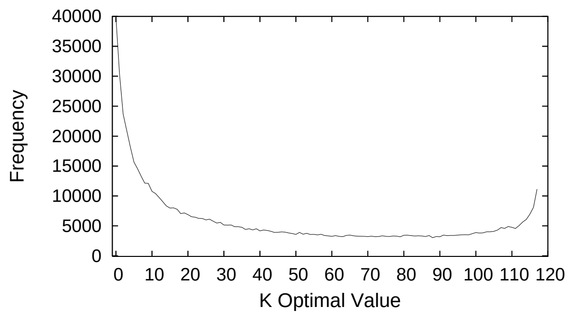


Figure 2: k^{opt} distribution for stock data stream

buffer management algorithms are those that capitalize on such properties. We propose such an algorithm next.

4 The GreedyDual-Join Algorithm

In this section, we propose a novel locality-aware algorithm for approximate data stream join processing. Our algorithm—which we refer to as the GreedyDual-Join (GDJ)—attributes a value to each entry in the cache based on the expected "profit" from keeping that entry in the cache. For a newly arriving entry, the value is set to some initial expected profit. Over time, this value is incremented when the cached entry participates in producing a join result, and is decremented otherwise. The initial value and the increase/decrease rules we use are detailed later. GDJ is in fact an extension of the GreedyDual (GD) algorithm [36] in the same way that GreedyDual-Size (GDS) [7] and GD* [18] are extensions, in the sense that it allows the value attached to a cache entry to depend on its potential for producing join results (as opposed to its "size" as with GDS, for example).

Classic cache replacement algorithms (such as LFU and LRU) could be seen as special instantiations of our algorithm. For example, with LFU, the initial value of an entry is set to a constant, which is proportional to the number of joins that the entry is expected to produce based on the frequency of past join results, under an assumption of stationarity.[2] LFU assumes that the cost of an entry is constant and thus there are no increase or decrease rules, and an entry is replaced if its value is less than a newly arriving entry. With LRU, the value of an entry reflects its distance from the "bottom" of the LRU stack. The increase rule is simply to set that value to the maximum possible value, the LRU stack size (*i.e.* the number of objects the cache can hold),[3] whereas the decrease rule is to decrement that value by 1.

GDJ has a number of desirable features: (1) By employing an aging mechanism (*a la* LRU) GDJ allows the buffers to retain entries that have been most valuable over a timescale that is proportional to the buffer size. (2) GDJ is readily applicable to Multiple Streams/Multiple Joins, since the value of an entry would reflect the contribution of that entry to *all* joins, irrespective of the number of streams or attributes over which these joins are defined. (3) GDJ has a constant (memory) cost of *O(B)*, where B is the total number of tuples that can be buffered in the system. Note that, if each tuple has multiple attributes, then $B << M$ (a similar assumption is used in [32]).

[2]Notice that this is an on-line approximation of a "perfect" LFU, which must consider frequency of future references.

[3]This is also the initial value attached to an entry.

4.1 GDJ Algorithm Description

We assume that we have a join over two streams R and S, with two separate buffers B_R and B_S and a common time-based window W.[4] Every tuple in each buffer is associated with a *credit* that is used in buffer management. Also, we assume that time is discrete.

Algorithm 1 GDJ(Stream S,R; Window W; Buffers B_s,B_r)

```
new tuple s arrives in S at time $t_c$
join s with $B_r$
if $B_r[i]$ joins with s then
    increase credit of $B_r[i]$
Remove old tuples from $B_s$ (tuples arrived before $t_c - W$)
if $B_s$ is full then
    Remove $s_i \in B_s$ with smallest credit
Assign initial credit to s
Insert s into $B_s$
Decrease credits in $B_s$ and $B_r$
Return join results
```

When a new tuple s comes in stream S, we use one of the existing join algorithms (e.g., [22, 14]) to perform the join of this tuple with the buffer of the other stream R. For every tuple in B_R that produces a join with s, we update its credit. Next, we remove the tuples from B_S that have expired. If B_S is still full, we have to chose a tuple to evict and insert the new tuple. We do so by removing the tuple with the least credit. Next, the new tuple s is assigned an initial credit and is inserted in the buffer. Finally, at the end of each time unit, we decrease the credits of the tuples in both buffers.

It is clear that GDJ is a general algorithm that can be instantiated with different choices for a number of parameters. There are three important components in GDJ, namely: a) Initial Credit Assignment, b) Credit Increment Rule, and c) Credit Decrement Rule. Next we discuss possible options for each one of these components:

Initial Credit Assignment: When a new tuple is inserted into a buffer, it must be assigned an initial credit, which should range between the minimum and the maximum credit of the tuples currently in the buffer. The value of the initial credit can be thought of as controlling the length of time we are willing to wait on the new tuple to "prove its worth". A typical approach is to assign a percentile of the current credits. The optimal setting for such a percentile would depend on the stream characteristics, and could be easily tuned in an on-line fashion if the stream characteristics are not stationary. For the traces we have considered, the optimal initial credit assignment ranged between 0.8 to 0.99.

Credit Increment Rule: When a tuple in the buffer produces a join result its credit must be updated. The simplest approach is to increase the credit by one. Another approach is to increase the credit using the rank of the tuple (or some function thereof), assuming tuples are ordered by credits.

Credit Decrement Rule: At each time instant, the credits of all tuples in a buffer are decreased to reflect aging. The simplest approach is to skip this step (or equivalently decrease by zero). Another possibility is to decrease the credits in such a way that the total credits remain constant. Therefore, at each decrement step, the value is different and depends on the previous increment step.

[4]Later in this paper, we consider the buffer allocation problem.

As we hinted earlier, for different choices of the above rules, the GDJ algorithm reduces to other well known (basic) techniques such as LRU, LFU, and FIFO.

4.2 Sampling with GDJ Algorithm

Uniform random sampling over windowed join results that guarantees a given sampling fraction is hard in general [32]. However, sampling of the PDF of the exact result (termed cluster sampling in [32]) is possible. PDF sampling is useful because it can be used to compute unbiased aggregates and other statistical properties of the result.

One simple approach that would result in an unbiased sampling of the PDF of the windowed exact join results is to use random load shedding ahead of a FIFO buffer. Using that approach, upon arrival a tuple from a stream S is dropped with a probability p, otherwise it is added to the FIFO buffer. Setting the probability p to $1 - B_S/(\lambda_S W)$, where λ_S is the stream rate, will ensure that on average, a tuple will remain cached in the buffer for W units of time (this follows directly from Little's Law). Clearly, one can see that the stream that survives the dropping step is a random sample from the original stream. If this load-shedding FIFO buffering is applied to both streams S and R, then the PDF of the resulting join could be used as an unbiased approximation of the PDF of the exact windowed join results. Notice that if the distribution of the exact windowed join is stationary, then one can show that the results obtained using load-shedding FIFO buffering will asymptotically approach the exact distribution.

The major issue with the above load-shedding FIFO buffering is precisely that the resulting distribution will only approach the exact one asymptotically, in the sense that it may take a very long time for enough results to be produced due to the fact that join results of "rich" items may be dropped (and exactly these tuples contribute more to the aggregate [32]). To alleviate this, we would need to reduce the value of p. However, doing so will result in tuples leaving the FIFO queue in less than W units of time, which would bias the results as the effective window size would be reduced.

Notice that if we replace the FIFO buffer with a GDJ buffer, the resulting join results will be identical and hence an unbiased approximation of the exact PDF. This can be seen by noting that $p = 1 - B_S/(\lambda_S W)$ will equalize the rate of tuples into the buffer (due to arrivals that survive the dropping operation) and out of the buffer (due to tuples reaching their expiration time after W units of time). If the rates into and out of the GDJ buffer are equal, then GDJ is no different from FIFO since no replacement will be necessary.

Now consider the implications of reducing the value of p below $1 - B_S/(\lambda_S W)$ for a load-shedding GDJ buffer. Once we do so, the advantages of GDJ's cache replacement will start playing out, which would result in a better recall rate than under a load-shedding FIFO buffer (while possibly biasing the resulting PDF).[5] As p is further decreased, this trend will continue, until $p = 0$ at which point we get the most number of joins, but without the guarantee of an unbiased PDF approximation. Notice that the bias will be more pronounced for smaller-sized buffers, as this will force GDJ to discard more tuples that could have produced join results, if they were allowed to remain in the buffer. For larger buffer sizes, GDJ's bias will be negligible since dropped tuples are likely to be the ones that would not have produced a join, even if they were allowed to stay in the buffer. Thus, one may view load-shedding GDJ buffering as providing a convenient continuum of tradeoffs between the conflicting goals of PDF approximation accuracy and recall rate.

[5]Notice that one may replace the FIFO buffering with any type of buffer management, including the model-based approaches we discussed earlier. However, we also note that the bias introduced in the sampled PDF will be less pronounced for buffer management approaches that make better eviction decisions.

4.3 Unified Buffer Management using GDJ

As we hinted earlier, unified buffer management allows tuning of the allocation of the available memory to each stream (and/or each join) to maximize some performance metric (e.g., recall rate).

GDJ can be readily used for unified buffer management by allowing the credit of each tuple in the unified buffer to reflect the global value of that tuple across all streams/joins (and not only its value relative to other tuples in the same stream). In other words, it is possible for a tuple from one stream to displace a tuple from another stream. To do so, the initial credit, increment, and decrement rules must be chosen so as they reflect the perceived global value of a tuple over all streams and joins. Another (coarser) alternative to using GDJ for apportioning available memory to different joins is to simply use the average credits per tuple for each join.

For the GDJ unified buffer management experiments presented later in this paper, we have used the same initial credit and the same increment/decrement rules for all streams/joins. When an eviction is necessary, we pick the tuple with the smallest credit (regardless of which stream it belongs to).

5 Performance Evaluation

We compared GreadyDual-Join (GDJ) against the other existing methods, namely, the random eviction (RAND), the First-In-First-Out (FIFO) method, the frequency-based eviction (PROB) [11] and the AGE algorithm [32] on both synthetic and real data sets. In addition, we also compute the exact results by supplying memory enough to contain the full windows (FULL). Some of the results of the approximate methods are presented as fractions of the exact results produced by FULL. In all our experiments we used time based sliding windows.

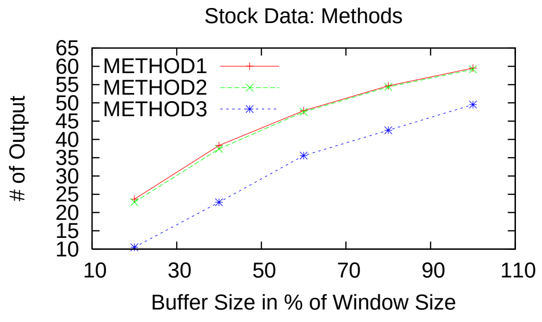
5.1 Experiment Setup

We implemented GreadyDual-Join as it is discussed in the previous section. As we discuss next, we tested different methods for initial credit assignments and increment/decrement rules and we chose the best combination. For the PROB algorithm, we keep track of the exact number of occurrence of each individual value, so it utilizes a precise count in its probabilistic measure of frequency. Hence, it should perform at least as good as any other implementations of this algorithm using summarization techniques such as histograms. Similarly, the implementation of AGE first computes the exact k^{opt} value for each tuple offline in order to optimize its performance, whereas in reality this process is impractical and computationally expensive.

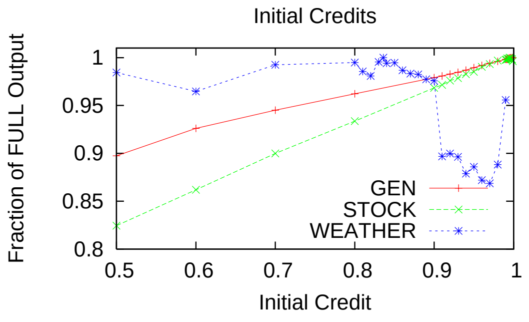
5.1.1 Synthetic Data Streams

To generate the synthetic data sets, we created a data stream generator [6] with adjustable parameters based on the temporal locality model developed in GISMO [20] for Internet streaming media traffic. The distribution of the join attribute values is skewed and follows a Zipf distribution so that the frequency of a value is inversely proportional to its popularity (rank), i.e. $P(r) \sim r^{-\alpha}, 1 < r \leq N$, where N is the number of objects, r is the rank, and P is the access frequency of the r-ranked value. To model the temporal correlation, we characterize the inter-arrival times of each value using Pareto distribution with an adjustable skewness parameter such that the temporal correlations are inversely related to the inter-arrival times. In our experiments, the default object domain size is set to 100 and the simulation runs for a data stream of one day with a time unit of one minute. The default value for the skewness parameter α is 0.75, unless otherwise specified.

[6]The data generator as well as the datasets and the code that we used for our experiments are available on the web: http://cs-people.bu.edu/jching/gdj/



(a) Increment/decrement rules



(b) Varying initial credit

Figure 3: Tuning GDJ parameters

5.1.2 Real Data Streams

For the real world data sets, we use U.S. stock trading data [17] and Pacific Northwest weather data [29]. In the weather data, each tuple contains the date and time, dewpoint temperature, air temperature, wind speed, cloud height, etc. We execute a sliding-window join on the dewpoint with two streams of three-month long weather data extracted from two different years. The time unit is set to 1 minute with a constant rate of one tuple coming in per time unit. For the stock data, each 'time-sand-sales' transaction contains records of stock name, time of sale, matched price, matched volume, and trading type (buy or sell). Every transaction for each of the top 500 most actively traded stocks, by volume, is extracted, which sums up to about one million transactions throughout the trading hours of a day. The time unit is set to 1 second with an average rate of 23 transactions coming in per time unit.

5.2 Tuning GreedyDual-Join Algorithm

As discussed earlier, GreedyDual-Join is a general and flexible algorithm that can be implemented in a number of different ways. In the first set of experiments we investigated different choices for the three main components of the algorithms.

For the increment/decrement rules, we tried the methods described in section 4.1. In particular, we tested three methods: 1) a simple approach (METHOD 1), where the increment rule is to increase the credit by one combined with zero decrement, 2) another approach (METHOD 2) that increases the credit using the rank of the tuple and decreases the credits by one, and 3) the last approach (METHOD 3), where we increase by one the credits of the matched tuples and decrease the credit of each tuple by the ratio of the total credits just added in the current pass over the total number of tuples in the buffer. Note that in METHOD 3, the total credits across the buffer remains a constant at the end of each pass. The results for the Stock dataset are shown in Figure 5.1.2. Note that METHOD 1 gives similar results to METHOD 2. On the other hand, this approach is much faster and therefore it gives a good trade-off between efficiency and performance. Thus, for the rest of our experiments we used METHOD 1 as our approach for increasing and decresing credits.

We also experimented with a wide range of initial credit assignment values (Figure 5.1.2), and found that their performance vary significantly on different data sets (rank 0.995 percentile works the best for stock data, 0.836 for weather data, and 0.998 for synthetic data). The choice of initial credit assignment values is closely related to the distribution of stack distance. For

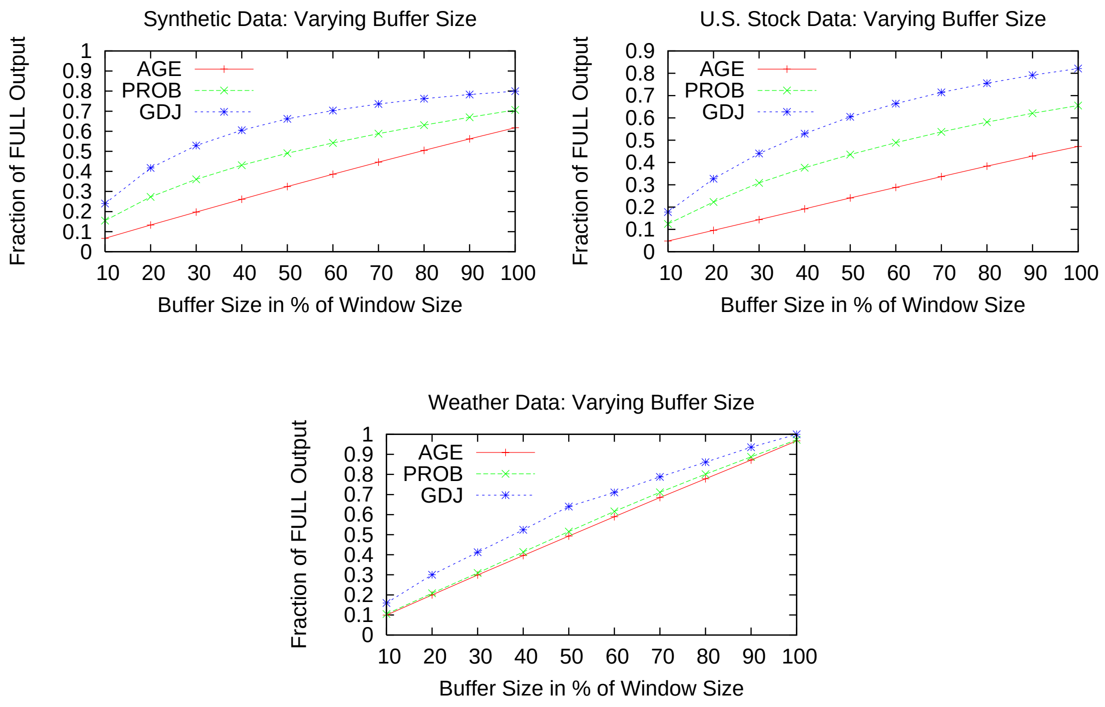


Figure 4: Varying buffer size

example, if the stack distance distribution is highly skewed, it is likely to see a value to reappear almost immediately if it were ever to reappear again anytime soon. Therefore, by setting a low-rank credit to newcomers in a highly skewed data set, it would discourage unproductive items to stay too long and yet provide adequate time to capture the skewed temporal locality. The stack distance characteristics could be obtained by the prior knowledge of the data set or by observing the history statistics. A practical approach would be to build a dynamic feedback system that automatically tunes the algorithm by adjusting its initial credit value.

5.3 Maximizing the Join Output

5.3.1 Effect of Buffer Size

In this set of experiments, we study the effect of the buffer size as a fraction of the total memory requirements for computing the exact results. As shown in the Figure 4, with an increasing buffer size, the performance is always increasing and GDJ outperforms PROB and AGE approaches in all cases. GDJ's outstanding performance over PROB complies with our previous analysis of the existence of temporal locality in data streams. Note that the percentage of buffer size is calculated base on the average arrival rate. Hence, even when a 100% buffer size is supplied, the buffer cannot retain all tuples at times when the rate is higher than the average rate. This is particularly true for stock data since its arrival rate is quite bursty. On the other hand, the weather data set achieves convergent exactly at 100% buffer size since its arrivals are synchronized at one per minute. AGE algorithm's performance increases linearly, which resembles the behavior of random eviction approach. Its poor performance could be explained by the failure of the data set in complying with the age-model assumption. We found that not only do the data not share the same k optimal value (which we solved by computing and using the exact k^{opt} for each tuple), but also most tuples violate the two major assumptions - the nonexistence of minima and the same productivity of join results for every tuple in the same stream. Since the popularity distribution is highly skewed (Zipf), it is very common to see tuples that produce very little are kept much longer than they deserve. On the other hand, there are also many tuples that are very productive but are underestimated due to the presence of minima.

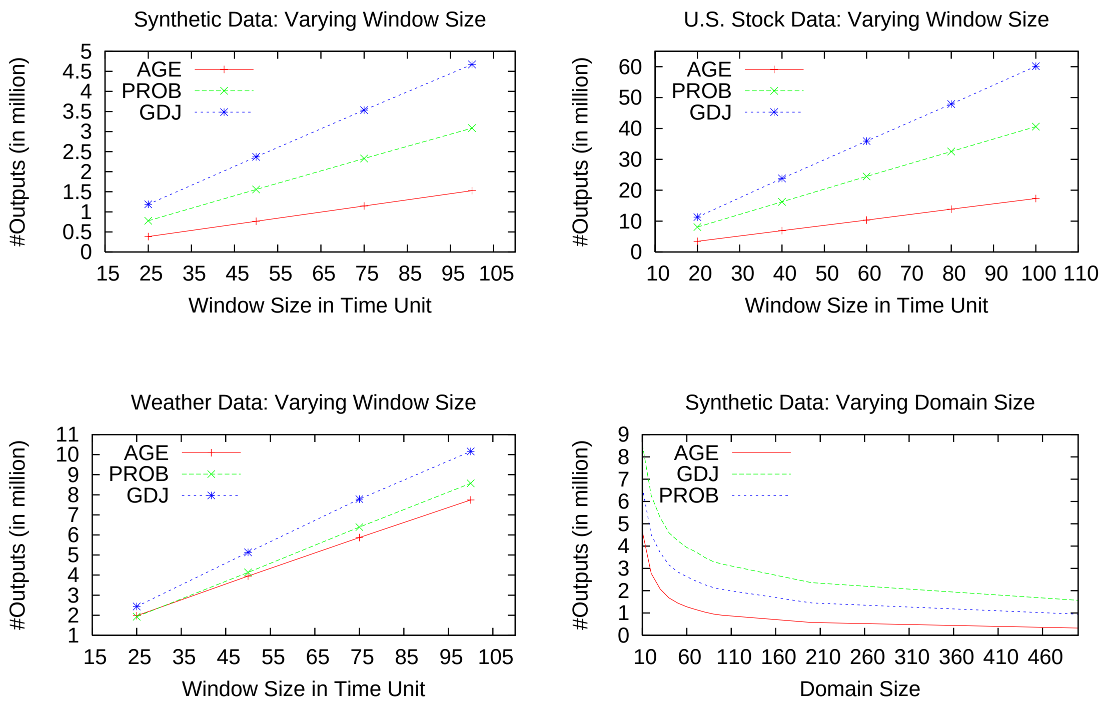


Figure 5: Varying window size

5.3.2 Effect of Window Size and Domain Size

To study the scalability of our algorithm, we execute the same set of experiments using a wide range of window sizes on a 20% buffer size. Figure 5 shows that the behavior of each algorithm is remarkably persistent for different window size, which, interestingly, complies with the observation and experimental results by Das et al's [11]. To observe the effect of domain size on the joining attribute, we generate data sets of various attribute cardinalities using our synthetic generator. Similarly, the performance of each algorithm is consistently decreasing (Figure 5). Therefore, neither window size or domain size play an effectual role on the performance. For all these cases, the GDJ outperforms the other algorithms by a large factor.

5.3.3 Unified Buffer Management

In the previous experiments, the available memory is allocated to each stream in proportion to their rate [22]; this method is abbreviated as PROP. In this experiment, we compare this naive approach with the unified buffer method. In that case, the available memory is not divided into different buffers, but all streams have access to the whole memory as a single buffer. Next, we compare the unified memory version of GDJ and RAND with the naive proportional buffer allocation of PROB and RAND. We consider two joins, each one over two different streams; thus the total number of streams is four. All streams follow the Zipf distribution with the same skewness except one pair of the streams is directly correlated (DC) and the other pair is inversely correlated (IC). A similar experiment is used in [32] to study the effectiveness of maximizing the minimum recall of multiple joins. In Figure 6, each bar represents the total number of the join output. As expected, the performance of RAND would remain the same since in both PROP and UNIFIED every tuple in either stream has the same chance of being evicted. On the other hand, UNIFIED-GDJ approach significantly outperforms PROP-PROB because the credit system of GDJ successfully identifies that tuples from one join are more productive than the other since they have higher credits. Therefore it allocates more space for the most productive join and that maximizes the output results.

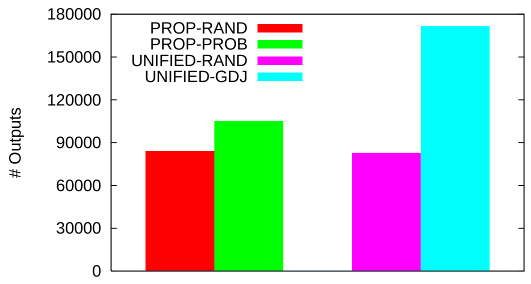
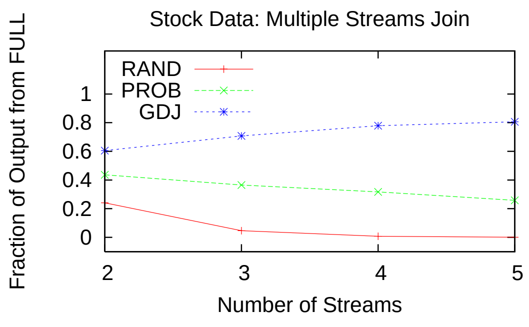


Figure 6: Unified Buffer Max-subset results

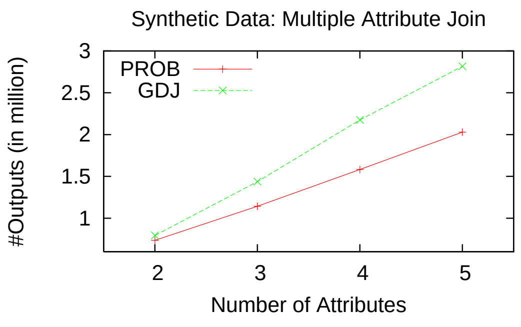
5.3.4 General Workload: Multiple Streams and Multiple Attributes

As we mention earlier, an advantage of GDJ is that it can be easily extended for multi-way and/or multiple sliding window joins. Here we investigate two interesting cases. First, we consider a multi-way sliding window join, where the number of streams in the join ranges from 2 to 5, and the join is defined over all participating streams (e.g. $S1[W] \bowtie S2[W] \bowtie ...Si[W]$). We used the stock market data streams for this experiment. In another experiment, we used two streams with multiple attributes and we applied multiple joins on both streams using different attributes. For example, for two streams S and R, each with two attributes (s_1, s_2) and (r_1, r_2), respectively, we defined the following joins: $S[W] \bowtie_{s_1=r_1} R[W]$ and $S[W] \bowtie_{s_2=r_2} R[W]$. The streams were generated using our data generator. Since the methodology of computing k^{opt} on these general problems has not been discussed in [32], it is unclear how to apply AGE here. From this point on, randomized eviction (RAND) or FIFO is used for comparison instead.

Multiple Streams Figure 5.3.4 shows that GDJ is able to cope with multi-stream join graciously whereas the performance of PROB and RAND degrade gradually. Again, the results are normalized by the output from FULL and 20% buffer is provided for each stream. This experiment shows that the effect of temporal locality remains even with an increasing number of streams, and in fact, it is increasingly dominant. It is obvious that random load shedding will deteriorate with the increasing number of streams, since in order to produce a join result, all the streams must have the same value on the joining attribute and the probability of that to happen is decreasing with an increasing number of streams. Using the same argument, keeping the most popular items of the next stream is not always the best thing to do if it wipes out the existence of a less popular tuple that is temporally popular in other streams. On the contrary, GDJ, which values not only the long-term popularity but also the temporal locality, allows every tuple a trial period to express its productivity in all streams for a short time period. The experiment has demonstrated that this merciful tactic outperforms all other techniques in real data sets.



(a) Multiple-stream-and-multi-way join



(b) Multiple Attribute Results

Figure 7: Multiple Stream and Multiple Attribute Results

Multiple Attributes

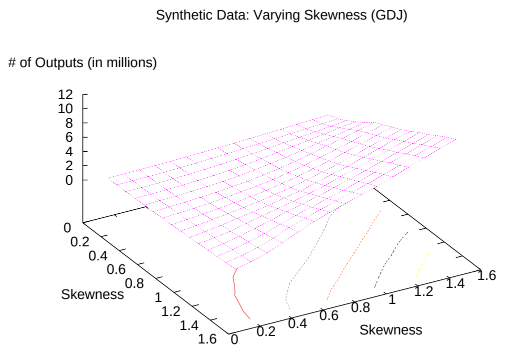
We execute multiple joins on two streams with varying number of attributes. The attributes on one of the streams are Zipf distributed with increasing skew for each additional attribute; the attributes in the other stream are uniformly distributed. The computation of statistics in PROB is extended to include multiple attributes and the credit in GDJ is updated to aggregate the credit of all attributes. Figure 5.3.4 shows that GDJ still outperforms the other methods in all cases and it gets better as the number of attributes increases.

Effect of Varying Skewness Parameters

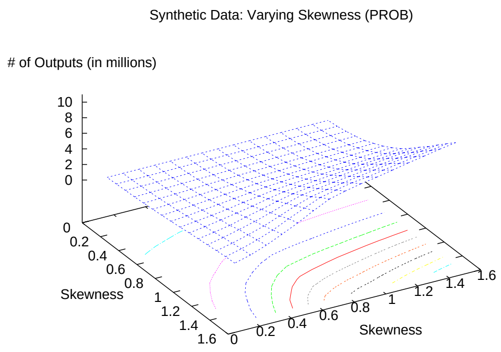
In our multiple attribute experiments, we observe that the skewness of the popularity has an interesting effect on the performance of our algorithms. We execute a two-way join on each corresponding attribute from two two-attribute streams produced by our synthetic data generator. The distribution of the values for the first attribute is set to uniform, whereas the second attribute of each stream varies from 0.1 to 1.5 skewness level (where 0.1 is uniform and 1.5 only contains the most popular object in the join attribute). Figure 8 shows that the plane curvatures representing the number of output produced by PROB and GDJ are of very different shapes, however, they both produce more when the popularity of the second attribute has more skew. By overlapping the planes, we observe that they occasionally intersect each other. For some small values of α the PROB performs better than GDJ. This is expected since the dataset is close to uniform and the temporal correlation is limited. However, as we increase the value of α GDJ becomes better than PROB as expected.

5.4 Sampling

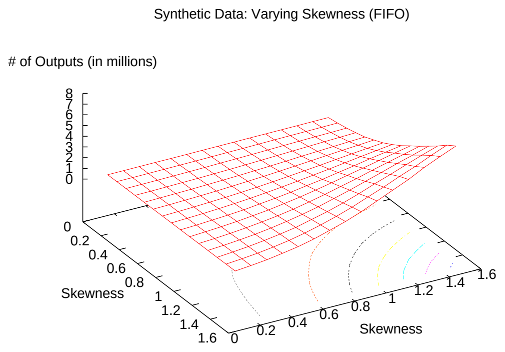
In the last set of experiments, we tested the performance of GDJ against PROB and FIFO for the sampling problem. We used the two real datasets, namely Stock and Weather data streams. We used a buffer of 20% of the window size and we implemented the method discussed in section 4.2. We changed the drop rate p from 0% to 80% and we computed the number of total join results produced. Also, we used the output to create a histogram that represents the distribution of the output result. We first normalize the histogram (by dividing each frequency with the size of the total output) and then we computed the “difference” or “distortion” between the histogram of the approximate results against the histogram of the exact results. The method that we used to compute the difference was the Sum Squared Error (SSE). Note that other possible methods to compute the difference between two distributions exist, such as the Kullback-Liebler (KL) distance



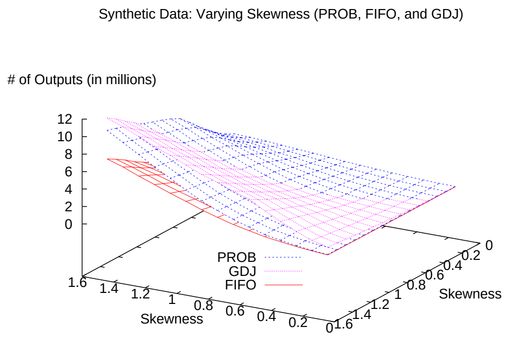
(a) GDJ: Varying skewness



(b) PROB: Varying skewness



(c) FIFO: Varying skewness



(d) GDJ, FIFO and PROB: Varying skewness

Figure 8: Varying skewness experiments on FIFO, PROB, and GDJ

(Figure 10). In Figure 9 we plot the total number of output and the distortion for PROB and GDJ for different values of the drop probabilities. As expected, for drop values close to 0%, the output is maximized, but at the same time the distortion is large, since both methods introduce bias that favor the most frequent/productive tuples. On the other hand, as the drop rate goes close to 80%, the methods start to behave like FIFO. This means that the distortion gets smaller, but at the same time the output is decreased. However, still GDJ is better than PROB. For the same distortion, say 0.02, the output of GDJ is more than 18 million tuples and the output of PROB is less than 15 million. Also, for the same number of output tuples, the distortion of GDJ is much smaller than the distortion of PROB. We do not show the results for FIFO because the output and the distortion are very small. The above fact is important, since larger output can be used to produce better aggregate results as suggested in [32].

5.5 Discussion of Experimental Results

In all our experiments GDJ outperformed the other competitors (namely, PROB, AGE, FIFO and random) by a large margin on both synthetic and real datasets. GDJ has two more advantages compared to the other techniques: First, it is adaptive and makes no assumptions about the data

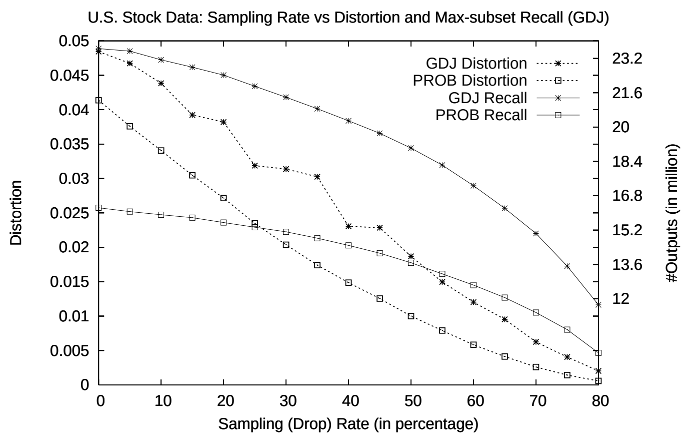


Figure 9: Effect of sampling rate on distortion (in Sum of Squared Error) and number of recalls

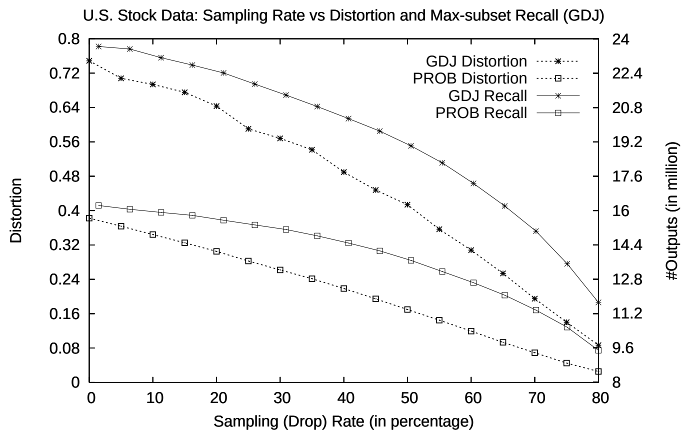


Figure 10: Effect of sampling rate on distortion (in KL Distance) and number of recalls

distribution, which means that it will work well, even for data streams with non-stationary data distributions (as soon as the distributions change slowly). Second, it is simple to implement with minimal CPU overhead. The only disadvantage of GDJ compared to AGE and PROB is that it uses constant space (although usually very small) for each tuple in the buffer. However, in many cases, tuples in data streams contain many attributes and in that cases the space overhead of GDJ is small and comparable to the other methods.

6 Other Related Work

Stream Databases have received a lot of interest recently. General issues and architectures for stream processing systems are discussed in [4, 6] (Stanford's STREAM), [23] (TelegraphCQ), and [8] (AURORA). Continuous queries are studied in detail in [6, 26], where the authors proposed methods to provide the best possible query performance in continuously changing environments. Load shedding is used in [33, 5]. In [33] a utility based load shedding scheme is introduced for a general data stream manager that executes a query plan. The utility is defined for processing each tuple from a data stream that is specified by quality of service (QoS) functions. It is limited by the usage of utility for each individual tuple. A load shedding approach for aggregation queries is presented in [5]. The core problem there is where and how much of the tuples must be dropped in order to minimize the error in approximating aggregate queries. Random sampling for estimating the selectivity of join results in the traditional relational databases has been considered in [9]. However, most of the techniques presented there are optimized for relations with fixed size and require multiple passes over the data. Also, the main goal is to estimate the selectivity of the joins and not maximizing the output.

Also, a number of stream join processing methods have been presented recently. In [16] scheduling issues for shared window joins are discussed. By carefully scheduling the join sequence for a set of joins with different window sizes over the same stream data, the response time of the system can be significantly improved. Multi-way stream join queries are studied in [34]. The problem addressed there is mainly the disk-memory coordination for evaluating these joins. Join over multiple streams also has been considered in [14], where they analyze the cost of various join algorithms. However, in both of these works, they did not consider approximate results and load shedding issues.

The most related work to ours except the ones discussed in section 3, are [22, 35]. In [22], a technique based on a unit-time cost model is proposed. The scheme selects the join implementation and memory allocation for the two input streams according to their arrival rates. Load is then shed by simple random eviction. In [35], a stochastic based approach for buffer management in sliding window join scenaria has been proposed. The authors assume that the data distributions are shifting over time and the system knows a stochastic model about the tuples, e.g. how the value distributions are changing. However, the method for inferring the general stochastic model is not clear.

7 Conclusion

In this paper we proposed and evaluated an adaptive buffer management techniques for approximate evaluation of sliding window joins over multiple data streams. We did so by casting the problem of buffer management for approximate join processing as a generalized cache replacement problem. Our GreedyDual-Join (GDJ) approach—which could be thought of as a generalization of the well-known Greedy-Dual (GD) algorithm [36]—enables the reference stream of one cache to affect the cache-ability of entries in other caches. Among its many features, our approach capitalizes effectively on temporal locality properties in the streams, allows an integrated buffer management for all streams, and is readily amenable to approximate join processing involving multiple joins on multiple attributes in multiple streams. In support of our claims, we have presented simulation results using synthetic as well as real traces, which confirmed the superiority of GDJ.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS*, December 1996.

[2] A. Arasu, B. Babkcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirement for queries over continuous data streams. In *Proc. of PODS*, 2002.

[3] M. Arlitt and C. Williamson. Web server workload characteristics: The search for invariants. In *Proceedings of ACM SIGMETRICS'96*, May 1996.

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

[5] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of ICDE*, 2004.

[6] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record 30(3)*, 2001.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of USITS*, December 1997.

[8] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams – a new class of data management applications. In *Proc. of VLDB*, 2003.

[9] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. of ACM SIGMOD*, 1999.

[10] E. G. Coffman and P. J. Denning. *Operating systems theory.* Prentice-Hall, 1973.

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

[12] P. Denning and S. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3):191–198, 1972.

[13] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. of ACM STOC*, 2002.

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

[15] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of ACM STOC*, 2001.

[16] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of VLDB*, 2003.

[17] INET ATS, Inc. http://www.inetats.com/.

[18] S. Jin and A. Bestavros. Greedydual* Web caching algorithm: Exploiting the two sources of temporal locality in Web request streams. In *Proc. of Web Caching Workshop*, May 2000.

[19] S. Jin and A. Bestavros. Temporal Locality in Web Request Streams: Sources, Characteristics, and Caching Implications (Extended Abstract). In *Proceedings of Sigmetrics'2000: The ACM International Conference on Measurement and Modeling of Computer Systems*, 2000.

[20] S. Jin and A. Bestavros. Gismo: A generator of internet streaming media objects and workloads. In *ACM SIGMETRICS Performance Evaluation Review*, 2001.

[21] S. Jin and A. Bestavros. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams. *International Journal on Computer Communications*, 24(2):174–183, February 2001.

[22] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbouded streams. In *Proc. of ICDE*, 2003.

[23] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: An architectural status report. *IEEE Data Eng. Bulletin*, 2003.

[24] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft. Structural Analysis of Network Traffic Flows. In *Proceedings of ACM SIGMETRICS / Performance 2004*, New York, NY, June 2004.

[25] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proc. of ACM SIGMETRICS*, 1999.

[26] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of SIGMOD*, 2002.

[27] A. Mahanti. Web proxy workload characterization and modelling. Master's thesis, Department of Computer Science, University of Saskatchewan, September 1999.

[28] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[29] Northwest Weather Resource. http://www-k12.atmos. washington. edu/ k12/grayskies/.

[30] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proc. of ACM SIGMOD*, 1993.

[31] S. Sarvotham, R. Riedi, and R. Baraniuk. Connection-level Analysis and Modeling of Network Traffic. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, 2001.

[32] U. Srivastava and J. Widom. Memomry-limited exectuion of windowed stream joins. In *Proc. of VLDB*, 2004.

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

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

[35] J. Xie, J. Yang, and Y. Chen. On joining and caching stochastic streams. Technical report, Duke University, 2003.

[36] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

[37] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of VLDB*, 2002.