Characterizing and Exploiting Reference Locality in Data Stream Applications

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

Abstract

In this paper, we investigate a new approach to process queries in data stream applications. We show that reference locality characteristics of data streams could be exploited in the design of superior and flexible data stream query processing techniques. We identify two different causes of reference locality: popularity over long time scales and temporal correlations over shorter time scales. An elegant mathematical model is shown to precisely quantify the degree of those sources of locality. Furthermore, we analyze the impact of locality-awareness on achievable performance gains over traditional algorithms on applications such as MAX-subset approximate sliding window join and approximate count estimation. In a comprehensive experimental study, we compare several existing algorithms against our locality-aware algorithms over a number of real datasets. The results validate the usefulness and efficiency of our approach.

1. Introduction

Stream database systems have attracted quite a bit of interest recently due to the mushrooming number of applications that require on-line data management on very fast changing data. Sample applications include network monitoring, sensor networks, and financial applications. The main difference between a traditional database and a data stream management system (DSMS) is that instead of relations, we have unbounded data streams and relational operations are applied over these streams [5].

Hence, 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 this 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 each data stream. However, there are still many cases where the full contents of the tuples of interest cannot be stored in memory. In these cases, it is desirable that the use of the available memory space be optimized so as to produce the best possible approximate results.

In this paper, we identify a very important property of many data streams that can be used in order to reduce the cost of stream operators and/or improve the quality of their results. First, we observe that many problems in data stream query processing with memory constraints can be casted as caching (or buffering) problems. An example is approximate sliding window join [12, 27, 29]. Furthermore, it is well known that locality of reference properties are important determinants of caching performance—specifically the potential for a small cache size to incur a very high hit rate. Denning and Schwartz [14] established the fundamental properties that characterize locality (and hence a baseline for achievable hit rates) in memory access patterns. Similar characterizations have also been documented for many other "reference" streams, most notably for Web access [4].

Therefore, we believe that many problems in data stream query processing can be solved more efficiently if the system can detect that the input streams exhibit locality of reference. In that respect, we identify two different causes of reference locality (popularity over long time scales and temporal correlations over shorter time scales). An elegant mathematical model is shown to precisely quantify the degree of those sources of locality in many real data streams. The advantage of the proposed model is that it is very simple and easy to estimate. At the same time, it captures some very essential properties that can be used to improve query processing in data stream systems and applications. We show how we can achieve that in the case of approxi-

0 This work was supported in part by NSF grants IIS-0133825, IIS-0308213, CNS-0524477, and CNS-0205294.

mate sliding window joins. Also, we demonstrate how the knowledge of reference locality can be used in order to reduce the cost of some stream algorithms, such as approximate frequency estimation.

The rest of the paper is organized as follows: In section 2, we characterize the two causes of reference locality in data streams and then provide an elegant mathematical model in section 3 to quantify the degree of different contributors to the reference locality. We demonstrate the importance of locality awareness in designing algorithms for three specific applications in section 4 where significant performance gains are shown theoretically. Lastly, we present an empirical study in section 5 on real data sets to validate our analytical results.

2. Reference Locality: Sources and Metrics

A *data stream* is said to exhibit reference locality if recently appearing tuples have a high probability of appearing again in the near future. One approach to *measure* reference locality in a stream is to characterize the **inter-arrival distance** (IAD) of references.

Definition 1. *We define the **inter-arrival distance** for a given data stream as a random variable that corresponds to the number of tuples separating consecutive appearances of the same tuple in the data stream.*

In the above definition and in the rest of the paper, we use the term "tuple" to refer to the value(s) of the attribute(s) related to a particular operation (e.g., if the operation is a stream join, then we refer to the values of the joining attributes). Therefore, when we say that two tuples are the "same", we mean that the attributes of interest have the same values. Mathematically, if x_n is the tuple at the position n, we define $d_i(k)$ as follows:

$$d_i(k) = Prob(x_{n+k} = i, x_{n+j} \neq i, for\, j = 1 \ldots k-1 | x_n = i)$$

where $d_i(k)$ is the probability that the tuple appears at time n with value i will be referenced again after k tuples. Let *d(k)* denote the average over all tuples, i.e. $d(k) = \sum_i p_i d_i(k)$, where p_i is the frequency (*i.e.*, popularity) of tuple i in this data stream.

While IAD characterization enables us to measure the degree of reference locality that exists in a data stream, it is not able to delineate the causes of such locality—namely, whether it is due to popularity over long time scales or correlations over short time scales. One way of quantifying the causes of reference locality is to compare the IAD distribution for a reference stream with that of a randomly permuted version of that same stream. By randomly permuting the reference stream, we effectively force that stream to follow the Independent Reference Model (IRM), or equivalently, the independent, identically-distributed (IID) tuple arrival model. In that case, $d(k)$ could be calculated as follows:

$$d(k) = \sum_{i=1}^{N} p_i d_i(k) = \sum_{i=1}^{N} (p_i)^2 (1-p_i)^{k-1} \qquad (1)$$

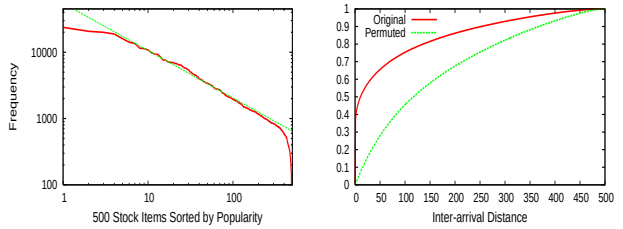
Notice that, by definition, IRM only captures locality due to a skewed popularity profile. Specifically, if N is the total number of unique tuples and all tuples are equally popular (*i.e.*, $p_i = \frac{1}{N}$), then $d_i(k) = d(k) = \frac{1}{N}(1-\frac{1}{N})^{k-1}$. For large N, this is independent of k ($\approx \frac{1}{N}$). On the other hand, when the tuple space exhibits a highly skewed popularity profile the above is not true. Actually, we can prove the following in the case of Zipf popularity distribution:

Theorem 1. *If the popularity distribution of tuples in a data stream follows a Zipf distribution with parameter $z = 1$, then the IAD distribution in a random permutation of this data stream can be characterized by $d(k) \sim \frac{1}{k}$.*

The above discussion leads us to the following two concise statements: (1) Locality in a reference stream due to long-term popularity is captured well by the IAD distribution of a random permutation of that stream, and (2) Locality in a reference stream due to short-term correlations is captured well by the difference between the IAD distribution of the stream and that of a randomly permuted version thereof.

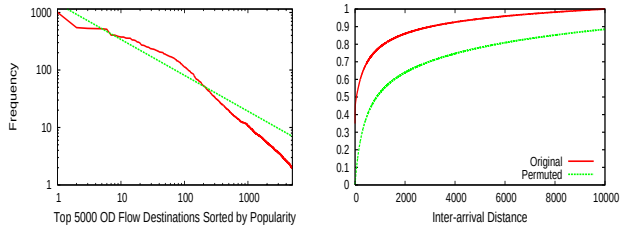
To validate this in real streaming datasets, we obtained traces from a number of different sources and computed the distribution of the popularity of different values as well as the CDF [1] of the IAD for the original and a randomly permuted instance of the trace. Next, we show the results for two of these traces: a stock dataset that was created using multiple days of real stock transaction data from [20] and a large network dataset that consists of traces of Origin-Destination (OD) flows in two major networks (US Abilene and Sprint-Europe) provided to us by the authors of [22]. The stock dataset contained about a million records per day, where the network dataset was in the order of hundred of millions of OD flow records. In the graph that we show here we used the trace collected at a particular Abilene router and consists of 76 millions OD flow records. The attribute of interest for the stock dataset was the stock name and for the network dataset was the destination address.

1 CDF: Cumulative Distribution Function.



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

(b) CDF of IAD for Original and Randomly Permuted Stock Trading Trace



(c) Zipf-Like Popularity Profile for OD Flow Traces

(d) CDF of IAD for Original and Randomly Permuted OD Flow Traces

Figure 1: Causes of reference locality in real streams

Figure 1(a) shows the highly-skewed popularity profile of stocks in the trace for one of the days. The relationship between frequency and rank (a straight line on a log-log scale) is consistent with a Zipf-like distribution. Figure 1(b) shows the CDF of the IAD for the stock trading trace, and for a randomly permuted version of that trace. The two distributions are quite different where more than 40% of the original trace exhibits short inter-arrival distance. This indicates that a significant source of reference locality is due to correlations over short time scales—correlations that are not captured using long-term frequency measures, or IRM assumptions. The same is true for the network dataset as it is shown in Figures 1(c) and 1(d). Similar results were obtained for other attributes in the network dataset and for other *real* datasets that include web traces, weather, and sensory data.

3. Locality-Aware Stream Model

In this section we present our first contribution: an elegant and succinct mathematical model that captures the reference locality caused by popularity over long time scales and correlations over shorter time scales.

3.1. Model Definition and Computation

Consider a data stream S and let P be the probability distribution of the popularity over long time scales of tuples in S. Let x_n be the tuple that appears at position n in S.

The model we propose mirrors our hypothesis that locality of reference is the result of *some* weighted superimposition of two very different causes: one that produces references over a domain with a skewed popularity profile P, and one that produces references that appeared in the most recent h units of time, *i.e.*, at positions *n-1, n-2, . . . , n-h*. Specifically, we propose:

$$x_n = \begin{cases} x_{n-i} & \text{with probability } a_i; \\ y & \text{with probability } b. \end{cases}$$

where $1 \leq i \leq h$ and $b + \sum_{i=1}^{h} a_i = 1$. y is a random variable that is independent and identically distributed according to P. Now, for a given tuple c, the probability that it will appear at position n is given by,

$$Prob(x_n = c | x_{n-1}, \ldots, x_{n-h}) = bP(c) + \sum_{j=1}^{h} a_j \delta(x_{n-j}, c)$$

where $\delta(x_k, c) = 1$ if $x_k = c$, and it is *0* otherwise.

Therefore, the model $\mathcal{M}$ can be described by two components, the locality parameters $\vec{a} = (a_1, a_2, \ldots, a_h)$ and the popularity distribution P. In this model, the value of b could be seen as a determinant of the extent to which the locality of reference is due to an IID sampling from some popularity distribution P, whereas the value of $1 - b$ could be seen as a determinant of the extent to which locality of reference is due to temporal correlations of references within some short time frame–namely h. Our empirical characterization of real data sets has demonstrated the existence of both factors. This model represents a more complete dynamics of the data stream and the conventional IID assumption can be reduced to our model by assigning $b = 1$.

Given a history of the data stream S, it is easy to infer P from the empirical distribution using histograms [18, 16]. The value of h is related to the degree of reference locality of the data stream and should be set according to the application. The most interesting part is to estimate a_i. Our approach is to use the least-squares method [25]. Suppose the data stream S has been observed for the past N tuples. Given a segment of length h in S, the expected value for x_n can be estimated using our model as: $\hat{x_n} = \sum_{j=1}^{h} a_j x_{n-j} + b \sum_{i=1}^{D} iP(i)$, where we assume tuples are taking integer values in the range of $[1, D]$. Therefore, we should find the parame-

ters that minimize the total error of the model:

$$\text{minimize over } a_1, \ldots, a_h, b: \sum_{i=h+1}^{N} [x_i - \overset{\frown}{x_i})]^2 \qquad (2)$$

By definition, $b = 1 - \sum_{i=1}^{h} a_i$. Let $a_{h+1} = b$ and $Y = \sum_{i=1}^{D} iP(i)$. $\overset{\frown}{x_i}$ is the prediction of x_i based on the model. Taking the derivative of equation 2 with respect to parameters a_k, we get:

$$0 = \frac{\partial\{\sum_{i=h+1}^{N}[x_i - \overset{\frown}{x_i})]^2\}}{\partial a_k}, k = 1, \ldots, h+1$$

This in turn yields the following $h+1$ equations:

$$\begin{cases} \sum_{i=h+1}^{N} 2[x_i - \overset{\frown}{x_i}]x_{i-k} = 0 & \text{k=1,...,h;} \\ \sum_{i=h+1}^{N} 2[x_i - \overset{\frown}{x_i}] \sum_{i=1}^{D} iP(i) = 0 & \text{k=h+1.} \end{cases}$$

Using the model definition and the observed actual values, we can rewrite the above equations as follows: (the first one represents h equations where $j = 1, \ldots, h$)

$$\begin{cases} \sum_{k=1}^{h} a_k \sum_{i=h+1-j}^{N-j} x_i x_{i+j-k} + bY \sum_{i=h+1-j}^{N-j} x_i = \sum_{i=h+1-j}^{N-j} x_i x_{i+j} \\ \sum_{k=1}^{h} a_k \sum_{i=h+1-k}^{N-k} x_i + bY(N-h) = \sum_{i=h+1}^{N} x_i \end{cases}$$

Thus, we have $h+1$ independent equations and we can solve for the $h+1$ variables, hence we estimate a_i, b. The next lemma bounds the space and time complexity of inferring the parameters of $a_1, \ldots, a_h, b$ for a data stream.

Lemma 1. *The space requirement for inferring the parameters of $\{a_1, \ldots, a_h\}$ for a data stream is $O(h^2)$ and the time complexity is $O(h^3)$.*

To construct the model, we monitor each stream for a short period of time and we update the coefficients of the $h+1$ linear equations. After observing N data tuples, we use a linear solver to estimate the parameters of the locality based model M for each data stream. The total space needed to compute the model is $O(h^2)$ and the running time of the solver is $O(h^3)$.

To validate our model, we test it against both the stock data traces and network OD flow traces. We plotted the relationship between b and h. The results are shown in figure 2. In both cases, for the original data streams, as the value of h (the characteristic time scale of short-term correlations) increases, the value of b converges to 0.1. In the randomly permuted version of the same

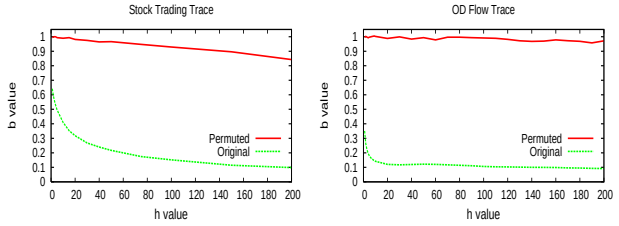


Figure 2: Applying the Locality Model to Stock and Network Data Streams

stream, the value of b stays very close to *1*. Notice that the rapid decrease in the value of b as h increases suggests that the impact of short-term correlations is most pronounced by considering only a short period of time. These results demonstrate the correctness of our model: it successfully captures two causes of reference locality in different data streams and is able to provide the concise measurements for the degree of two contributors. Based on this model, we can set the h value to the extent of reference locality we wish to capture. For example, from figure 2, we can see that $h \sim 50-80$ is a sufficient setting for these datasets.

3.2. Utilizing the Model for Prediction

Suppose that the model $\mathcal{M}_S$=$\{\vec{a}, P\}$ for a given data stream S has been estimated. We would like to use the model to predict the future tuples of the stream. Since the model is probabilistic, we can use a Monte Carlo approach and estimate the probability that a given tuple will appear in the next position of the stream. However, this approach can be expensive, since we need to sample from a large space of possible future values and then take avergares. Next, we discuss a simpler method that pre-computes parts of the computation off-line and makes the prediction easier.

Given a prediction window T and stream S, we define the locality characteristic matrix $M_{(\mathsf{T}+1)\times(\mathsf{T}+1)}$ for S as follows:

$$M = \begin{bmatrix} a_1 \cdots a_h & \vec{0}_{(\mathsf{T}-1-h)\times 1} & 0 & b \\ \vec{I}_{(\mathsf{T}-1)\times(\mathsf{T}-1)} & & \vec{0}_{1\times(\mathsf{T}-1)} & \vec{0}_{1\times(\mathsf{T}-1)} \\ \vec{0}_{(\mathsf{T}-1)\times 1} & & 0 & 1 \end{bmatrix} \qquad (3)$$

Let vector $\vec{X}_n$ denote $[x_n, x_{n-1}, \ldots, x_{n-\mathsf{T}+1}, Y]^T$. It is easy to check that: $\vec{X}_n = M \times \vec{X}_{n-1}$. By applying this equation recursively, we get:

$$\vec{X}_{n+\mathsf{T}-1} = M^{\mathsf{T}} \times \vec{X}_{n-1} \qquad (4)$$

Therefore, the expected values for tuples in a future time period T could be obtained by multiplying M^{T} with the last T tuples in the data stream. Given the model, M^{T} can be computed by simple matrix multiplications. Let c_{ij} represent an entry in M^{T} where $i, j \in [1, \mathsf{T}+1]$. The following lemma summarizes the expected number of occurrences for a given tuple during a period of length T.

Lemma 2. *The expected number of occurrences $E_{\mathsf{T}}(x_{n-j})$ for a tuple x_{n-j} in a future period of T tuples is given by the following formula:*

$$E_{\mathsf{T}}(x_{n-j}) = \sum_{i=1}^{T} c_{ij} + \sum_{i=1}^{T} c_{i(\mathsf{T}+1)} * P(x_{n-j})$$

Now, consider a tuple e. Using Lemma 2, we can compute $E_{\mathsf{T}}(e)$ as follows:

Lemma 3. *Given the operator $\mathcal{S}$ over vector $\vec{x}_{n-1}$ and e as $\mathcal{S}(\vec{x}_{n-1}, e) = \{k, k \in [1, T], s.t. \;\; x_{n-k} = e\}$, then $E_T(e)$ is:*

$$\sum_{k \in \mathcal{S}(\vec{x}_{n-1}, x)} E_{\mathsf{T}}(x_{n-k}) = \sum_{k} \sum_{i=1}^{T} c_{ik} + \sum_{i=1}^{T} c_{i(\mathsf{T}+1)} * P(e)$$

4. Application of the Locality-Aware Model

In this section, we show the applicability of our locality-aware model for three very important stream applications: (1) approximate sliding window joins, (2) approximate frequency counting and (3) information measures for data streams.

4.1. Approximate Sliding Window Join

The sliding window join is 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+W. r expires beyond time t+W and can be safely removed from the memory. The window W can be defined as time based, tuple based, or landmark based, and each stream may have a different sliding window size. When the memory size is insufficient to store the entire sliding window, buffer management optimization is essential to produce the best possible approximate result [12, 27, 29]. Previous work mainly focus on the *MAX-subset* metric where the goal is to produce the largest subset of the exact answer. This paper focuses on this metric as well, however our discussion can easily extend to other metrics.

Previous Approaches. The simplest approach to computing sliding window stream joins with limited memory is to use *random load shedding*, which drops tuples with a rate proportional to the rate of the stream over the size of the buffer [21]. A better approach is to adopt a *model* for the stream and drop tuples selectively using that model. This approach is called *semantic* load shedding. The first model-based approach is the frequency-based stream model [12], which makes an IID assumption for the tuples in the data streams and hence drops tuples based on their popularity; this algorithm is known as *PROB Heuristic*. Two other model-based approaches were proposed recently: the age-based data stream model [27] and the stochastic model [29].

In the age-based model, every tuple in the data stream is confined to follow a special aging process defined such that the expected join multiplicity of a tuple is independent of the value of the join attribute and dependent *only* on the arrival time. This model is appropriate for only a specific set of applications, namely on-line auctions, which were shown to produce data streams that follow the proposed model. Moreover, even for such applications, the assumption that all tuples follow the *same* aging process can be very limiting. We argue that in many real data streams the above model is unlikely to hold, making its usefulness quite limited. Indeed, in all our real datasets, the performance of this *aging* heuristic was indistinguishable from the simple random load shedding [9].

In the stochastic model, the system is assumed to observe the stochastic properties of the data streams and makes cache replacement decisions by exploiting the statistical information of these properties. The main problem of this approach is the complexity for representing and computing the model on-line. That makes this approach to have limited practicality and applicability. In contrary, our model is much simpler, easy to estimate, and likely to hold (albeit with varying parameterization) independently of the application or processes underlying the data streams. Furthermore, the model we advocate leverages a fundamental property of data streams—namely locality of reference—that can be used to improve data stream query processing as we will see next.

Locality-Aware Approach. Consider two streams R and S and the sliding window join $R[W] \bowtie_A S[W]$ (i.e. equi-join on a common attribute A). We assume equal window size W, a constant arrival rates *1* for all streams, and discrete values for the joining attribute A. Our discussion can be extended to varying arrival rates, window sizes, and other join conditions as shown in the experiments. Also, we assume that the memory is full and new tuples are arriving from both streams.

Therefore, we must have a method to chose some of the tuples already in memory and replace them with the new tuples (or decide to drop some of the new tuples).

We define the marginal utility of a tuple x in stream R to represent the expected number of results produced by this tuple by joining it with the *other* stream S over the time period $[n, n+\mathsf{T}]$, i.e. the number of tuples in the other stream X_k^S that satisfy the join condition during the next T time instants. Formally, we represent marginal utility for a tuple X as a function U(X) as follows:

Definition 2. *Given a tuple* X=x *in stream* R *and a time period* T*, the* marginal utility *of* X *at time* n *is:* $U_{X^R}^n(\mathsf{T}) = \sum_{k=n}^{n+\mathsf{T}} P\{X_k^S = x\}$

In the above definition, T is the remaining lifetime of tuple x. If we had no memory constraints, the lifetime of a tuple would be equal to W. However, due to the space constraint, tuples may be evicted before they expire. The choice of eviction depends on many factors (such as buffer size constraints, competing tuples in the same stream, and correlated marginal utility with tuples in the other streams) that complicates the analysis and makes it hard to predict the actual lifetime of a tuple. [29] has given a detailed discussion on this issue and suggested heuristics to approximate tuples lifetime. However, for the simplicity of our analysis, we assume a constant value T as the global lifetime for each tuple.

Using the model proposed in Section 3, we can estimate the expected marginal utility of any tuple x in the data stream R and thereupon make replacement decisions. Suppose that there are k tuples in stream S that have the same value with x during $[n-h, n-1]$. Let those tuples be $x_{c_1}, \ldots, x_{c_k}$. The probability that the next tuple arriving at S has the same join attribute with x is $P_1 = bP(x) + \sum_{j=1}^{k} a_{n-c_j}$. Recursively applying the model in a similar fashion, we get that the probability that x will appear at the time i, $1 \le i \le \mathsf{T}$, is:

$$P_i = bP(x) + \sum_{j=1}^{k} a_{\Theta(n+i,c_j)} + \sum_{j=1}^{min(h,i-1)} P_{i-j}a_j$$

$\Theta(n+i, c_j)$ is an indicator function which equals to $n+i-c_j$ if $n+i-c_j \le h$ (i.e. within the h range) and *0* otherwise. Also $a_0 = 0$. P_i is an h order linear recursive sequence (that depends on $P(x), a_1, \ldots, a_h, b$). We can solve the hth order characteristic equation to get P_i. The marginal utility of x in stream R is then calculated as:

$$U_{x^R}^t(\mathsf{T}) = \sum_{i=1}^{\mathsf{T}} P_i = \mathsf{T}bP(x) + F(a_1, \ldots, a_h, b, P(x))$$

Algorithm 1 LBA(Stream S,R; Predict Window T)

```
(M, P) = Warmup(S)
Compute characteristic matrices M and M^T
c⃗s = column sum of M^T
Insert P and c⃗s into histogram H
while New tuple arrives in R at time instance t do
    if R's Buffer is full then
        For tuples in R's Buffer apply lemma 3 to compute
            U^t_{x^R}, get necessary column sum and P(x) from
            H
        Evict tuple with minimum U^t_{x^R}(T)
        Insert the new tuple into the buffer
```

This formula shows that the marginal utility of a tuple depends on both its long-time popularity and short-term correlations. When T is large enough to span multiple locality epochs, $P(x)$ dominates the marginal utility which complies with the frequency-based model. In that case, $P(x)$ alone can constitute a good approximation of $U_{x^R}^t$. However, if T is small enough to embody dispersed locality, *both* terms govern the stream, and both should be considered in order to make an informed caching decision. We denote this way of calculating marginal utility and hence making caching decisions as exact locality based algorithm *(***ELBA***)*.

To estimate the marginal utility with ELBA, we have to perform $O(B\mathsf{T})$ operations for every eviction, where B is the buffer size, since we have to recompute P_i for every tuple in the cache. However, given that T is fixed [2], we present a more efficient approximation of the marginal utility of each tuple. Recall that from lemma 3, given the sum of the columns of M^{T} of S, $\sum_{i=1}^{\mathsf{T}} c_{ij}, for\ j \in [1, \mathsf{T}+1]$, and the popularity distribution P of S, we could compute the marginal utility for any tuple in R at any given time instant. Based on this observation, we present a simpler locality-aware buffer replacement algorithm *(***LBA***)* for MAX-subset optimization. The pseudo-code is shown in Algorithm 1. Note that when more than one tuples have the same minimum marginal utility, the tie is broken by choosing the tuple that will expire sooner.

Next, we discuss the complexity of *LBA* against other existing algorithms. First, notice that both *PROB* and *LBA* construct similar storage structures (e.g. histogram) to gather their statistical information during the initial setup. Given a histogram of N items, we define the space usage of such storage structure as $f(N)$ [3] and the time complexity for querying the structure as Q. The space complexity of *PROB* is $O(f(D))$ where D is the domain size of the stream and it is $O(f(D+\mathsf{T}))$ for

2 In our experiments we set $\mathsf{T} = W$.

3 Recent work, for example [16], has shown that f can be poly(log(N)).

LBA as it requires additional storage for the each column sum of the matrix M^{T}. Normally $D > T$ (or even $D >> T$) and f is a logarithmic function, so the space overhead of PROB and LBA is comparable. In terms of time complexity, the operations of interest in our analysis are insertions and evictions. When a new tuple arrives at the stream, it takes *LBA* constant time to insert it into the memory and a time complexity of $O(B * Q)$ [4] to select a tuple to evict. *PROB* incurs similar time complexity if implemented in the same manner. However, another approach *PROB* could adopt is to maintain the buffer as a priority queue and associate each tuple its popularity value computed on the fly during the insertion phrase. This method has an insertion complexity of $O(Q + logB)$ and a constant time eviction. Finally, for the other existing methods, the stochastic model is at least as expensive as the PROB approach and in practice is expected to be much more expensive.

4.2. Approximate Count Estimation

Lossy Counting [24] is a well-known algorithm for approximate frequency counting over data streams. In this algorithm (shown as Algorithm 2), the incoming data stream is divided into buckets with width $w = \frac{1}{\varepsilon}$. Buckets are labeled with bucket IDs, starting from 1. The current bucket is denoted as $b_{current}$ and its value is $\lceil \frac{N}{\varepsilon} \rceil$ (where N is the current length of the data stream). For an element e, its true frequency is denoted as f_e. The algorithm keeps a data structure $\mathcal{D}$ that contains entries of the form $(e, f, \triangle)$, where e is an element in the stream, f is an integer representing its estimated frequency, and $\triangle$ is the maximum possible error in f. [24] proved that the space bound for Lossy Counting is $\frac{1}{\varepsilon} log(\varepsilon N)$. This is a very general bound as it makes no assumptions about the data stream (popularity distribution or temporal correlations). However it is also a very loose bound (as already shown in [24]) and furthermore, not a very useful one as N is included in it. Next, we provide a better estimation for the space requirement for Lossy Counting taking into account the reference locality that may exist in the stream.

First, we give a general formula to estimate the space requirements of Lossy Counting. For an element e, let p_e be the probability that e will be the next element in the stream. Let $B = b_{current}$ be the current bucket id. For $i = 2, ..., B$, let d_i denote the number of entries in $\mathcal{D}$ with $\triangle = B - i$. Consider the

4 The actual time complexity should be O(b*Q) where $b \leq B$ represents the number of tuples having unique join attributes in the buffer. With skewed distribution, $b << B$.

Algorithm 2 LossyCounting(Stream S; Error ε; Frequency s; Structure $\mathcal{D}$)

```
Initialize N=0 w = 1/ε
while new tuple s arrives in S do
    if s is in D then
        Increase its frequency by one
    else
        Insert a new entry (e, 1, b_current − 1) to D
    Increase N by one
    if N%w == 0 then
        Delete entries in D if f + △ ≤ b_current
    if require outputs then
        Outputs entries from D if f ≥ (s − ε)N
```

case that e contributes to d_i. The arrival of the remaining elements in the buckets $B - i + 1$ through B can be viewed as a sequence of Bernoulli trials, where each trial succeeds with probability p_e and fails with probability $1 - p_e$. If the random variable X denotes the number of successful trials, then X follows the binomial distribution. Thus, given N trials, the probability to have exactly n successful trials is calculated as: $P_X(n|N) = \binom{N}{n} p_e^n (1 - p_e)^{N-n}$. In this case $N = w(i - 1)$. In order for e to contribute to d_i, it requires $X \geq i - 1$ (otherwise it will be deleted at some point). For any given bucket, the expected number of appearances of e is given by $p_e w = \frac{p_e}{\varepsilon}$. So the expected value for d_i is:

$$E(d_i) = \sum_e \frac{p_e}{\varepsilon} \sum_{k=i-1}^{N=\frac{i-1}{\varepsilon}} \binom{N}{k} p_e^k (1 - p_e)^{N-k} \qquad (5)$$

and the overall space requirement is simply given by $\sum_{i=2}^{B} E(d_i)$. The only question left is how to compute p_e. If we assume IID, then p_e is simply the popularity of element e in the long term popularity distribution of this data stream. However, when the data stream exhibits reference locality due to short time correlations, we should compute p_e based on the analysis of the reference locality of this data stream. Using our locality model and following the discussion in section 3.1, we could compute $Pr(x_N = e) = \sum_{i=1}^{h} a_i \delta(x_{N-i}, e) + bP(e)$.

Assuming IID and that the values in the stream come from a general fixed distribution, [24] showed that the expected size of the data structure is $\frac{7}{\varepsilon}$. However, both in their and our experiments, this space bound is still very pessimistic, especially for skewed distributions (Zipf-like). This indicates that for data stream with higher degree of reference locality, either caused by skewness in long term popularity or correlations in short time scales, tighter space bound might exist.

Effect of Reference Locality by Skewness. When the popularity of the values appearing in the stream

follows a skewed distribution, the space bound can be improved. Here, we discuss the case of Zipf distribution, since it is a very common distribution in practice. We have shown already in section 2 that the Zipf distribution creates reference locality in the stream.

Formally, a Zipf distribution with parameter z has the property that f_i, the relative frequency of the ith most frequent item is given by $f_i = \frac{c_z}{i^z}$, where c_z is an appropriate scaling constant. Consider a Zipf distribution with range $[1 \dots U]$ and $1 < z \leq 2$. The following observations were shown in [11]:

- Fact 1. $1 - \frac{1}{z} \leq c_z \leq z - 1$
- Fact 2. $\frac{c_z k^{1-z}}{z-1} \leq \sum_{i=k}^{U} f_i \leq \frac{c_z (k-1)^{1-z}}{z-1}$

Based on the previous results, we can show the following:

Theorem 2. *For Lossy Counting, if stream elements are drawn independently from a Zipf distribution with $1 < z \leq 2$, then $E[|\mathcal{D}|] < \frac{2}{\varepsilon} + \frac{4}{2^{z-1}\varepsilon^{2-z}}$.*

Proof. Our proof follows the one in [24]. Separate the elements in the stream into two parts. The first part contains elements with $p_e \geq \frac{\varepsilon}{2}$. There are at most $k = \frac{2}{\varepsilon}$ number of such elements. The second part contains elements with $p_e < \frac{\varepsilon}{2}$. Following the analysis in last section, let X denote the number of successful trials of an element e from the first part. Then, we have that $E(X) = p_e N = \frac{p_e(i-1)}{\varepsilon} < \frac{i-1}{2}$. Using Chernoff bounds, we can show that $Pr[X \geq i-1] \leq (\frac{e}{4})^{\frac{i}{2}}$(already shown in [24]). Thus, by equation 5,

$$E(d_i) \leq (\frac{e}{4})^{\frac{i}{2}} \sum_{e} \frac{p_e}{\varepsilon} = (\frac{e}{4})^{\frac{i}{2}} \sum_{j=k}^{U} \frac{p_j}{\varepsilon}$$

Using Facts 1 and 2 above, we get

$$E(d_i) \leq (\frac{e}{4})^{\frac{i}{2}} \frac{1}{\varepsilon} (k-1)^{1-z} \approx (\frac{e}{4})^{\frac{i}{2}} \frac{1}{\varepsilon} k^{1-z}$$

The overall space contributed by the second part is:

$$\sum_{i=2}^{B} E(d_i) \leq \frac{1}{\varepsilon} k^{1-z} \sum_{i=2}^{B} (\frac{e}{4})^{\frac{i}{2}} \leq \frac{1}{\varepsilon} k^{1-z} \frac{e/4}{1 - \sqrt{e/4}} \leq \frac{4}{\varepsilon} k^{1-z}$$

Since $k = \frac{2}{\varepsilon}$ and $2 \geq z > 1$, we have $\sum_{i=2}^{B} E(d_i) \leq \frac{4}{2^{z-1}\varepsilon^{2-z}}$. This completes the proof. □

To compare the new bound against the previous one ($\frac{7}{\varepsilon}$), we list their values for different Zipf distributions (varying z) and $\varepsilon = 0.1\%$ in table 1. As we can see, our bound is much tighter than the previous one. Furthermore, it is able to capture the important fact that the skewer the distribution is, the less the space is used by Lossy Counting. In fact, when $z = 1.5$, the second term in our bound is only 89 and its value is mainly dominated by $\frac{2}{\varepsilon}$.

z value	old bound	our bound
$z = 1.1$	7000	3869
$z = 1.3$	7000	2409
$z = 1.5$	7000	2089

Table 1: Space Bound Comparison With $\varepsilon = 0.1\%$

Algorithm 3 MTOEPSILON(Error ε; Frequency s; Memory B; Threshold b)

```
Initialize ε = s, e1 = s, e2 = 0, B' = 0
while |B − B'| > b do
    B' = EpsilontoM(ε)
    if then B' < B
        ε = ε − (e1 − e2)/2, e1 = ε
    else
        e2 = ε, ε = ε + (e1 − e2)/2, e1 = ε
```

Effect of Reference Locality by Both Causes. Based on the locality model, we can derive an even tighter space bound for Lossy Counting. Using lemma 3 in section 3.2, we can compute the accumulative expectation for an element e to appear in a future period of length T. To calculate the expected space usage for the first bucket, we set $T = w$ and count the number of elements who have an accumulative expectation greater than one. To extend this idea to more buckets, we compute the accumulative expectation of each element e by adjusting T to a greater value. Using the characteristic matrix $M_{(w+1)\times(w+1)}$, we generate $M^w, M^{2w}, \dots, M^{kw}$ and define $U^i = M^{(i+1)w} - M^{iw}$, $for\ i \in [0, k-1]$. We can see that U^i summarizes the expectation in the range of $[iw, (i+1)w]$. Thus, if we apply lemma 3 with U^i, we get the accumulative expectation of any element in the ith bucket. Then for each bucket, we can count the number of elements that are expected to appear at least once and occupy a space in that bucket which give us the estimated maximum space bound. Note that, k is an input parameter to be decided by the system. Assuming that the reference locality properties of the data stream are stationary or changing slowly, even a small value for k is sufficient.

Practical Implications. Given the previous discussion, if we have the reference locality model for a data stream and a given ε, we can accurately estimate the expected space that will be occupied by the Lossy Counting algorithm and therefore, optimize memory allocation. On the other hand, given a limit on the available memory M, we can find the smallest ε that must be used with Lossy Counting in order to stay inside

the memory limit (see Algorithm 3). The main advantage of this, is that smaller ε means smaller number of false positives and therefore better estimation of the frequencies.

4.3. Information Measures of Data Streams

Many data stream applications require summarization of some stream characteristics. However, despite the extensive work in this area, it seems that still there is consensus on a general metric to measure the amount of information existing in a data stream. In this section, we propose a new approach to measure that using the notion of *entropy*. For a random variable X, the entropy is a well-known measure for the amount of information (or uncertainty) underlying knowledge of X. In particular, the entropy of X is defined as $H(X) = -\sum_i p_i log p_i$, where p_i is the (discrete) probability distribution of X. A straightforward approach to extend the notion of entropy to a data stream S, is to view the whole data stream as single random variable and generate each tuple in S by drawing samples from this random variable. Thus, the defined entropy is simply the measure of randomness in the long term popularity distribution of S. However, this definition fails to capture locality of reference characteristics that may exist in the data stream. To clarify, let S' be a randomly permuted version of S. Then, the entropy of S' is always equal to the entropy of S.

A more comprehensive approach is to view the data stream S as a vector of N random variables where N is the length of the stream, i.e., represent S as $S = [X_1, X_2, \ldots, X_N]$ where each X_i is a random variable. In this case, we define the entropy of the data stream as the average entropy for all these random variables.

Definition 3. *Given a data stream* S *represented as a vector of* N *random variables, its entropy is defined as the average entropy over all random variables, i.e.* $H(S) = \frac{1}{N}\sum_{i=1}^{N} H(X_i)$, *where* $H(X_i) = H(X_i|X_{i-1}, X_{i-2}, \ldots, X_1)$ *is the corresponding entropy of random variable* X_i *in* S.

For a data stream under the IID assumption, each random variable in S is identically distributed, as with a random variable X. In this case, $\forall i, H(X_i) = H(X)$ and $H(S) = H(X)$. However, this is not the case when the data stream exhibits a certain degree of reference locality due to short time correlations. Then, even though every tuple in S is still drawn from a fixed distribution represented by the random variable X, each one of them is not independent from the previous ones. Let us model the entropy of each tuple as a conditional variable that depends on k previous random variables. This leads to the expression $H(X_N) = H(X_N|X_{N-1}, \ldots, X_1) = H(X_N|X_{N-1}, \ldots, X_{N-k})$. Now, it is possible to compute $H(S)$ based on the locality model. The basic idea is to condition the entropy measure by obtaining the probability distribution for each random variable x_N given the locality model that we defined in 3.1. By defining the entropy in data stream in this way, we can show the next lemma, the proof of which follows from the well-known fact that the conditional entropy is always not greater than the unconditional entropy:

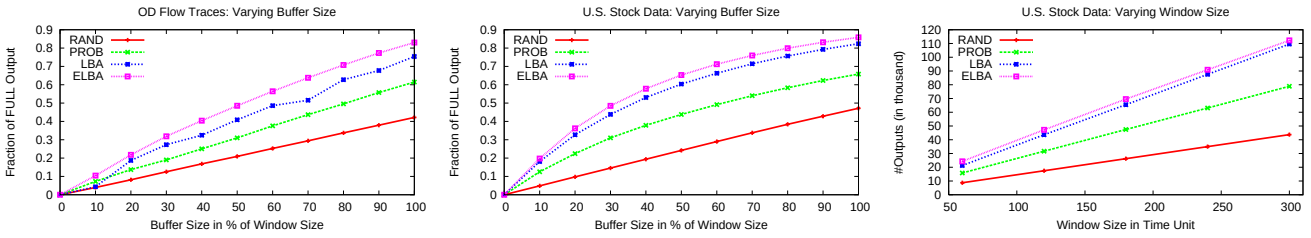
Lemma 4. *Let* S' *denotes the randomly permuted version of data stream* S. *Then* $H(S) \leq H(S')$ *where the equal sign only holds when* S *has no reference locality due to short time correlations.*

5. Performance Evaluation

In this section, we present results from an extensive experimental evaluation of the methods presented in the previous sections. The experiments were conducted using two real datasets: the stock data [20] and the OD flow streams [22]. Similar results were obtained from other real and synthetic datasets. Each stock transaction contains the stock name, time of sale, matched price, matched volume, and trading type. 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 network dataset was collected from Abilene routers, where the system sampled packets during the collection process (one out of every 100 packets is measured and recorded). However, it still exhibits a significant amount of reference locality as we show earlier. Each OD flow tuple contains a timestamp, source and destination address, number of packets, bytes transfer, and other information. We selected the destination address as the join attribute.

5.1. Approximate Sliding Window Joins

We implemented both the exact (ELBA) and approximate locality-based (LBA) algorithm. We compared LBA and ELBA against random eviction (RAND) and the frequency-based eviction (PROB) [12] approaches. We measured the quality of the results using the *MAX-subset* metric. In addition, as a baseline, we compute the exact results by supplying memory enough to contain the full windows (FULL). This allows us to present the results of the other methods as fractions of the exact results produced by FULL. Each algorithm is setup to utilize the same amount of memory. Our implementation on PROB and LBA demonstrate comparable



(a) Varying Buffer Size With OD Flow Traces

(b) Varying Buffer Size With Stock Data

(c) Varying Window Size With Stock Data

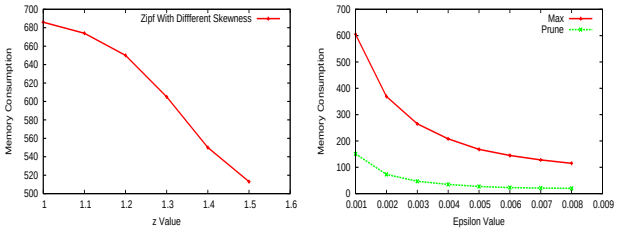
Figure 3: Varying Buffer Size and Window Size

running time performance. RAND executes relatively faster because it takes constant time for both deletion and insertion.

The first set of experiments focused on the effect of the buffer size as a percentage of the total memory requirement. Note that, the buffer size that we report in the experiments is calculated based on the *average* arrival rates as opposed to instantaneous rates. Since the arrival rates of our data streams are quite bursty, even when a 100% buffer size is supplied, the buffer cannot retain all tuples at times when the instantaneous rate is higher than the average rate.

Figure 3(a) and 3(b) show that the locality-aware algorithms outperform PROB and RAND in all of our experiments on both stock and OD flow data sets. This confirms our expectations from the previous analysis that PROB does not capitalize on locality of reference stemming from correlations over short timescales (Figure 1(b) and 1(d)). PROB assumes that the data stream follows the IRM and only considers long term popularity, but both our analysis and experiments indicate that reference locality contributes to data stream dynamics. Our results also show that the performance of LBA is quite close to that of ELBA, which reinforces our argument of LBA being a practical approximation of ELBA. Both algorithms outperform PROB by a large margin. For some specific buffer sizes, they produce close to 100% more result tuples than PROB.

To study the scalability of our algorithm, we execute the same set of experiments using a wide range of window sizes on a 50% buffer size. In Figure 3(c), we show the actual number of join results produced by different methods. Furthermore, the results as fractions to the FULL results are almost constant w.r.t different window sizes. The relative results for all algorithms are remarkably invariant over different window sizes. This is consistent with the observations and experimental results by Das et al [12]. Note that the results for OD flow are very similar and are omitted for brevity.



(a) Space vs. Skewness

(b) Space vs. Epsilon

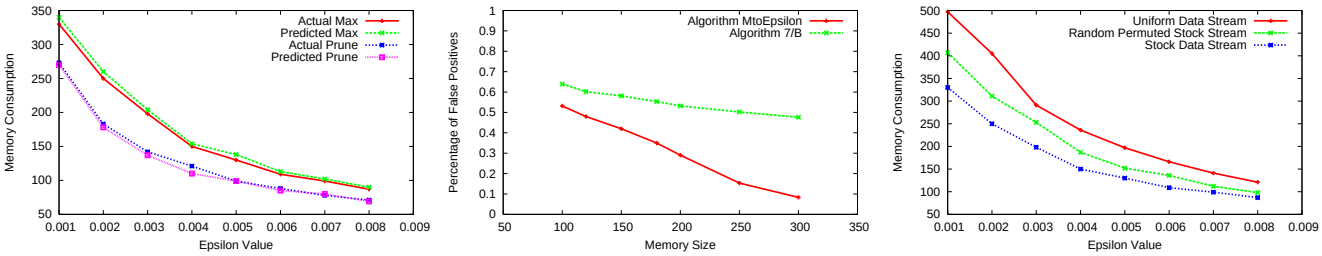
Figure 4: Space Bound Analysis With Reference Locality by Skewness

5.2. Approximate Count Estimation

Here we present results for Lossy Counting. Except otherwise specified, we run Lossy Counting with parameters $s = 0.01$ and $N = 100,000$. Also, in our figures, "Max" indicates the maximum memory consumption across all buckets at any given time. "Prune" represents the maximum memory usage after pruning at the end of every bucket.

First, we study the space bound for data streams with reference locality caused by skewness. We generated a set of data streams by varying the "z" value of the Zipf distribution and studied the effect of skewness on the memory overhead of Lossy Counting. We set the domain size to 3000 and $\varepsilon = 0.001$. For each data stream, we run the Lossy Counting algorithm and we computed the maximum space consumption. From figure 4(a), we observe that less memory is used as the distribution becomes skewer. Then, we fix "z" at 1.3 and vary the ε value as shown in figure 4(b). The results in both figures agrees with our upper bound analysis in Theorem 2.

Next, we present results on real data streams. We designed a simple algorithm to estimate the memory consumption of Lossy Counting using the locality based model of the data stream.Figure 5(a) shows the results



(a) Space Bound Analysis With Reference Locality by Both Causes

(b) Reducing the False Positive

(c) Entropy Vs. Space Consumption

Figure 5: Approximate Count Estimation

of our space bound analysis and the actual space consumption of Lossy Counting for the stock dataset and varying ε. Similar results obtained for OD flow data streams as well. As we can see, our algorithm produces bounds that tightly match the actual space consumption. This means that our model captures the essential properties of the stream to estimate very good bounds on the space consumption of Lossy Counting.

Finally, we investigated the effect of finding the optimal ε value given a fixed memory size. In this experiment we set $s = 0.05$ and used again the stock data stream. Given a fixed memory size, we used the algorithm 3 to estimate the ε value. Also, we used the previous best approach [24], that estimates $\varepsilon = \frac{7}{B}$ (where B is the memory size in number of entries of $\mathcal{D}$). The value of ε is directly related to the number of false positives produced by the algorithm. Therefore, we plot the percentage of false positives for different memory sizes. As we can see in Figure 5(b) our approach produces much smaller number of false positives, especially when the memory size increases.

5.3. Entropy

Finally, we performed a simple experiment to study our proposed approach to quantify the entropy of a data stream. We generated a data stream with uniformly distributed values and we computed its empirical locality based entropy. Furthermore, we computed the empirical entropy of the original stock data stream, and a randomly permuted stock data stream. The results are shown in table 2. Note that the entropy of the original stock data stream is much smaller that the permuted version. To show a practical implication of the data stream entropy measure, we run Lossy Counting on all three data streams. As we can observe from figure 5(c), the space consumption of Lossy Counting depends on the entropy of the stream. A stream with smaller entropy can be summarized with smaller space. We found similar results on experiments using the OD flow data streams. We believe that these results are very promising and we plan to investigate this issue further.

Data Streams	Entropy
Uniform IID	6.19
Permuted Stock Stream	5.48
Original Stock Stream	3.32

Table 2: Entropy of Different Data Streams

6. Related Work

Stream data management has received a lot of interest recently. General issues and architectures for stream processing systems are discussed in [1]. Continuous queries are studied in detail in [8, 23], where the authors proposed methods to provide the best possible query performance in continuously changing environments. Approaches to process sliding window joins and provide exact results are discussed in [17, 19].

Most of related works to the applications we studied in this paper have already been discussed. General load shedding is used in [28, 6]. In [28] a QoS utility based load shedding scheme is introduced which is similar to our methods discussed here. However, their scheme is not targeted to sliding window joins but to other Aurora operations. A load shedding approach for aggregation queries is presented in [6]. Approximate frequency count is also studied in the context of sliding window [13, 3], where the problem becomes much harder as one has to deal with deletions. We believe our discussion on the reference locality in data streams could extend to these applications as well. A recent approach to decrease the memory requirements of stream operators has been proposed in [7] that uses the notion of *k-contsraints*. This approach is similar to ours since it tries to exploit a form of short term temporal correlations. However, as a stream model is much weaker than ours, since it does not consider the correlations

over large time scales.

Model based approaches to improve query processing in sensor networks discussed in [15]. The idea is to exploit temporal and spatial correlations in sensor network generated data. Temporal locality in the context of sensor networks has been exploited also in [26]. The importance of skewness to data summarization has been studied in [11]. In particular, they showed how to improve the performance of Count-Min sketch [10] under skewed distributions. However, no connection between skewness and reference locality was made there.

7. Conclusion and Future Work

In this paper we propose a new approach to process queries in data stream applications. A generic locality model is discussed to identify and quantify the degree of two causes of reference locality in data streams. Based on this model, locality-aware algorithms for buffer management in sliding window joins, approximate frequency count, and data summarization are proposed. In support of our claims, we have presented experimental results using real traces, which confirmed the superiority of locality-aware algorithms over other existing algorithms. For our future work, we plan to apply locality-aware techniques discussed in this paper to other metrics and other operations in data stream management systems. It is also possible to extend our locality model by augmenting our parameters to synthesize and combine other temporal patterns and to include a more comprehensive multi-criteria analysis in a distributed setting.

References

[1] Data Stream Processing. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

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

[3] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, 2004.

[4] M. Arlitt and C. Williamson. Web server workload characteristics: The search for invariants. In *SIGMETRICS*, 1996.

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

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

[7] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM TODS*, 29(3):545–580, 2004.

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

[9] C. Chang, F. Li, A. Bestavros, and G. Kollios. GreedyDual-Join: Locality-aware buffer management for approximate join processing over data streams. Technical report, TR-2004-028 - Boston University, 2004.

[10] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *LATIN*, 2004.

[11] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

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

[13] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 2002.

[14] P. Denning and S. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3), 1972.

[15] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

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

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

[18] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *STOC*, 2001.

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

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

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

[22] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. Kolaczyk, and N. Taft. Structural Analysis of Network Traffic Flows. In *SIGMETRICS*, 2004.

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

[24] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2003.

[25] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C(second edition)*. Cambridge University Press, 2002.

[26] M. Sharaf, J. Beaver, A. Labrinidis, and P. Chrysanthis. TiNA: a scheme for temporal coherency-aware in-network aggregation. In *MobiDE*, 2003.

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

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

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