Approximate Join Processing Over Data Streams

Abhinandan Das
Cornell University
asdas@cs.cornell.edu

Johannes Gehrke[*]
Cornell University
johannes@cs.cornell.edu

Mirek Riedewald
Cornell University
mirek@cs.cornell.edu

ABSTRACT

We consider the problem of approximating sliding window joins over data streams in a data stream processing system with limited resources. In our model, we deal with resource constraints by shedding load in the form of dropping tuples from the data streams. We first discuss alternate architectural models for data stream join processing, and we survey suitable measures for the quality of an approximation of a set-valued query result. We then consider the number of generated result tuples as the quality measure, and we give optimal offline and fast online algorithms for it. In a thorough experimental study with synthetic and real data we show the efficacy of our solutions. For applications with demand for exact results we introduce a new *Archive-metric* which captures the amount of work needed to complete the join in case the streams are archived for later processing.

1. INTRODUCTION

In many applications from IP network management to telephone fraud detection, data arrives in high-speed *streams*, and queries over those streams need to be processed in an online fashion to enable real-time responses. Data streams pose a serious challenge for data management systems as the traditional DBMS paradigm of set-oriented processing of disk-resident tuples does not apply. Recently several new proposals for *data stream processing systems* have emerged [2, 6, 18]. These systems are specifically architected to process data streams in real time.

As for traditional relational database systems, the join operator is a very important operator in a data stream processing system. Let R and S be two data streams that contain a joint attribute A, which is selected as the join attribute. The equi-join of R and S is the subset of the cross-product of the two streams that contains exactly those pairs of tuples (r, s) such that $r \in R$, $s \in S$, and $r.A = s.A$.

While joins are very important, their computation is resource intensive. For instance, a standard equi-join carries conceptually unbounded state for two infinite input streams. To deal with the problem of unbounded state for data stream joins, the semantics of the join are usually changed to restrict the set of tuples that participate in the join to a bounded-size window of the most recent tuples [2]. Since the window conceptually slides over the input streams, this type of join is often called a *sliding window join*. Notice that there are several possibilities to define the window boundaries—based on time units, number of tuples, or landmarks. For simplicity we will assume that the window is defined in terms of time units (wall-clock time) and that at each time unit a new tuple arrives on each input stream. Our discussion and techniques can be generalized to windows defined in terms of the number of tuples and to asynchronous tuple arrival.

In the following we will use w to denote the window size. Let $r(i)$ refer to the tuple of stream R that arrives at time i. For simplicity we will also use $r(i)$ to denote the value of the tuple's join attribute ($s(i)$ is defined and used similarly). According to our model, at each time t the sliding window contains all tuples $r(i)$ and $s(i)$ with $t - w < i \leq t$.

The online nature of data streams and their potentially high arrival rates impose high resource requirements on data stream processing systems. Especially in applications where several queries are processed concurrently, the availability of resources that can be devoted to each query is limited and might vary over time. As another example, it is often impossible to estimate the peak tuple arrival rate for data streams, and thus sizing a data stream system for peak loads is a hard problem. Thus although changing the semantics of the join operator to sliding windows has already reduced the resource requirements of the join operator, computing sliding window joins can still exceed resource availability.

Resource limitations can have two effects. For streams with high arrival rates, the *CPU* might not be fast enough to process all incoming tuples in a timely manner, i.e., we have a *slow CPU* compared to the arrival rate of the stream. For large windows w the available *main memory* M might be too small to keep all relevant tuples in-memory (and frequent access to hard disk will be too slow when arrival rates are high).

In order to deal with resource limitations in a graceful way, returning approximate query answers instead of exact answers has emerged as a promising approach to save resources [4]. In data stream processing systems, one way of approximating query answers is to shed load, for example, by

[*]This work was supported by NSF Grants IIS-0084762, IIS-0121175, IIS-0133481, and CCR-0205452, the Cornell Information Assurance Institute, and by gifts from Microsoft and Intel.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

dropping tuples before they naturally expire (i.e., leave the window) or even before they reach the operator. The current state of the art consists of two main approaches. The first relies on random load shedding, i.e., tuples are removed based on arrival rates, but not their actual values [21]. The second proposes to include QoS specifications which assign priorities to tuples and then shed those with low priority first [6]. However, the result of a join consists of *pairs of matching tuples*, hence both the join attribute of a tuple and the number of its partner tuples (i.e., those that match the tuple) in the other stream determine the output. For this reason both random load shedding and simple QoS assignments to single tuples do not fully capture the semantics of the join.

For example, it is well known that random sampling from the inputs R and S of a join, or biased sampling from R and S without taking the distribution of the other relation into account, can greatly skew the output of the join, and lead in the worst case to an empty join output even though the actual size of the join is very large [8].

Semantic Load Shedding. In this paper, we address the problems outlined above by introducing the notion of *semantic load shedding*. In semantic load shedding, we approximate the output of an operator by maximizing a user-defined similarity measure between the exact answer and the (approximate) answer returned by the system. Semantic load shedding avoids the above problems by intelligently selecting which tuples to drop and when they should be dropped — all in order to minimize the error of the output of the query. This paper contains an in-depth study of this problem for the case of sliding window joins. Let us shortly discuss some related scenarios where semantic load shedding can lead to great improvements.

Static Join: Consider a network of small battery-powered sensors with limited CPU speed and memory which measure environmental data. Furthermore there are *sensor proxies* in the network that are not power constrained and have ample CPU and memory resources. The purpose of the proxies is to collect sensor data and to execute user-supplied queries (cf. [22]), for instance a join over an attribute of the measured data tuples. In order to compute that join for a given time interval, the proxy needs to query the sensors for their data. Since transmitting data is very expensive in terms of sensor battery power [23], the goal of the system is to transmit as little data as possible to extend the sensors' lifetime. Hence we have an optimization problem to select the right data to transmit such that the approximation error of the result is minimized subject to power consumption constraints (which is equivalent to data transmission constraints).

Join with Archive Support: Streams are not always arriving at a consistently high rate and it is not always desirable to obtain results of limited accuracy. One example are retail applications where sales activity is much higher during daytime than over night. Also, business analysis typically is more involved and hence incoming data is archived for future analysis. In other applications, e.g., intelligence and disaster monitoring, shedding load simply is not acceptable due to the risk of "missing the needle in the haystack". In such scenarios the stream processing system would operate in cooperation with an archive, e.g., a data warehouse. In *day* (i.e., peak load) mode it will produce approximate results for incoming data in real-time. In *night* (i.e., low load) mode when the arrival rate of new tuples is low, data from the archive is read and processed in order to refine earlier approximate results. For the join operator this means that during day time only a subset of the overall join result is computed. All tuples which are not completely processed are then post-processed at night by accessing the archive. Notice that now load is actually not shed, but rather deferred. Hence we will use the term semantic load *smoothing*.

Contributions of this Paper. In this paper we give an in-depth examination of semantic load shedding for data stream window joins. We present novel algorithms for approximating set-valued join results at tuple granularity. Our algorithms obtain the *best possible approximate result* according to a given error measure subject to given resource constraints. Specifically, we make the following contributions:

- We outline the design space of possible error measures and introduce a new *Archive-metric* for sliding window joins with archive support. We describe architectural models for approximating data stream sliding window joins. (Section 2)
- We then present results for one selected error measure—the *MAX-subset* measure, which maximizes the number of tuples in the approximate output of the join. More precisely, we present hardness results and algorithms for the static join case and optimal offline algorithms and very fast lightweight online heuristics for the sliding window join problem. (Section 3)
- We evaluate our algorithms on a large set of synthetic and real-life data (Section 4).

A discussion of related work (Section 5) and a summary and outlook to future work conclude this article (Section 6).

2. MODELS AND MEASURES

In this section we define the problem space. In particular we introduce different models for the approximate join computation problem and discuss measures for evaluating the quality of an approximate join result.

2.1 Models

Our goal is to process a sliding window equi-join between two data streams R and S. The join operator allocates memory for keeping internal state, i.e., those tuples which are within the current window. Newly arriving tuples are joined with all matching tuples of the other stream in the join memory. In order to guarantee the computation of the exact result the system at any point in time needs to accommodate the $2w$ tuples of the current window and it has to process incoming tuples at least as fast as they arrive. If these conditions are satisfied, an incoming tuple will remain in memory until it expires, i.e., it will spend its whole lifetime (w time units) in memory and generate output with all matching tuples in the other stream. However in case of resource shortage, tuples have to be dropped *before* they expire.

Modular vs. Integrated. We identify two general models for processing window joins–*modular* and *integrated* (cf. Figure 1). In both cases there is join memory of size M for the tuples in the current window and a queue for newly arriving stream tuples. In order to make educated decisions about which tuples to evict from the join or the queue in case

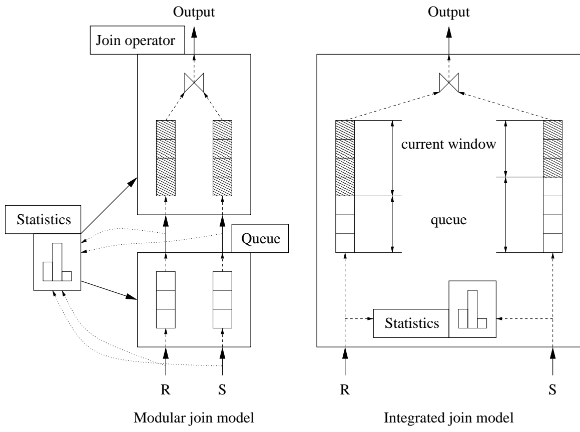

Figure 1: Join processing models

of resource shortages a statistics module gathers information about the tuple distribution. The main difference between the two models lies in the degree of integration between the components.

In the modular case the queue module only has limited knowledge about the contents of the join memory (for example, just a histogram about the frequencies of join attribute values in memory) and vice versa. Each module uses its own policy for deciding which tuples to drop in case of resource shortages. These decisions are only influenced by the input from the statistics module. The modular design offers the advantage that different operators can be designed and implemented independently. If streams provide input for multiple operators, queues can be shared, increasing memory efficiency. Note that different operators might have different preferences for which tuples to evict from the queue. This can be taken into account by considering input from several statistics modules. The integrated model combines the queue with the join memory. The benefits are potentially better decisions based on the combined exact knowledge of both memory contents. On the other hand the internal queue cannot be shared easily with other operators.

Fast CPU vs. Slow CPU. For analysis purposes we also distinguish between the *fast CPU* and the *slow CPU* case (similar to [21]). The system is a fast CPU system if incoming tuples can be processed at least as quickly as they arrive. The queue is not needed since tuples are directly *pushed* into the join, therefore both modular and integrated model essentially are equivalent. Conceptually the join has internal state of size M and two additional buffer cells for the new arriving tuple of each stream. When tuples arrive they are instantly joined with their partner tuples of the other relation in the join memory. Then it is decided if the tuple will be added to the join memory (potentially evicting another tuple). Hence an arriving tuple will *always* be seen by the join. The optimization goal is to evict and retain tuples such that the approximation error is minimized. Possible error measures will be discussed shortly.

In the slow CPU case tuples arrive faster than they can be processed. This implies that the queue is necessary for buffering incoming tuples. The join operator now *pulls* tuples from the queue whenever it has processed the previous input. Clearly, the queue will fill up over time and overflow, hence tuples have to be dropped from it without ever reaching the join. This is referred to as load shedding in [6]. If a tuple reaches the join, it is processed as discussed for the fast CPU case. The slow CPU case therefore constitutes a generalization of the fast CPU case. In the latter case, approximations arise due to memory restrictions, while in the former case, approximations arise due to both memory and processing constraints. The load shedding in the queue affects the contents of the streams that reach the join operator, therefore the optimization goal is to find the best join approximation over *all possible* load shedding strategies.

2.2 Error Measures

The output of the join operator is a set of tuples, more precisely a multiset. In the following for simplicity the term *set* will refer to multisets as well. There is no single universally accepted measure for evaluating the quality of an approximation to a set-valued query result [19]. One well-known and widely used measure is the symmetric difference. For two sets X and Y it is computed as $|(X-Y)\cup(Y-X)|$. For equi-joins dropping tuples before they expire naturally leads to a situation where the generated output is a *subset* of the exact join result (i.e., the result if there was no resource shortage). In that case the symmetric difference simplifies to the number of *missing* output tuples. We will therefore refer to it as the *MAX-subset* measure. This measure will be the principal focus of this paper.

In the data mining and information retrieval communities several set-theoretic similarity measures have been used [17, 28]. The most widely used similarity measures between two sets X and Y are Matching coefficient $|X \cap Y|$, Dice coefficient $2\frac{|X\cap Y|}{|X|+|Y|}$, Jaccard coefficient $\frac{|X\cap Y|}{|X\cup Y|}$ and Cosine coefficient $\frac{|X\cap Y|}{\sqrt{|X|+|Y|}}$. For $X \subseteq Y$ all these measures are maximized by maximizing the size of set X, hence they are equivalent to MAX-subset. The Overlap coefficient $\frac{|X\cap Y|}{\min\{|X|,|Y|\}}$ equals 1 for $X \subseteq Y$.

The recently introduced *Earth Mover's Distance* (EMD) [26] is mainly used as a similarity measure in image processing. It is defined as the amount of work required to transform a set X into another set Y of equal or greater mass (number of tuples). Both sets are viewed as masses distributed over a data space for which a distance measure exists. The work required to transform X into Y is measured as the sum over all products of mass transported by the transport distance. EMD is the minimum work over all possible transformations of X into Y. If $X \subseteq Y$ it trivially evaluates to 0.

The Match And Compare (MAC) [19] set similarity measure also requires a distance metric between the tuples of the two sets. First a minimum cost cover of the complete bipartite graph whose nodes correspond to the tuples and whose edges have the weight of the respective distances is found. Then the overall set distance is computed as a function of the weights of the edges in the cover and the number of edges incident to each node.

We introduce a novel "error" measure, the *Archive-metric* (ArM), which is relevant for semantic load smoothing. As mentioned earlier, in some applications input data is stored in archives and hence during low-load periods can be used to refine approximate results which were obtained during periods of high load. Since the final result will be exact, ArM does not measure the approximation quality but rather

the amount of work required for finishing incomplete work. Since accessing an archive is much slower than memory access, we approximate this post-processing cost by the number of tuples which could not be matched with all their partner tuples because of resource shortage (during peak load). This quality measure is a simplification in that it assumes that the post-processing cost (reading incomplete tuples and their partners from the archive and joining them) depends linearly on the *number* of such tuples only. Other measures like average processing time per tuple or average output delay are reasonable as well but not discussed in detail here due to space constraints.

ArM is formally defined as follows. Let

$$\delta_R(i,j) = \begin{cases} 1 & \text{, if } r(i) \text{ survived for at least } j \text{ time in memory} \\ 0 & \text{, otherwise.} \end{cases}$$

This variable indicates when a tuple that arrives at time i is still in memory and is similarly defined for stream S. Furthermore we use

$$S^{<}(i) = \{j : j \in [i - w + 1, i - 1] \wedge s(j) = r(i)\} \ .$$

This index set contains the arrival times of all join partners of $r(i)$ in S which arrived before it. A similar set $R^{<}$ is defined for each $s(i)$.

Note that $r(i)$ is completely processed if all $s(j)$ for $j \in S^{<}(i)$ are in memory at time i and if $r(i)$ is in memory until time $j_{r(i)}$, where $j_{r(i)} = \max\{j : j \in [i, i + w - 1] \wedge s(j) = r(i)\}$ (i.e., the time when the last join partner of $r(i)$ arrives on stream S). Since $\delta_R(i,j)$ is equal to 0 iff $r(i)$ has not survived for j time, we know that $r(i)$ is incomplete iff $\delta_R(i, j_{r(i)} - i) \prod_{j \in S^{<}(i)} \delta_S(j, i-j) = 0$. Thus ArM is defined as

$$\sum (\delta_R(i, j_{r(i)} - i) \prod_{j \in S^{<}(i)} \delta_S(j, j - i) + \delta_S(i, j_{s(i)} - i) \prod_{j \in R^{<}(i)} \delta_R(j, j - i))) \ .$$

In the following sections we will focus on the load shedding problem for the fast CPU, integrated join model and the MAX-subset error measure. Examining the other models and error measures is left for future work.

3. MAX-SUBSET MEASURE

We first examine the static join case, then present optimal offline and efficient online algorithms for the sliding window join. The latter is focused on the fast CPU, integrated join model.

3.1 Static Case

We consider the following two relation (static) join load shedding problem: We wish to compute an equi-join of two (non streaming) relations A and B. However, as motivated in the introduction with a sensor network scenario, due to reasons such as transmission, memory, or processing time restrictions, a total of k tuples need to be dropped from the input buffers. Hence the join of A and B needs to be computed on the resultant *truncated* input. Each of the k dropped tuples may be chosen from either relation, and we call the resultant join the *k-truncated join* of A and B. For most join conditions, such as equality joins, inequality joins etc., dropping of input tuples only implies a (potential) dropping of tuples in the output set, and hence the output of the k-truncated join on A and B is a subset of the output of the non truncated join of A and B. In such cases, assuming that all output tuples from the join of A and B have the same importance, a natural measure of the *loss* or *approximation* to the join of A and B is the MAX-subset measure (cf. Section 2.2). Thus our aim is to find a set of k tuples to be dropped from the input relations such that the size of the k-truncated join result is as large as possible.

We can model the above as a graph problem, as follows: Consider a bipartite graph $G(V_A, V_B, E)$, with its two partitions V_A and V_B representing the relations A and B respectively. Each partition has one node for every tuple in the relation it represents. We have an edge between a node $n_A \in V_A$ and a node $n_B \in V_B$ if the tuples corresponding to n_A and n_B satisfy the join condition. Thus the bipartite graph G has an edge for every tuple in the join result of A and B. Since our join condition is an equality on one or more of the attributes of A and B, it is easy to see that G will consist of a union of mutually disjoint fully connected bipartite components (called *Kurotowski* components). Each Kurotowski component can be represented by a pair of integers m, n where m and n are the number of nodes from V_A and V_B respectively in the component. We denote such a Kurotowski component by $K(m, n)$. Thus our k-truncated join approximation problem is equivalent to finding a set of k nodes in the bipartite join-graph whose deletion results in the deletion of the fewest number of edges (which represent join tuples). Note that since dropping a tuple from one of the input relations of a join results in the dropping of all the output tuples produced as a result of that tuple joining with other tuples from the other join relation, our definition of node deletion requires that deleting a node results in the deletion of all the edges incident on that node. For arbitrary bipartite graphs, i.e., bipartite graphs not necessarily representing a join, the above problem can be shown to be NP-Hard.

We are now ready to state a couple of versions of the k-truncated join approximation problem, modeled as a graph optimization problem as described below.

Primal version

Input: A bipartite graph consisting of c mutually disjoint *Kurotowski* subgraphs specified by the c integer pairs $K(m_1, n_1), K(m_2, n_2), \ldots, K(m_c, n_c)$, and an integer k.
Output: A set of k nodes from the bipartite graph whose *deletion* from the graph results in the deletion of the fewest number of edges. Note that when we delete a node, all edges incident on the node get deleted.

A potentially useful **variant** of the above problem is the k_A, k_B-truncated join approximation problem in which we are required to delete k_A, k_B tuples from the two joining relations respectively as opposed to k tuples overall.

Dual version

Input: Same as for primal version.
Output: A set of k nodes to be *retained* in the bipartite graph such that the subgraph induced by them has the highest number of edges amongst all subgraphs with k nodes.

Since an optimal solution to the primal version where k

nodes are selected for deletion is an optimal solution to the dual problem where $n-k$ nodes are retained (n denotes the total number of nodes in the bipartite graph), an optimal algorithm for either one of them trivially implies an optimal algorithm for the other.

In the context of the motivating sensor networks scenario, a solution to the problem formulated above may be used for join approximation at a proxy as follows: A compact value distribution histogram of the join attribute is transmitted independently by each sensor to the proxy, which will then run the algorithm for suitable parameters based on its knowledge of the power constraints (which may be conveyed to the proxy by the sensors themselves) and determine the set of tuples to be requested from each sensor. The aim here is to maximize the size of the truncated join, subject to an upper bound on the number of input tuples transmitted by the sensors.

3.1.1 An Optimal Dynamic Programming Solution

We consider the dual formulation, where a total of k nodes need to be retained. Given c Kurotowski components, we order the components as per some arbitrary ordering, and let $K(m_i, n_i)$ denote the i-th component ($0 \leq i \leq c$) as per this ordering.

Given a *single* Kurotowski component $K(m, n)$, the optimal way to retain $0 \leq p \leq m+n$ of its nodes (or equivalently, to delete $m+n-p$ of its nodes), is to retain $m' \leq m$ nodes from the first partition and $n' \leq n$ nodes from the second partition such that $m' \cdot n'$ (i.e., the number of retained edges) is as large as possible. It can easily be shown that this corresponds to choosing m' and n' such that $m'+n' = p$ and $|m'-n'|$ is as small as possible. Thus, the p nodes to be retained can be chosen one by one by selecting alternately a node from the 'm partition' followed by a node from the 'n partition' until a count of p is reached. If all the nodes of one partition are exhausted before a count of p is reached, we simply select the remaining nodes to be retained from the larger partition. Let $C_{m,n}(p)$ denote the maximum number of edges that can be retained when p ($\leq m+n$) nodes are retained from a Kurotowski $K(m, n)$ component. It can be shown that $C_{m,n}(p)$ is given by: (w.l.o.g., assume $m \geq n$)

$$C_{m,n}(p) = \begin{cases} (p/2)^2 & \text{if } p \leq 2n,\ p \text{ even} \\ (p^2-1)/4 & \text{if } p \leq 2n,\ p \text{ odd} \\ n(p-n) & \text{else.} \end{cases}$$

Let $T(i, j)$ denote the optimal benefit (i.e., the max. number of edges retained) of retaining j nodes from the first i Kurotowski components, as per our ordering. Then, for $i > 1$:

$$T(1, j) = \begin{cases} C_{m_1,n_1}(j) & \text{if } 0 \leq j \leq m_1 + n_1 \\ -\infty & \text{if } j > m_1 + n_1 \end{cases}$$

$$T(i, j) = \max \begin{cases} T(i-1, j), \\ T(i-1, j-1) + C_{m_i,n_i}(1), \\ T(i-1, j-2) + C_{m_i,n_i}(2), \\ \vdots \\ T(i-1, j-m_i-n_i) + C_{m_i,n_i}(m_i+n_i) \end{cases}$$

Intuitively, the second formula states that the optimal way to retain j nodes from i components is to choose the best from the following options: Either retain j nodes optimally from the first $i-1$ components, or retain $j-1$ nodes optimally from the first $i-1$ components and retain 1 node optimally from the i-th component, or retain $j-2$ nodes optimally from the first $i-1$ components and retain 2 nodes optimally from the i-th component, and so on. Thus, the value we are interested in is $T(c, k)$. By keeping track of the terms which provide the maximum in the second formula above, we can also maintain the exact set of nodes retained from each component in the optimal solution.

Analysis: To compute $T(c, k)$, we need to compute $c \cdot k$ entries in the dynamic programming matrix T, and each entry takes $O(k)$ time to compute. Thus, the overall running time of this algorithm is $O(c \cdot k^2)$. By considering a three-dimensional matrix T with entries of the form $T(c, k_A, k_B)$, it is possible to extend the above algorithm to handle the variant where one needs to delete k_A, k_B nodes from the two bipartite partitions respectively.

Strictly speaking, the above algorithm is pseudo-polynomial in the input size ($O(c \cdot \log(max_i\{m_i, n_i\}) + \log k)$), since the input is logarithmic in the parameter k. However, in our case, since we wish to apply the algorithm for retaining/deleting k nodes, we do need to spend at least $O(k)$ processing the two relations. Also note that the algorithm is polynomial in the sizes of the input relations.

3.1.2 A Hardness Result for Multi-Relation Joins

Consider a join of three relations A, B and C, and suppose that we need to delete (or retain) k_A, k_B, k_C tuples from the input relations respectively, or k tuples overall, so as to maximize the number of join tuples that are produced from the retained input tuples. We call this the 3-relation static join load shedding problem.

THEOREM 1. *The 3-relation static join load shedding problem is NP-Hard.*

PROOF. Omitted due to space constraints. The main idea is to model the problem using a tripartite graph and to use a reduction from the *balanced biclique problem* [11]. □

The above result can be used to show that the m-relation static join load shedding problem is NP-hard for $m \geq 3$. However, there is a trivial m-approximation to this problem for the formulation where one needs to delete (or retain) k_i tuples from join relation A_i ($1 \leq i \leq m$). The idea is to *independently* select for each A_i the k_i tuples for deletion which produce the fewest output tuples. Assume the number of lost output tuples caused by removing k_i tuples from A_i is p_i. The optimal algorithm at least loses $\max\{p_1, p_2, \ldots, p_m\}$ output tuples. The approximation algorithm will at most lose $\sum_{i=1}^{m} p_i$ output tuples, therefore guaranteeing an m-approximation.

3.2 Fast CPU and Offline

We are now considering the standard window join algorithm as discussed in Section 2.1. We develop an algorithm *OPT-offline* that minimizes the MAX-subset error in the fast CPU case under the assumption that all tuples that will arrive in future are already known to the algorithm. Note that streams are infinite, and therefore knowing the whole future can not be modeled. However, this idealized algorithm is used to provide the baseline for measuring the efficiency of any real online algorithms over a given *subset* of the overall stream. For this subset we can compute the optimal result using OPT-offline and compare this result to how an online technique which does not know the future

performs on the same input. Since in the slow CPU case even more tuples have to be dropped, OPT-offline also constitutes an upper bound for any technique for the slow CPU case.

Recall that the join memory holds a total of M tuples, not necessarily distributed evenly between R and S. We will now describe how to formulate the OPT-offline optimization problem as a network flow problem that allows the efficient computation of the best possible approximation under the MAX-subset measure.

3.2.1 The Flow Graph

The main idea is to define a flow graph such that each node corresponds to a tuple being in memory at a certain time. The arcs implicitly model all possible combinations of keeping or dropping tuples. Sending flow through an arc intuitively indicates that the corresponding tuple is in memory, i.e., was not dropped. Since we want to minimize the MAX-subset error, our goal is to find the optimal strategy of keeping and dropping tuples such that the overall result size is maximized.

We assign costs to the arcs in such a way that an optimal flow corresponds to the strategy which produces the most output tuples. To do so we assign cost factor -1 to each arc which corresponds to a result tuple. Arcs that do not correspond to output tuples have cost factor 0. Solving a *min-cost* linear flow problem will then find our optimal strategy efficiently. For the sake of simplicity we will illustrate the flow graph construction with an example where the memory M is evenly shared between streams R and S. Later we generalize the approach.

Let the streams be $R = 1, 1, 1, 3, 2$ and $S = 2, 3, 1, 1, 3$ and assume the first value arrives at time 0, the next at time 1, and so on. Furthermore let the window size be $w = 3$ and the available join memory $M = 2$. Recall that R and S each receive $M/2$, i.e., one memory unit to keep tuples in the current window. The corresponding flow network is shown in Figure 2. For simplicity arc cost factors are only indicated for arcs with cost -1. Overall the nodes in the upper half correspond to events related to R-tuples, while the nodes on the lower half correspond to the events related to S-tuples. Nodes with label $x(i) : j$ correspond to the event that the tuple that arrived at time i in stream X is in memory at time j. Nodes s and t are the source and sink of the flow graph, respectively. The node labeled $t = 2$ models the fact that at time 2 the tuples arriving in both streams have the same join attribute value (equal to 1).

The flow graph is constructed as follows. All arcs have capacity 1, i.e., they can transmit any flow between 0 and 1 (both inclusive). Node s has supply $M + 1$ and node t has demand $M + 1$. All other nodes have no supply/demand. Except for the top path $s \rightarrow (t = 2) \rightarrow t$ which has a special purpose and will be discussed later, s has M outgoing arcs. $M/2$ of them point to R-tuple nodes, the other $M/2$ to S-tuple nodes, modeling the arrival of the first $M/2$ tuples from each stream (arcs from s to $r(0) : 0$ and $s(0) : 0$ in the example). The idea behind these arcs is that the first M arriving tuples will always fit in memory, which will be reflected by a flow of 1 through each arc (a total flow of M).

Since the memory is now filled, the next arriving tuples could replace existing tuples in memory. Recall that we currently fix the memory allocated to R and S, therefore a newly arriving R-tuple can only replace another R-tuple in memory, but not an S-tuple (and vice versa). The possibility of replacement is modeled by the non-horizontal arcs. In the example arc $r(0) : 0 \rightarrow r(1) : 1$ indicates that tuple $r(0)$ which is currently in memory could be replaced at time 1 by the newly arriving $r(1)$.

The horizontal arcs model the fact that a tuple survives in memory. For instance $r(0) : 0 \rightarrow r(0) : 2$ indicates that $r(0)$ could still be in memory at time 2. Notice that $w = 3$, therefore $r(0)$ will expire at time 3. This means there is no benefit in keeping $r(0)$ in memory after it has been matched with a partner tuple arriving at time 2, therefore there is no outgoing horizontal arc from node $r(0) : 2$.

Finally, at the end of the "stream" (recall that OPT-offline works on finite subsets of a real data stream) all nodes that correspond to tuples in the current window are connected to the sink t.

In Figure 2 the general design patterns of the flow graph are marked by dotted line boxes. The tall box shows a subset of nodes which correspond to the events at a certain time $t = 3$. At that time the window contains $r(1)$, $r(2)$, $s(1)$, $s(2)$, and the newly arriving $r(3)$ and $s(3)$. Which tuples are actually in memory after the arrival of the new tuples is determined by where flow is sent. Similarly, the wide box corresponds to the events of a tuple $(s(1))$ being in memory at time 1, 2, and 3, respectively. Again, sending flow through an arc indicates that $s(1)$ is in memory at the corresponding time.

The path on the top contains a node for each pair $(r(i), s(i))$ where $r(i) = s(i)$. In our fast CPU processing model the newly arriving tuples are always joined with their partners in the join memory, and also with the tuple that arrives on the other stream at the same time. The latter is modeled by the top path.

As mentioned before, all arcs $i \rightarrow j$ in the flow graph have capacity 1, i.e., they can transport any flow $f(i, j)$ with $0 \leq f(i, j) \leq 1$. The cost of an arc flow is computed as $f(i, j) \cdot c(i, j)$ where $c(i, j) \in \{0, -1\}$. The $c(i, j)$ values are determined as follows. Recall that a flow through an arc corresponds to a tuple being in memory. The tuple in memory produces exactly one output tuple iff the tuple arriving at the corresponding time in the other stream has the same join attribute value. If that is the case, the arc cost is set to -1, otherwise to 0. In the example we have $r(0) = 1$. Hence when $s(2) = 1$ arrives and $r(0)$ is in memory, an output tuple is produced. This is modeled by arc $r(0) : 0 \rightarrow r(0) : 2$ which has cost factor -1. If there is a flow of 1 through this arc, this corresponds to $r(0)$ being in memory at time 2, hence an output tuple with $s(2)$ is produced (reflected by cost -1 for this flow).

Strictly speaking the arcs adjacent to s (except for the top path) could have a cost factor smaller than -1 if one of the first $M/2$ tuples in a stream generates more than one output tuple with the first $M/2$ tuples from the other stream. This does not affect the complexity of the model, since the optimal solution always has to send full flow along each arc adjacent to s. Also, to avoid "startup-effects" when comparing different techniques, the output produced until time $M/2$ should not be counted because any algorithm no matter how good or bad it is will keep the first $M/2$ tuples from each stream in memory.

In Figure 2 the optimal flow is indicated by the bold arcs. The output corresponding to this optimal flow is shown in the figure. Note that because of insufficient memory two

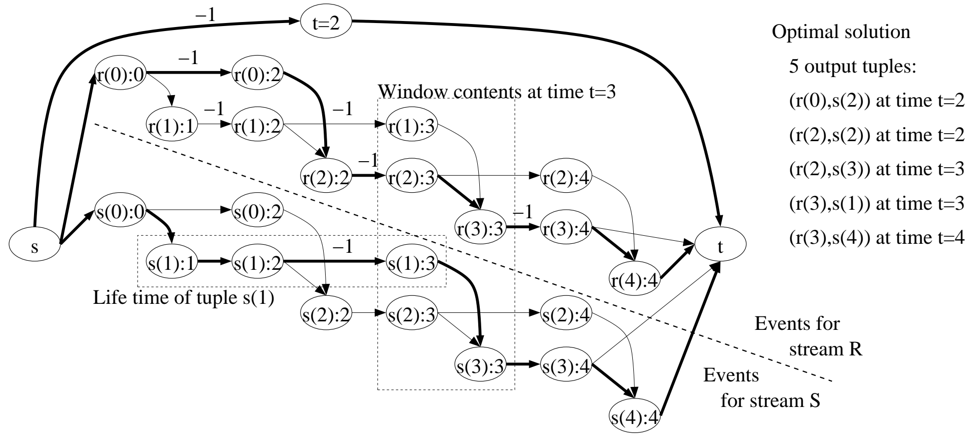

Figure 2: Example flow graph

output tuples are missed $((r(1), s(2))$ and $(r(1), s(3)))$.

The *generalization to variable memory allocation*, i.e., sharing the memory in any ratio between R- and S-tuples is simple. We just need to add "cross-arcs" between R-nodes and S-nodes in the graph which model the fact that now an R-tuple can replace an S-tuple and vice versa. In Figure 2 such arcs would be $r(0):0 \rightarrow s(1):1$, $s(0):0 \rightarrow r(1):1$, $r(0):2 \rightarrow s(2):2$, $r(1):2 \rightarrow s(2):2$, and so on. In general each node (except s, t, and the top path nodes) now not only has an outgoing arc to the newly arriving tuple of its own stream, by also another arc to the newly arriving tuple in the *other* stream.

3.2.2 OPT-offline Algorithm

As the flow network is modeled, the overall flow cost corresponds to the number of generated output tuples for the given input. Hence solving the linear *minimum-cost flow* problem for this graph produces the optimal strategy for deciding which tuples to drop from memory at each time instant.

First we show that the flow graph discussed above correctly models the offline algorithm. Note that the arcs do not allow more than a flow of M through the main network, and exactly a flow of 1 through the top path. This ensures the memory constraint. Also, the way the arcs are combined correctly models the tuple events. It is not possible for a dropped tuple to re-enter the memory and only tuples in memory produce output. The exact proof of correctness is omitted due to space constraints.

There is one major property left to be shown in order to establish the correctness of the model. We have to ensure that there are no *partial flows*, i.e., flows $f(i,j)$ which are not either 0 or 1. This is ensured by the following theorem.

THEOREM 2. *If the flow problem has an optimal solution, and all capacity constraints and costs are integral, then there is an optimal solution which is also integral.*

PROOF. See [25], page 239. □

We can use any standard linear minimum cost flow algorithm that finds the optimal integer solution of the flow problem. Since the highest absolute arc cost in our network is 1, known algorithms find the optimal integer solution in time $O(n^2 m \log n)$ (cf. [13]), where m is the number of arcs and n the number of nodes.

For our problem we can derive the following upper bounds for the number of nodes and arcs. Let N denote the number of tuples in each stream. Each node belongs to at most w windows. Furthermore there are at most N pairs $(r(i), s(i))$ with $r(i) = s(i)$. Together with source and sink node there are at most $2wN + N + 2 = \theta(wN)$ nodes. Each node has at most three outgoing arcs (for the events "remain in memory", "being replaced by new R-tuple", "being replaced by new S-tuple"). Only the source node has $M+1$ outgoing arcs, the sink has none. Hence the total number of arcs is at most $(M + 1 + 3 \cdot (\text{numberNodes} - 2))$, i.e., is $O(wN + M)$. The formulation as a flow problem enables the computation of the optimal offline solution in time polynomial in stream, window, and memory size.

3.3 Fast CPU and Online

An online algorithm does not know which tuples will arrive in the future. Hence all we can do is maximize the expected output size assuming certain arrival probabilities. However, even such probabilities and possible independence assumptions only approximate the true future. At the same time any real online algorithm faces the challenge that the memory and CPU resources it consumes are not available for the actual join processing. Hence our goal is to design very fast and lightweight techniques which add the lowest possible overhead but nevertheless try to maximize the output size based on approximate future knowledge.

3.3.1 PROB Heuristic

PROB estimates for each value in the domain of the join attribute the probability of a tuple with this value arriving on stream R and stream S. For attribute value a let these probabilities be $p_R(a)$ and $p_S(a)$. A tuple's priority is equivalent to the corresponding probability value. For instance for $r(i)$ the priority is $p_S(r(i))$. Ties are broken by giving higher priority to the tuple that arrived later.

Note that PROB favors high partner arrival probabilities over age. In other words, a tuple $r(i)$ with a high $p_S(r(i))$ value is always preferred over $r(j)$ with $p_S(r(j)) < p_S(r(i))$, even if $r(i)$ arrived much earlier than $r(j)$ and possibly only has a few more time units until it expires.

This heuristic is motivated by the expectation that tuples with a higher probability of finding incoming partner tuples are the ones that produce the most output results. Even if

a newly arriving tuple with low partner-arrival probability was admitted to memory, it would soon be replaced by a later arriving tuple with higher partner-arrival probability, hence it seems better to greedily "hold on" to the best tuples available.

PROB can be used both for fixed memory allocation between R and S, but also when the allocation is variable. In the former case there are two priority queues—one for R and one for S-tuples. In the latter case there is a single priority queue for all in-memory tuples of both streams. PROB can also easily deal with varying memory and window sizes.

A practical issue is to compute the values of $p_R()$ and $p_S()$ without knowing the future. This problem is similar to the online caching problem. Hence we can use the same techniques of using history to predict the future. Depending on the amount of available memory the history statistics can be exact or approximate, e.g., any of the previously proposed data stream histograms or wavelets (see discussion of related work).

3.3.2 LIFE Heuristic

The LIFE heuristic is also based on estimates of the $p_R()$ and $p_S()$ values. However, while PROB favors partner arrival probability over remaining lifetime of a tuple, LIFE aims at giving more weight to the latter. The priority of a tuple $r(i)$ with remaining lifetime t is computed as $t \cdot p_S(r(i))$.

Note that with increasing window size newly arriving tuples at some point are almost guaranteed to enter the memory because of their high lifetime value. LIFE in general overestimates the expected number of output tuples because tuples might be evicted before they expire, whereas the priority calculations are based on time to expiry. This holds especially for tuples with low $p_R()$ or $p_S()$ values.

Like PROB, LIFE can be used for both fixed and variable memory allocation between R and S-tuples, and also for varying window size and memory.

4. EXPERIMENTS

We perform an extensive experimental evaluation of the sliding window join approximation techniques suggested in Section 3.3 on both synthetic and real life datasets. We compare the performance of these techniques with the state of the art, i.e., dropping tuples randomly from the join input buffers (henceforth referred to as RAND), as well as with the optimal offline approach OPT-offline described in Section 3.2. For brevity we will abbreviate OPT-offline as OPT where appropriate. Our experiments indicate that the simple heuristic approach (PROB) of dropping tuples from buffers based on the probabilities of the corresponding tuples arriving in the other stream does surprisingly well in practice.

For solving the linear min-cost network flow problem arising out of the optimal offline join approximation algorithm we used the CS2 network flow solver as described in [13]. This solver is based on one of the fastest known algorithms for min-cost flow problems, which still is super-linear in the input size (cf. Section 3.2.2). Hence for all the experiments involving comparison with OPT, we restrict the stream length to 5600 tuples. This number was selected to guarantee for any window size that at least 4000 tuples are processed after a warmup phase of $2w$ (see below for more discussion).

As we shall see later in this section, the input size does not affect the validity of the conclusions drawn from the graphs obtained on these streams, since our heuristics scale well with stream length. Also, the graphs for larger stream lengths and window and memory sizes resemble closely the graphs obtained on stream lengths of 5600. For our synthetic datasets, we used Zipfian distributions with varying degrees of skew and correlation between the data in the two joining streams. Within a stream, the data values were generated in iid (independently and identically distributed) fashion from the corresponding Zipfian distribution. For our real-life dataset experiments, we used a weather dataset [16]. The input streams had the same tuple arrival rates, with a tuple arriving on each stream at every timestep.

4.1 Effect of Window Size

Our first set of experiments was aimed at studying the behavior of the various join approximation algorithms for different window sizes. Figures 3 and 4 show the number of join output tuples as the amount of available memory is varied for the different algorithms for window sizes (w) of 400 and 800 respectively. In all our experiments where we vary memory M, we vary it as $0.1w$, $0.25w$, $0.5w$, w and $1.5w$. To guarantee exact computation of the join result, $M = 2w$ would be necessary.[1]

Note that all algorithms store the first M/2 tuples from each stream in memory and hence output the same set of resulting join tuples. Hence in order to prevent such startup effects we introduce a *warmup phase* during which output is not counted. The warmup phase is selected as twice the window size. This ensures that all the tuples that filled the memory at the start of the experiment will have expired, and the join approximation algorithm will have reached a stable phase before generating output.

The input data streams in Figures 3 and 4 are generated from uncorrelated Zipf distributions with parameter 1. The domain size of the data was set to 50. We shall justify this choice of domain size later on in this section when we examine the behavior of the algorithms for different domain sizes. As expected, the behavior of the various algorithms RAND, PROB, OPT and LIFE is similar for different window sizes. In the figures, EXACT refers to the number of output tuples generated if the sliding window join were to be computed *exactly*, i.e., with $2w$ memory. The number of output tuples generated by RAND increases linearly with available memory, as expected. As can be seen from Figures 3 and 4, the PROB heuristic by far outperforms the RAND and LIFE approaches, and is very close to the OPT curve, which is the optimal offline algorithm representing an upper bound on the best performance (in terms of number of output tuples generated) possible by any online algorithm. Similar behavior was observed for other window sizes.

The reason for the poor performance of the LIFE algorithm is that it tries to predict the number of output tuples generated by a tuple in memory *assuming that the tuple will remain in memory for its entire lifetime*. However, tuples whose join attribute value has a very low probability of appearing in the other stream are unlikely to survive until they expire, since they will be evicted by incoming tuples which have a higher probability of finding a matching tuple in the other stream. Hence the priority value based on the prod-

[1]Strictly speaking only $M = 2w - 2$ is needed because of the extra memory cells provided by the input buffer in the fast CPU model (cf. Section 2.1).

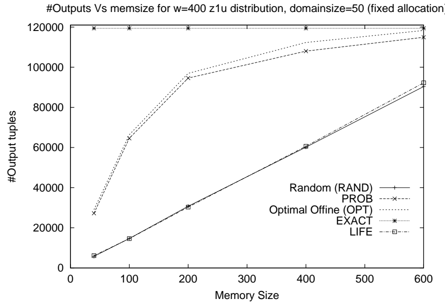

Figure 3: Window size 400

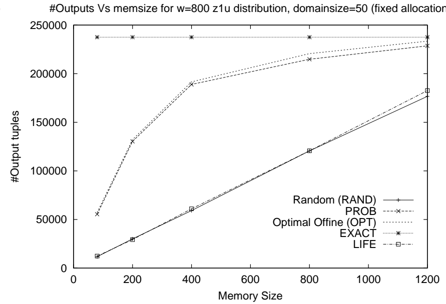

Figure 4: Window size 800

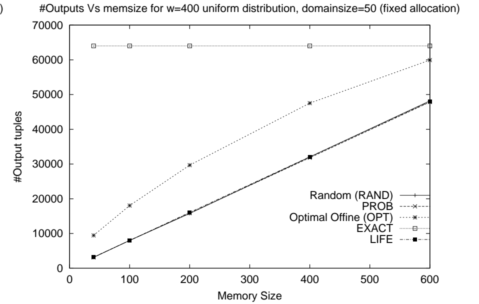

Figure 5: Uniform input

uct of probabilities and time to expiry used in LIFE differs significantly from the actual number of output tuples produced. This in turn leads to a higher percentage of tuples in memory that produce little output as compared to PROB.

While the relative performance of the different algorithms varies depending on the distributions of the input streams, they do not change as the window size is varied for the same pair of joining input streams. Since the window size does not impact the nature of the graphs obtained, the results for the rest of the experiments in this section are shown only for a window size of 400. Similar graphs were obtained for various other window sizes in each of these cases.

4.2 Uniform Data on Both Inputs

If both incoming data streams consist of tuples with uniformly distributed join attribute values, we expect all online algorithms to produce about the same number of output tuples. The reason for this is that all the tuples in memory have the same probability of seeing a counterpart (i.e., a tuple with the same join attribute value) in the other stream, therefore there is no reason to prefer keeping one tuple over another. This is equivalent to RAND's strategy of evicting random tuples from memory. Figure 5 confirms our prediction, showing the performance of the different algorithms for a window size of 400 when both the incoming streams have a uniform data distribution. As can be seen from the graph, the online algorithms (RAND, PROB and LIFE) perform equally poorly. Notice that even knowing the future (OPT curve) does not result in a major improvement. This is in contrast to the results shown in Figure 3. There for an almost identical setup (the difference being Zipf distributed join attribute values in one stream) both OPT and PROB are much more rapidly approaching the exact result with increasing memory. The non-uniform distribution generates tuples which are more valuable than others because of the frequency of their join attribute value in the stream. Both OPT and PROB successfully identify these tuples and keep them in memory.

PROB versus LIFE. As can be seen from Figures 3, 4 and 5, the LIFE heuristic does only marginally better than RAND for various data distributions, for reasons explained earlier. Similar behavior was obtained when the two input streams had non-identical distributions, e.g., Zipf(1.0) and uniform data distributions respectively, as well as on the real datasets used in the later experiments. Hence the LIFE approach is not included for comparison in the remaining experiments in this section.

4.3 Effect of skew

Figure 6 nicely brings out the effect of skew in the input data streams on the performance of the algorithms. The number of output tuples generated by the RAND and PROB algorithms is plotted as a fraction of the number of tuples generated by OPT as a function of the Zipfian skew parameter. Both the arriving input streams have Zipfian distribution with the same parameter. In Figure 6, the distributions of the two input streams are uncorrelated. Results for correlated Zipf distributions, e.g., where high (low) frequency values on one stream are also high (resp. low) frequency values on the other stream, were almost identical and hence are not shown here. The similarity of the results for different degrees of correlation indicates that the correlation between the two data streams does not affect the relative performance of the algorithms. This is because in the case of PROB, the decision to retain or drop tuples from one relation only depends on the data distribution of the other joining relation, and not on the its own data distribution or the correlation between the two. Clearly in the case of RAND, the eviction policy does not depend on the data distributions at all. Thus, while most of the experimental results shown were obtained for uncorrelated streams, the observations are similar for correlated and anti-correlated distributions as well. Note however, that correlation does affect the *total number* of output tuples generated by the joins.

As can be seen from the graph, for uniform data distribution (Zipf with parameter 0), the performance of RAND and PROB is essentially identical as has been noted earlier. However, as the skew in the input is increased, PROB gains an advantage over RAND because it is able to distinguish between tuples that have different probabilities of joining with tuples on the other stream. Hence, as the skew becomes larger, the performance gap between RAND and PROB increases rapidly.

Both window and memory size in the experiment were set to 400. Similarly shaped graphs were obtained for other memory sizes. Note that even for $M = w$ (i.e., at only 50% of the actually needed memory for exact computation), the PROB approach does extremely well, generating over 96% of the output tuples for input with moderate to high skew that can be generated by the optimal offline algorithm (OPT).

Variable memory allocation to the streams. The experimental results discussed so far were obtained for a fixed memory allocation of $M/2$ to R and S, i.e., incoming R-tuples (S-tuples) could only replace another R-tuple (S-tuple) in memory. We also compared the performance for the case when the memory allocated to each stream can vary while the total amount of memory is kept constant.

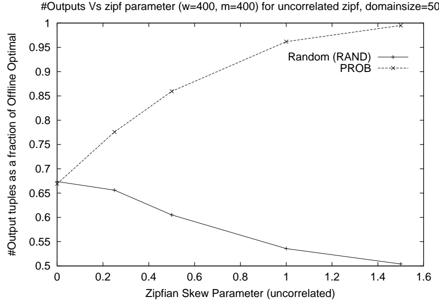

Figure 6: Uncorrelated Zipf

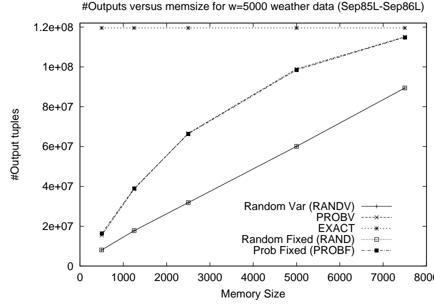

Figure 7: Weather data: Performance

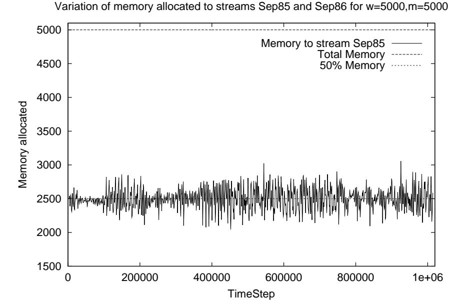

Figure 8: Weather data: Memory allocation

Hence an incoming R-tuple can replace an S-tuple in memory and vice versa. In the following we will use PROB, RAND, and OPT for the fixed memory algorithms, and PROBV, RANDV, and OPTV for their variable memory counterparts.

Our experiments for uniform and Zipf distributed data streams led to the expected results. Both PROBV and OPTV performed better than their fixed-allocation counterparts. The performance difference increased with increasing difference in skew between the distributions of R and S, but never exceeded 10% (output size). As a tendency the stream with the higher Zipf parameter received more memory, up to a share of 75%. Due to space constraints more detailed results are not presented here. Instead the issue of fixed versus variable memory shares is shown in Section 4.5 for real life data.

4.4 Effect of Domain Size

Figures 9, 10 and 11 bring out the effect of domain size (10, 50, and 200 respectively) of the join attribute on the performance of the algorithms. The graphs show the number of output tuples generated by the various algorithms as a fraction of the output generated by the optimal offline algorithm OPT as a function of memory size (window size 400, Zipf(1.0) distribution for both input streams). An increase in domain size has opposite effects on the performance of OPT and PROB. As the domain size increases, the performance of PROB seems to get worse as compared to OPT, while the number of tuples generated by OPT approaches the number of tuples in the EXACT sliding window join. Similar effects were observed for other input distributions (not shown here). As a tendency the lines for EXACT and OPT get closer as the domain size increases from 10 to 200, while the lines for PROB and OPT become more and more separated.

This interesting phenomenon can be explained as follows. As the domain size increases, the distribution for a given Zipf parameter becomes less skewed in the sense that a longer heavy tail reduces the maximum frequency values. At the same time the larger number of low-frequency join attribute values leads to a lower probability of these tuples encountering a matching join tuple in the other stream within the window. The optimal offline algorithm OPT, which is able to see the future, can select and keep those *few* small probability tuples which will be generating output within the window, and can safely discard the rest of them. PROB does not know the future, and hence has only more opportunities to "make a mistake" because the same probability mass is now distributed over more tuples. On the other hand the gap between OPT and EXACT becomes smaller since OPT mostly discards tuples which have no or only few matching tuples arriving within the window. In fact, as we can see in Figure 11 the graphs for OPT and EXACT meet already for $M = w$, i.e., at only 50% of the memory required to guarantee exact computation. This implies that by picking the right tuples to be kept in memory, the exact result can be obtained with only 50% of the memory usually required to hold the contents of the current window.

4.5 Real Life Dataset Experiments

For our real life dataset experiments, we used weather data available at [16] which consists of cloud measurements organized by month and collected over several years by thousands of sensors located all over the globe, in land and water. The data sets contain measurements such as the year, month, day and time the reading was taken, the location of the sensor, the brightness of the sky, cloud cover, solar altitude and others. For our experiments, we chose the readings taken by the land sensors in the month of September over two consecutive years (1985, 1986). These datasets contain just over a million tuples each. The attributes of interest were the latitude and longitude information, pinpointing the location of the sensor taking the readings. We then performed a streaming sliding window join on the two datasets using the latitude and longitude attributes to identify sensors located physically near each other. We divided locations on the earth into a 18 by 36 square grid consisting of 10 degrees of latitude and longitude each, and mapped sensors falling in the same grid cell to the same location for the purpose of the join. (There were about 650 distinct location values). Such a join query could potentially be used to aggregate information gathered from sensors located in the same region, with the join window enforcing that the matched readings are taken at nearby points in time.

To avoid startup effects, the warmup time was set to 10000. The size of the join window was set to 5000, and a plot of number of tuples output by the various join approximation methods with varying memory size is shown in Figure 7. This graph closely resembles those obtained for smaller stream lengths and window and domain sizes (see Figures 3, 4). The performance of the variable and fixed memory allocation versions PROB and PROBV were almost identical, indicating that the two input streams had similar data distributions. This is made more apparent by the graph in Figure 8 which indicates that the memory allocation remained more or less at the 50-50 mark (2500) for

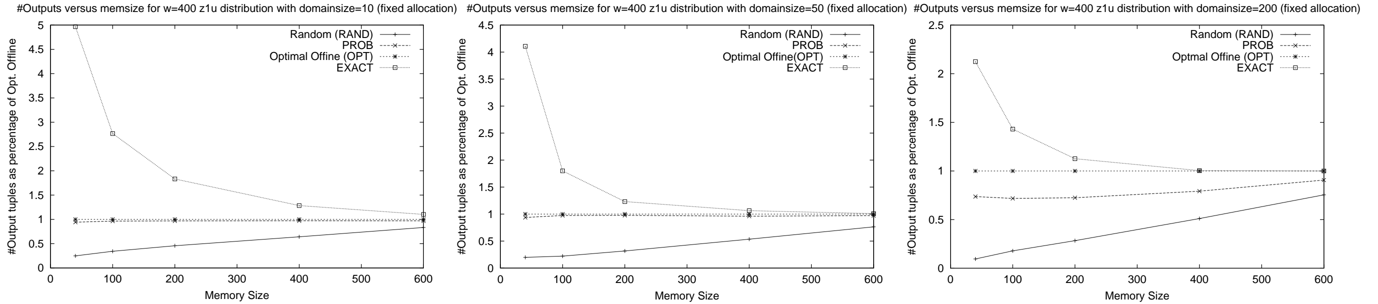

Figure 9: Domain size 10 **Figure 10: Domain size 50** **Figure 11: Domain size 200**

the entire duration of the join. The PROB and PROBV methods again performed very well, generating over 90% of the output tuples produced by EXACT with only 50% of the memory. Note that we did not include a comparison to OPT because the time and memory requirements of the flow solver exceeded available resources.

The frequency table of the data values in the dataset was used to estimate the probabilities of the next incoming tuple. While in practice such a table/histogram may not be available and needs to be constructed over the data stream on the fly, the aim of our experiments was to compare the quality of the various join approximation techniques, and not the quality of the estimator predicting the attribute value of the next arriving tuple. Clearly, all online algorithms would benefit from having a good predictor, and given a bad predictor of future tuples, no online algorithm would be able to perform well. Note that in the experiments performed, the frequency tables were not updated as the relations were streaming by (say by subtracting off the counts of the data values seen so far).

4.6 Discussion of Experimental Results

We presented a comparison of the performance of several join approximation techniques for computing sliding window joins with limited memory. We showed the efficacy of the fixed and variable memory versions of the PROB technique on both synthetic and real life datasets. PROB clearly improves on the state of the art, i.e., random tuple shedding, and it performs almost as well as the optimal offline algorithm OPT-offline. As seen from the graphs, the performance of PROB (measured in terms of the number of join tuples output) degrades gracefully as the amount of available memory decreases, and it performs exceptionally well for skewed data, typically producing over 90% of the total output with as little as 50% of the memory (compared to the EXACT algorithm). In cases where both the input streams have join attribute values distributed uniformly at random, no online algorithm can do better than evict tuples at random since there is no way of taking advantage of values that arrive with higher probability. In cases where there is a large disparity in the skew of the two joining streams, the variable memory allocation approaches fare better than the fixed memory approaches. Overall, the experimental results suggest that the PROB and PROBV approaches perform exceptionally well and the number of tuples produced are close to OPT on both synthetic and real life datasets. In addition, the techniques decay gracefully as the amount of memory available decreases.

5. RELATED WORK

There is a growing interest in the general field of data stream processing. The general issues and some architectures for stream processing systems are discussed in [2, 3] (Stanford's STREAM) and [6] (Aurora). The latter introduces the notions of QoS-optimization based on QoS graphs for response times, tuple drops, and values produced. Our work is the first to examine in detail efficient drop-based QoS optimization for sliding window joins.

The only other work on efficiently processing sliding window joins is a recently published paper by Kang et al. [21]. Their techniques use a unit-time based cost model, selecting the join implementation and memory allocation for the two input streams according to their arrival rates. Load is shed by simple random eviction. Our work addresses more complex memory allocation problems based on the *values of single* tuples (hence the notion of semantic load shedding introduced in this paper). In that respect our work is also related to uniform sampling over joins [8]. However, our goal is to maximize the *accuracy* of the output, not its statistical properties (e.g., being a uniform sample).

Adaptive query processing systems like Telegraph [18], NiagaraCQ [9], sensor database systems [5] and adaptive techniques as proposed in [7, 20, 24] aim at providing the best possible query performance in continuously changing environments like the Internet. Our algorithms can also adapt to changing amounts of available resources and hence can be used in adaptive query processing systems.

Arasu et al. [1] examine when stream queries can be computed with bounded storage. Joins in general might require unbounded memory, hence in data stream systems they are restricted to computation over sliding windows as discussed earlier.

For maintaining online data stream statistics, e.g., in order to compute the tuple priorities for join memory replacement, some of the recently proposed stream aggregation approaches could be applied. Recent work includes [10, 12, 14, 15, 27].

6. CONCLUSIONS AND FUTURE WORK

We discussed the problem of approximately computing sliding window joins for data streams. We defined the problem space of fine grained tuple-based join approximations using different set error measures, and we examined the MAX-subset measure in depth and gave optimal offline and good online algorithms for sliding window joins. We believe that this work shows that *semantic load shedding*, i.e., adapting to resource shortages by dropping tuples based on their val-

ues, is clearly superior to random load shedding at the cost of a small overhead for maintaining simple stream statistics.

We also proposed the novel Archive-metric as a new measure for evaluating the performance of join algorithms over sliding windows for data stream systems with support by an archive.

This work only examined part of the overall problem space, and many problems remain open. Developing efficient algorithms for the Archive-metric is part of our future work. We will also examine the other join processing models, especially the slow-CPU case. Another interesting direction of future work is to examine how multiple queries can efficiently share resources and how to combine semantic load shedding with the join implementation selection in [21].

Acknowledgements. We thank Rohit Ananthakrishna, Al Demers, and Alin Dobra for helpful discussions.

7. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 221–232, 2002.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[3] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, 2001.

[4] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. Int. Conf. on Mobile Data Management (MDM)*, pages 3–14, 2001.

[6] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.

[7] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.

[8] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274, 1999.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390, 2000.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 635–644, 2002.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[12] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 79–88, 2001.

[13] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 58–65, 2001.

[15] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. ACM Symp. on the Theory of Computing (STOC)*, pages 471–475, 2001.

[16] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. http://cdiac.esd.ornl.gov/ftp/ndp026b, 1996.

[17] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[18] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[19] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 174–185, 1999.

[20] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–310, 1999.

[21] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2003.

[22] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.

[23] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2002.

[24] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[25] R. T. Rockafellar. *Network flows and monotropic optimization*. John Wiley & Sons, 1984.

[26] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databaases. In *Proc. Int. Conf. on Computer Vision (ICCV)*, pages 207–214, 1998.

[27] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[28] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 2 edition, 1979.