Semantic Approximation of Data Stream Joins *

Abhinandan Das Johannes Gehrke Mirek Riedewald
Department of Computer Science
Cornell University
{asdas,johannes,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 make two main contributions. First, we define the problem space by discussing architectural models for data stream join processing and surveying suitable measures for the quality of an approximation of a set-valued query result. Second, we examine in detail a large part of this problem space. More precisely, we consider the number of generated result tuples as the quality measure, and we propose 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.

Keywords: Data streams, approximation algorithms, approximate query processing, load shedding models, semantic load shedding, set approximation error metrics, join processing, sliding windows

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 [11, 16, 17]. 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 [3, 7, 25]. These systems are specifically designed 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. Take for example an application that monitors the traffic at two routers. The first router generates a stream `R(sourceID, destinationID, length, time)`, the second router produces the analogous stream S with the same schema. To detect traffic patterns, the monitoring application determines for each incoming packet on one router, which packets that arrived within the last 2 hours on the other router have the same source address. This is a *continuous* query, i.e., a long-running query, which computes a join between the two streams by matching tuples with the same `sourceID`, restricting the set of possible join partners to a window of size 2 hours. The Stream Query Repository [37] contains further examples of queries involving joins, e.g., for online auctions, network traffic monitoring, and military logistics. Joins are needed whenever information from several streams has to be combined in order to compute correlations or to match events. Notice that these joins are not restricted to foreign-key joins. For instance computing the correlation between data streams typically involves a many-to-many matching of tuples (as in the example above).

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 since each tuple in one stream could potentially match each tuple in the other. To address this problem, 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 [3]. 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.

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. In addition, 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. Even if the peak load was known, it is often orders of magnitude higher than the average load. Hence guaranteeing resource availability for peak loads would require the system to keep most of its resources idle

*The authors are supported by NSF grants IIS-0330201, CCF-0205452, and IIS-0133481, and by a gift from Microsoft. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

during normal operation. Even parallelizing stream queries (cf. [35]) therefore does not guarantee availability of sufficient resources at all times. Resource limitations can have two effects. First, for streams with high arrival rates, the *CPU* might not be fast enough to process all incoming tuples in a timely manner. Second, 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 [5]. In data stream processing systems, one way of approximating query answers is to *shed load*, for example, by 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 [28]. The second proposes to include QoS specifications which assign priorities to tuples and then shed those with low priority first [7]. 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 [9].

Semantic Join Approximation. We address the problems outlined above by introducing the notion of *Semantic Join Approximation (SJA)*. In SJA, 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 join approximation avoids the problems described above by intelligently selecting which tuples to drop and when they should be dropped — all in order to minimize the error in the query output. This paper contains an in-depth study of this problem for the case of sliding window joins. We also discuss a related scenario (*static join* described below) where semantic join approximation can lead to great improvements. This scenario is similar to a join with *tumbling* window semantics [7], i.e., consecutive windows have no tuples in common.

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. [29]), 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 [30], the goal of the system is to transmit as little data as possible to extend the sensors' lifetime. Hence before sending the actual data, each sensor transmits a compact summary, e.g., a histogram, of the join attribute distribution of its measurements. The proxy uses this summary information to determine which data to request from the different sensors (the requests are also compact summaries, e.g., a list of join attribute values indicates that the sensor should send all measurements with these values). 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).

Contributions of this Paper. In this paper we give an in-depth examination of semantic join approximation for data streams. We present novel algorithms for approximating set-valued join results at tuple granularity. Our optimal offline 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 possible error measures and 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 (Section 3) and optimal offline algorithms and very fast lightweight online heuristics for the sliding window join problem. (Section 4)
- We evaluate our algorithms on a large set of synthetic and real-life data. (Section 5)
- While most of our techniques target equi-joins, we show how to extend the approaches for sliding window joins to joins with general predicates and ε-joins which are common in spatial databases. (Section 6)

A discussion of related work (Section 7) and a summary and outlook to future work conclude this article (Section 8).

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 Problem Definition

Let R and S be two data streams that contain a common attribute J, which is selected as the join attribute. The equi-join $R \bowtie S$ 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 $r.J = s.J$. For the static join problem the streams are finite (relations), since the sensors can only keep a limited amount of data, e.g., temperature readings from the last 24 hours. Hence the static join is equivalent to the join between relations with restricted access to the input tuples.

A sliding window join is a long-running query. 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 $t - w < i \leq t$. Whenever a new tuple $r(t)$ arrives at time t in stream R, this tuple generates output with all matching partners $s(i)$ in the current window $t - w < i \leq t$ (similar for newly arriving S-tuples). Hence the overall output of the join from time t_1 to time t_2 is

$$\bigcup_{t=t_1}^{t_2} \bigcup_{i=t-w-1}^{t} ((r(t) \bowtie s(i)) \cup (s(t) \bowtie r(i))) \ .$$

Note that the operators have bag-semantics, i.e., produce multisets and do not remove duplicates.

The sliding window join as defined above applies to windows whose size is specified in time units. For simplicity of presentation we will focus on this type of join and furthermore assume that time is discrete and that at each time instant t exactly one tuple $r(t)$ and $s(t)$ arrive on each stream. Notice that our techniques easily generalize to tuple-based window definitions and asynchronous tuple arrival, including the arrival of several tuples at the same time. We will also discuss how to generalize to cases where R and S have different window sizes and where the window size is allowed to change over time.

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 [26]. 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 [24, 39]. 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) [34] 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). If $X \subseteq Y$ it trivially evaluates to 0.

The Match And Compare (MAC) [26] 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 recently introduced a novel "error" measure, the *Archive-metric* (ArM) [12]. ArM is relevant for semantic load *smoothing*, i.e., for applications that can not afford to discard any input tuples during periods of high load. Instead these applications store the tuples which could not be processed in archives. During low-load periods the tuples from the archive can be used to refine approximate results which were obtained during periods of high load. Due to space constraints ArM is not discussed here.

2.3 Models for Window Joins

In order to compute the *exact* result of a sliding window join, the join operator has to keep track of the contents of the current window, i.e., the latest tuples from each stream. Hence for window size w the *internal state* of the operator consists of $2w$ stream tuples. Furthermore, to compute the exact result, the operator should process tuples at least as fast as they arrive.

As long as the system has sufficient memory and

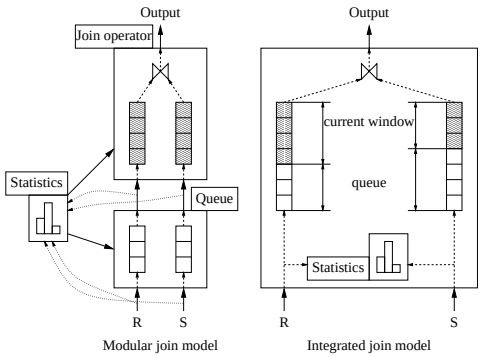

Figure 1: Join processing models

CPU resources, incoming tuples are added instantly to the join memory and remain there until they *expire*, i.e., are not part of the current window any more. In case of resource shortage, tuples either have to be dropped from memory before they expire (and hence spend less than w time in memory) or even never reach the join memory (dropped before participating in the join). In the following we discuss how to model the different cases.

Modular vs. Integrated. In practice the processing of the join is affected by fluctuations in resource availability and load. Hence in addition to the internal join memory for storing the current window tuples the join processing unit needs a queue that buffers incoming tuples and a statistics unit that gathers statistics about resources and load. The queue smoothes local fluctuations, while the statistics unit provides input for the join approximation algorithm for deciding how many and which tuples to evict from the internal memory or from the queue.

We identify two general models—*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. 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. 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 knowledge of both memory contents, but 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 [28]). 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. Notice that in practice one would still add a small queue to deal with fluctuations in resource availability and load, but conceptually this queue is irrelevant. Whenever the queue fills up beyond a certain threshold, the system could switch to the slow-CPU case which is discussed below.

In general we model the fast CPU case such that the join has internal memory 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 before it expires, due to lack of sufficient memory). Hence an arriving tuple will *always* be seen by the join.

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 [7]. 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.

Notice that we do not explicitly introduce another independent system resource dimension for available **memory**. For the fast CPU model the sufficient memory case would not be of interest since there is no resource shortage. For the slow CPU case the sufficient memory case is vacuous for the following reason: Since the join processes tuples at a slower speed than they arrive, any amount of available (queue) memory at some point would be exceeded. Hence for a given amount of memory the slow CPU case will always be an insufficient memory case as well.

3 Static Join Approximation

Before discussing approaches for sliding window joins over data streams in the next section, we present hardness results and algorithms for the static join approximation case. These results are important in their own

right, e.g., for the sensor network scenario we discussed before, or in the case of approximating tumbling window joins with limited memory. In addition to that they provide useful insights for the hardness of the problem of efficiently approximating joins of two or more relations, which can be viewed as the base case for approximating joins of data streams. For example, the slow CPU case is a generalization of the static join approximation (see Appendix E).

3.1 Problem Definition

We consider the following two relation (static) join approximation 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 restrictions, a total of k tuples need to be dropped from the input. 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. We measure the approximation quality by using the MAX-subset measure since most of the popular and common set approximation error and set similarity measures actually reduce to MAX-subset for our problem (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 maximized.

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 nodes $n_A \in V_A$ and $n_B \in V_B$ if the corresponding tuples 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). Figure 2 shows an example bipartite graph representing the join graph of an equality join between two relations A and B. In the figure, nodes with the same shape denote tuples having identical values for the join attributes, while nodes with different shapes represent tuples having different values on the equality join attributes.

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 $K(m, n)$, as shown in Figure 2. 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 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 it produced, 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.

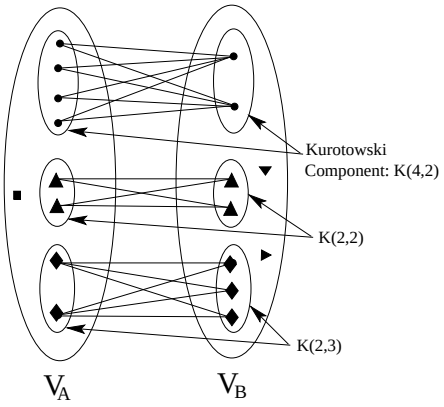

Figure 2: Equality join as a bipartite graph

We are now ready to state two 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.

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 and k_B tuples from the two joining relations respectively as opposed to k tuples overall. While in the following discussion, we switch between the two formulations for ease of exposition, in most cases the algorithms and hardness proofs developed for one case can easily be extended to the other. We will point out explicitly when this is not true.

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 version 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.

3.2 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. In the following we will first show an optimal solution for the special case of $c = 1$. Then the general algorithm is presented.

Lemma 1. *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. Then $C_{m,n}(p)$ is given by: (w.l.o.g., assume $m \geq n$)*

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

Proof. See Appendix A. □

For $c > 1$ we obtain the optimal solution by using dynamic programming based on the following observation. 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. Formally, 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 $i > 1$, $0 \leq j \leq k$:

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

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

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 (cf. formula above for $T(i, j)$, which takes the maximum over at most $j \leq k$ terms). Thus the overall running time of the algorithm is $O(c \cdot k^2)$ and space requirement is $O(c \cdot k)$. 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 and 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 need to spend at least $O(k)$ for processing the two relations. Also note that the algorithm is polynomial in the sizes of the input relations.

3.3 Fast 2-Approximation Algorithms

We present two fast polynomial-time 2-approximations—one is applicable to the formulation where one needs to delete k_A and k_B nodes from the two bipartite partitions respectively, while the other is applicable in the case where one needs to delete k nodes overall.

3.3.1 Node Degree Greedy (NDG) Algorithm

Suppose we are interested in *deleting* k_A and k_B nodes from the two partitions respectively (primal version). To select the k_A nodes to be deleted from the A-partition, we sort the nodes in this partition by their degrees in ascending order. We then select the k_A lowest degree nodes for deletion. Similarly, we select the nodes with the k_B lowest degrees in the B-partition for deletion. We can select the nodes from the B-partition either based on their degrees in the original bipartite graph, or based on the graph obtained after the k_A nodes and corresponding edges from the A-partition have been deleted. It is easy to see that the latter approach never does worse than the former one. However, both in the worst case provide a 2-approximation.

Theorem 1. *The node degree greedy algorithm provides a 2-approximation to both the primal and the dual version of the (k_A, k_B)-truncated join approximation problem simultaneously. Its running time and space requirement are $O(c \log c)$ and $O(c)$, respectively.*

Proof. See Appendix B. □

3.3.2 Average Degree Greedy (ADG) Algorithm

Consider the primal formulation where we need to *delete* k nodes overall. For a Kurotowski component $K(m,n)$, define its average degree to be $m \cdot n/(m+n)$. Sort the Kurotowski components by their average degree, and select the p lowest average degree components, where p is such that the first $p-1$ components contain less than k nodes, while the first p components contain at least k nodes. We then delete each of the first $p-1$ components completely. The remaining nodes are deleted from the last component using the optimal strategy for deleting nodes from a single component (cf. Lemma 1). By choosing the *highest* degree nodes for *retention* this algorithm can be easily extended to solving the dual formulation of the k-truncated join approximation problem.

Theorem 2. *The average degree greedy algorithm provides a 2-approximation to both the primal and dual versions simultaneously. Its running time and space requirement are* $\mathrm{O}(c \log c)$ *and* $\mathrm{O}(c)$*, respectively.*

Proof. See Appendix C. □

Note that in general for primal-dual algorithms, it is not necessarily the case that a 2-approximation to the primal also is a 2-approximation to the dual, and vice versa. Both the Average Degree and Node Degree Greedy algorithms, however, guarantee 2-approximations to *both* primal and dual at the same time and thus provide stronger approximation guarantees than just 2-approximations to any one of them.

Note that since the input is logarithmic in k (or k_A, k_B) simply using the strategy of trying out all possible 2-partitions of k (there are $k+1$ of them) does not yield a polynomial time reduction from the 'delete k_A, k_B' problem to the 'delete k overall' problem.

3.4 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, and 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 approximation problem.

Theorem 3. *The 3-relation static join approximation problem is NP-Hard.*

Proof. See Appendix D. □

Corollary 1. *The m-relation static join approximation problem is NP-Hard for any* $m \geq 3$.

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.

4 Dynamic Window Join Approximation

In Section 2 we defined the problem space for computing sliding window joins by introducing its dimensions (integratedness of model, resource bottleneck, approximation error measure). Examining each point in this space in detail is beyond the scope of this paper. Instead we will present an in-depth analysis for a large and important subspace. More precisely, we will restrict our attention to the fast CPU model and the MAX-subset error measure. Notice that this covers a large part of the problem space. As mentioned before, for a fast CPU system integrated and modular architecture are equivalent. Furthermore, recall that most of the popular and common set approximation error and set similarity measures reduce to MAX-subset for our problem. We present optimal offline and efficient online algorithms.

4.1 Fast CPU and Offline

We are now considering the sliding window join as discussed in Section 2.3. 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 cannot be modeled. However, this idealized algorithm is used to provide the baseline for measuring the efficiency of any real online algorithm over *any given finite 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.

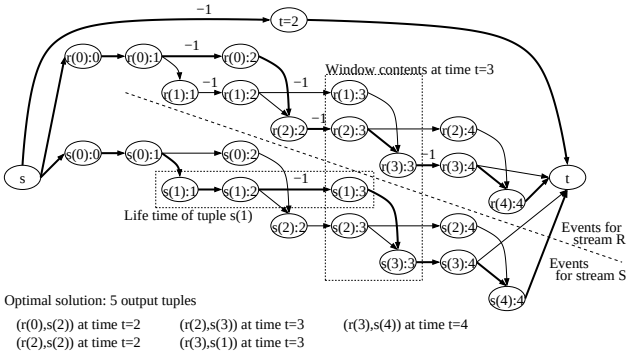

Figure 3: Example flow graph

4.1.1 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, all other arcs have cost factor 0. Solving a *min-cost* linear flow problem will then find the 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 input "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 3. 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 $r(0) : 1 \rightarrow r(1) : 1$ 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 $r(0) : 1 \rightarrow r(0) : 2$ 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 input sequence all nodes that correspond to tuples in the current window are connected to the sink t.

In Figure 3 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.

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 still in memory, an output tuple is produced. This is modeled by arc $r(0) : 1 \rightarrow r(0) : 2$ which has cost factor -1. In Figure 3 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 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 easy. 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 3 such arcs would be $r(0) : 1 \rightarrow s(1) : 1$, $s(0) : 1 \rightarrow r(1) : 1$, $r(0) : 2 \rightarrow s(2) : 2$, $r(1) : 2 \rightarrow s(2) : 2$, 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.

4.1.2 OPT-offline Algorithm

It is not hard to 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 reenter the memory and only tuples in memory produce output. Furthermore it is ensured by construction that no tuple can produce output after it has expired.

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 4. *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 [33], 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 $O(n^2 m \log n)$ [19] or $O(nm \log n \log m)$ [1], 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 $(O((wN) \cdot (wN + M) \cdot \log(wN) \cdot \log(wN + M)))$.

4.2 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. We present two deterministic heuristics, PROB and LIFE, which are intuitively appealing and extremely simple and lightweight.

4.2.1 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 of arrival of a tuple with the same join attribute value on the *other* stream. For instance for $r(i)$ the priority is $p_S(r(i))$. Whenever the buffer overflows, PROB ejects the tuple with the lowest priority. Ties are broken by giving higher priority to the tuple that arrived later. Note that the newly arriving tuple is also a candidate for eviction.

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.

Assuming that the arrival probabilities can be estimated fairly accurately (if this is not possible, any online strategy will perform poorly and hence one could use random-tuple eviction instead) there is another intuitive reason why PROB performs well: the probability of those inputs which cause PROB to perform poorly is low. There are two main scenarios where one

expects PROB (or any online algorithm for that matter) to perform poorly:

- PROB drops a tuple when in fact it should have retained it since many joining partner tuples arrive soon afterwards on the other joining stream. However, the fact that PROB did not retain the tuple implies that it had comparably low partner tuple arrival probability, and hence the probability that several partner tuples of the discarded tuple arrive in close succession while very few partner tuples arrive for the retained tuples is low.
- The second scenario where PROB performs poorly arises when PROB retains a tuple in memory for a long time and very few or no partner tuples arrive for that tuple on the other stream (S) during this interval. In this case, since PROB has retained the tuple in memory for a fairly long time, it implies that the partner arrival probability for the retained tuple is comparably high. Hence the likelihood of the event that very few partner tuples arrive on S for this retained tuple while many more partners arrive for some other tuple that arrived on stream R and was dropped is low.

PROB can be used both for fixed memory allocation between R and S, and 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.

A practical issue is to compute the values of $p_R()$ and $p_S()$. Any online algorithm that with high probability produces more output than an algorithm that randomly replaces tuples in memory needs at least "good" approximations of these probabilities. One possibility to estimate $p_R()$ and $p_S()$ is to assume that their future distribution will be similar to the distribution in recent history (similar to approaches for online caching). 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). Such statistics over data streams are usually maintained by default in most data stream processing systems since they constitute a basic primitive and can be shared between multiple queries. Note that rather than an exact knowledge of partner tuple arrival probabilities, PROB only needs information corresponding to the *relative* ordering between the partner tuple arrival probabilities in order to evict the correct tuples. Also, the performance of PROB is not affected much if the approximate summary statistics interchanges the relative ordering of the partner tuple arrival probabilities of tuples with 'similar' partner tuple arrival probabilities, and hence the estimates of the value probabilities do not need to be very precise. The performance of PROB is fairly stable as long as the priorities of tuples with vastly different partner tuple arrival probabilities are correctly ordered by the summary statistics used.

4.2.2 LIFE Heuristic

The LIFE heuristic is also based on estimates of the $p_R()$ and $p_S()$ values. However, LIFE aims at giving more weight to the remaining lifetime of a tuple. The priority of a tuple $r(i)$ with remaining lifetime t is computed as $t \cdot p_S(r(i))$. As with PROB, the LIFE heuristic ejects the tuple with lowest priority, with ties being broken by giving a higher probability to a tuple that arrived later. Like PROB, LIFE can be used for both fixed and variable memory allocation between R and S-tuples.

Note that for large window size newly arriving tuples 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. A better approach would be to use the *expected* lifetime of a tuple for computing the priority. This will be addressed in future work (note that more complex algorithms which are based on expected lifetime are also less robust against errors in estimating $p_R()$ and $p_S()$).

5 Experiments

5.1 Static Join Approximation

In this section we compare the performance of the Average Degree Greedy (ADG) approximation algorithm with the optimal dynamic programming based algorithm (henceforth called OPTDP). We consider the dual version of the static join approximation problem (maximize number of retained output tuples). Recall that ADG guarantees a 2-approximation, and that the running times of ADG and OPTDP are $O(c \log c)$ and $O(ck^2)$, respectively. In this section, we evaluate the actual running times of ADG and OPTDP as well as the approximation quality of ADG.

The input data is generated synthetically. For each relation we use a Zipfian distribution with skew parameter z to generate the number of nodes in each of the c disjoint Kurotowski components. The same skew parameter is used for generating data in both bipartite partitions, however, both distributions are generated independently. All running times reported in the experiments below are the averages of at least 5 runs on a 1.8 GHz Intel PIII machine running RedHat Linux 9.

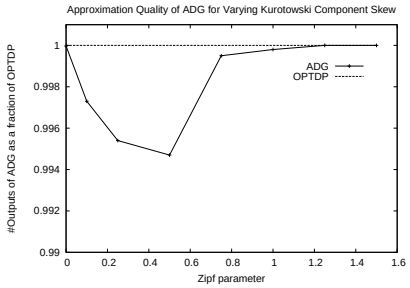

Figure 4: ADG Approximation

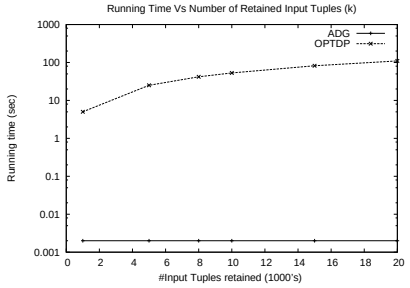

Figure 5: Running time vs. k

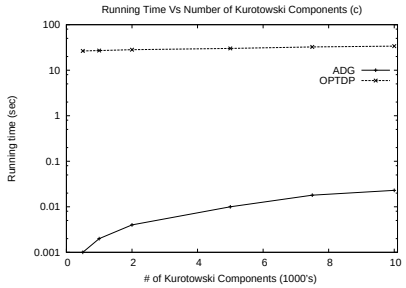

Figure 6: Running time vs. c

5.1.1 Approximation Quality

Figure 4 compares the performance of ADG and the optimal algorithm OPTDP for varying degrees of skew in the distribution of the Kurotowski component sizes. In these experiments, the size of the joining input relations was $50K$ tuples each, and the number of retained input tuples (k) was set to $5K$, while the number of Kurotowski components (c) was $1K$. As can be seen from the figure, ADG performs extremely well, producing over 99.5% of the output tuples produced by OPTDP for varying degrees of skew (note that the y-scale starts at 0.99). For moderate skew, the performance of ADG is marginally worse than OPTDP, but for very low and very high skew ADG produces almost the same number of output tuples as OPTDP. This behavior at low and high skews can be explained as follows: At very low skew, most of the Kurotowski components are of the same size, and hence choosing one over the other does not affect the join output by much. At very high skew, some Kurotowski components are clear 'winners', producing a large number of output tuples, and these are selected by both OPTDP and ADG. These components dominate the total number of join tuples produced, and thus the performance of ADG is very close to OPTDP. Similar results were observed for other values of k and c. In the above experiments, the average running time of OPTDP was 26 seconds, as compared to 0.002 seconds for ADG.

5.1.2 Running Time

Figures 5 and 6 show (on a logscale) the running times of ADG and OPTDP as the number of input tuples to be retained (k) and the number of Kurotowski components (c) is varied. In these experiments, the size of the joining relations was $50K$ tuples each, and the Kurotowski components were generated using a Zipfian distribution with skew parameter 0.5. In the experiments in Figure 5, the number of Kurotowski components (c) was $1K$, while in the experiments in Figure 6, the number of retained tuples (k) was held constant at $5K$.

As can be seen from Figure 5, the running time of ADG is three to four orders of magnitude less than that of OPTDP. The running time of ADG is independent of k and remains constant at 0.002 sec. As we showed analytically, the running time of OPTDP in the worst case increases quadratically with k. In practice the growth was almost linear. (This linear growth is not obvious in the figure because of the logarithmic y-scale.)

The effect of varying c on running times is shown in Figure 6. As expected, the running time of OPTDP increases linearly with c, while ADG's running time grows at a slightly faster rate ($O(c \log c)$). However, the running time of ADG, which is always below 0.02 seconds, still remains much smaller than OPTDP whose running time varies between 23 to 33 seconds.

5.1.3 Discussion of Experimental Results

The aim of the above experiments was to bring out the tradeoffs involved between running time and approximation quality of the OPTDP and the ADG algorithms. As can be seen from the graphs, the running time of ADG is orders of magnitude below that of OPTDP, while the number of outputs produced is almost identical. Hence ADG provides a viable alternative for scenarios where faster responses are required or where the processors are already heavily loaded, without sacrificing approximation quality by much. In addition, the space requirement of ADG, which can be implemented 'in-place' and thus requires only $O(c)$ space, is lower than that of OPTDP ($O(ck)$).

5.2 Dynamic Window Join Approximation

We perform an extensive evaluation of the sliding window join approximation techniques suggested in Section 4.2 on both synthetic and real life datasets. We compare the performance 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 4.1. 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 [19]. 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 4.1.2). Hence for all the experiments involving comparison with OPT, we restrict the input length to 5600 tuples. Note that all algorithms store the first $M/2$ tuples from each stream in memory and therefore output the same set of resulting join tuples for these tuples. Hence in order to prevent such startup effects from dominating the number of output tuples produced, 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. Since in our experiments, the join window size was at most 800, the chosen input length of 5600 tuples guarantees that for any window size w, at least 4000 tuples are processed after the warmup phase of $2w$. In our experiments it turned out that larger input size does not affect the validity of the conclusions drawn from the graphs obtained on these streams.

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 [22]. The input streams had the same tuple arrival rates, with a tuple arriving on each stream at every timestep.

For all experiments the probabilities $p_R()$ and $p_S()$ used by the heuristics (cf. Sections 4.2.1 and 4.2.2) are set according to the actual distribution over the *whole* input stream (determined empirically). Hence at each moment in time both PROB and LIFE are in fact using approximate values which might differ considerably from the true "local" distribution for a given window.

5.2.1 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 7 and 8 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] The input data streams in Figures 7 and 8 are generated from uncorrelated Zipf distributions with parameter 1. The domain size of the data was set to 50 (see Section 5.2.4 for justification).

As expected, the behavior of the various algorithms 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 7 and 8, 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. The poor performance of LIFE is caused by the way it computes the tuple priorities based on remaining lifetime, and not expected lifetime (cf. Section 4.2.2). Even though $w = 400$ and $w = 800$ are fairly small window sizes from a practical point of view, they are large enough to give even tuples with low $p_R()$ and $p_S()$ values a high enough priority to replace better tuples which have a lower remaining lifetime.

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.

5.2.2 Effect of Skew

If both 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 9 confirms our prediction, showing the performance of the different algorithms for a window size of 400 when both the input streams have a uniform data distribution. 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 7. 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.

As can be seen from Figures 7, 8 and 9, the LIFE heuristic does only marginally better than RAND for

[1] Strictly speaking only $M = 2w - 2$ is needed because of the extra memory cells provided by the two input buffer cells in the fast CPU model (cf. Section 2.3).

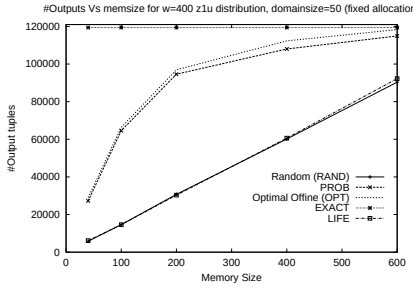

Figure 7: Window size $w = 400$

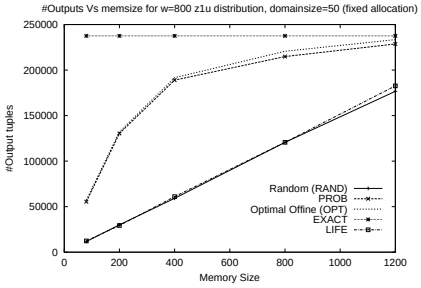

Figure 8: Window size $w = 800$

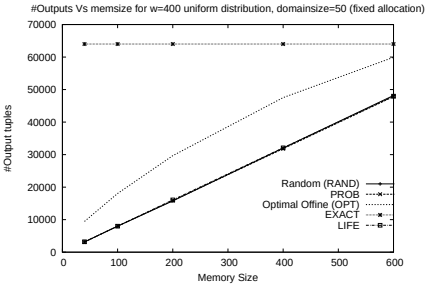

Figure 9: Uniform input

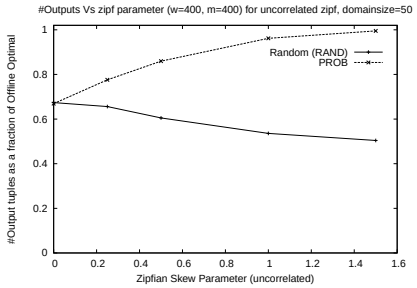

Figure 10: Uncorrelated Zipf

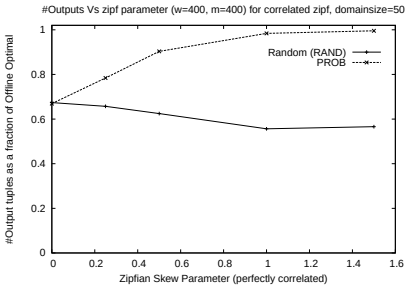

Figure 11: Correlated Zipf

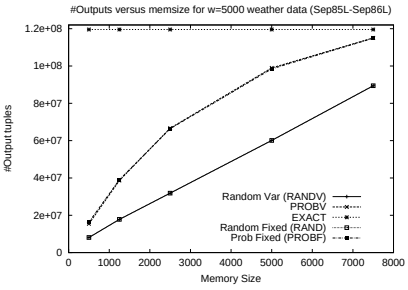

Figure 12: Weather data: Performance

reasons explained earlier. Similar behavior was obtained for other data, hence the LIFE approach is not included for comparison in the remaining experiments in this section.

Figures 10 and 11 nicely bring 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 10, the distributions of the two input streams are uncorrelated, while in Figure 11, the two Zipfian distributions are perfectly correlated, in the sense that the high (low) frequency values on one stream are also high (resp. low) frequency values on the other stream. 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.

The graphs for both cases are similar, indicating 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 Zipfian distribution experiments have been performed for uncorrelated streams, the results obtained hold for correlated and "anti-correlated" (i.e. the high frequency values on one stream are the low frequency values on the other stream and vice versa) distributions as well. Note however, that correlation does affect the *total number* of output tuples generated by the joins.

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).

5.2.3 Variable Memory Allocation

The experimental results discussed so far were obtained for a fixed memory allocation of $M/2$ to each of streams R and S. In the following set of experiments, we allow the memory allocated to each stream for storing tuples to vary, while keeping the total amount of memory constant. Hence an incoming R-tuple can replace an S-tuple in memory and vice versa. This obviously generalizes the fixed memory approach, and therefore improved output results are expected, especially if the join attribute value distributions in the streams are different. In the following, we will use PROB, RAND, and OPT for the fixed memory algorithms, and PROBV, RANDV, and OPTV for their variable memory coun-

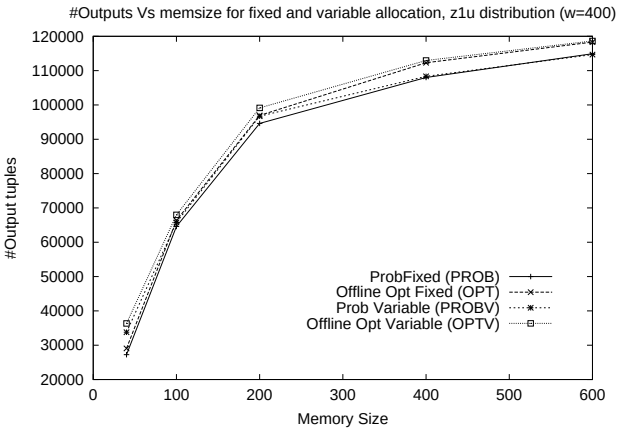

Figure 13: Zipf(1.0) both

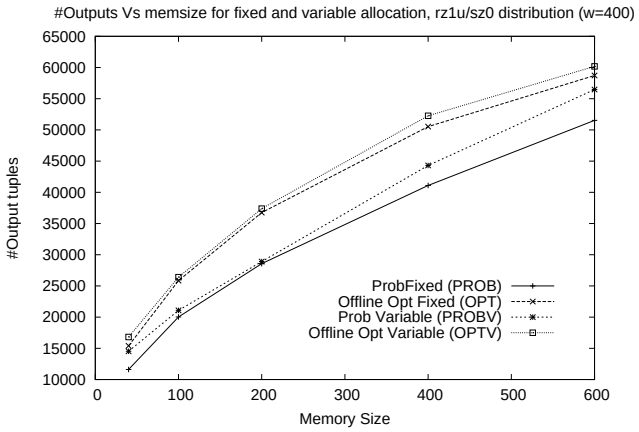

Figure 14: Zipf(1.0)/uniform

terparts.

We report the number of output tuples generated as a function of total memory size for window size 400. In Figure 13 both the input streams have an uncorrelated Zipf(1.0) distribution. In Figure 14 one of the streams (R) has Zipf(1.0) distribution while the other has uniform distribution. As can be seen from the graphs, when the two arriving streams have the same distribution, the performance of the fixed and variable algorithms is very similar (cf. Figure 13). However, Figure 14 shows that the performance difference increases (up to 10% of output size) as the disparity in the skew of the distributions of the input streams increases. This behavior is both expected and reasonable. If the two input streams have similar distributions, there is no reason to allocate more memory to one relation as compared to the other. Hence the fixed and variable memory allocation versions of the algorithms behave similarly. As the disparity in the skew of the input streams increases, it makes more sense to allot more memory to the relation having a greater number of high *priority* tuples arriving as compared to the relation where all tuples have the same moderate priority.

Figures 15 and 16 show the allocation of memory between the two streams done by the variable memory algorithms for various data distributions as a function of time. As expected, based on our discussion above, when the two input streams have the same distribution (Figure 15), both the relations get about half the memory. When the two joining relations have different skews, the relation with the higher skew is allotted more memory (up to a share of 85%).

5.2.4 Effect of Domain Size

Figures 17, 18 and 19 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 phenomenon can be explained as follows. As the domain size becomes larger, the distribution for a given Zipf parameter is less skewed due to a longer heavy tail, which makes the maximum frequency smaller as the size of the domain increases. This brings us closer towards the uniform distribution case, for which we know that all algorithms perform equally bad, at par with RAND. Since the effects of changing the domain size are similar to the effects of varying the skew, in our experiments with synthetic data we fixed the domain size (to 50) and varied the skew.

The relative improvement of OPT versus EXACT with increasing domain size is caused by the increasing number of tuples with low arrival probabilities in the tail of the Zipf distribution. This results in a higher percentage of tuples for which no matching partner tuple will arrive within the window. Since OPT knows the future, it can safely discard these tuples without much effect on the result size. In fact, as we can see in Figure 19 the OPT and EXACT lines meet at $M = w$, indicating that we can compute the exact join result with only 50% of the memory that would be required in general to guarantee the exact result.

5.2.5 Real Life Dataset Experiments

For our real life dataset experiments, we used weather data available at [22] which consists of cloud measurements organized by month and collected over several

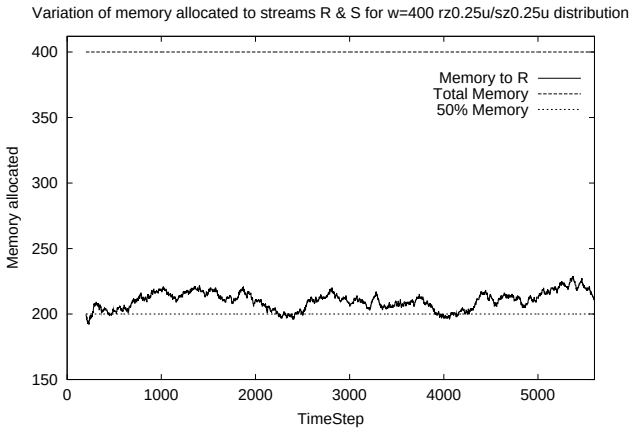

Figure 15: Zipf(0.25) both

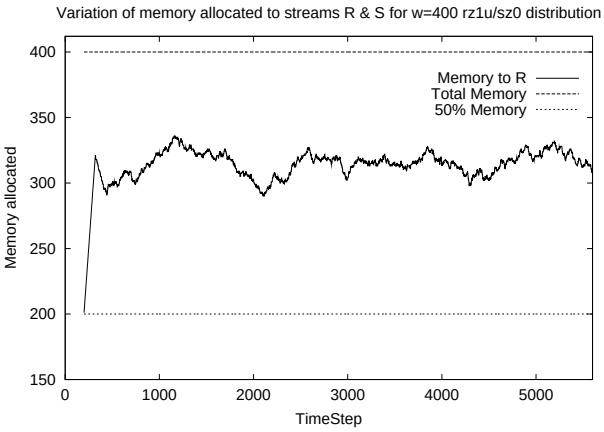

Figure 16: Zipf(1.0)/uniform

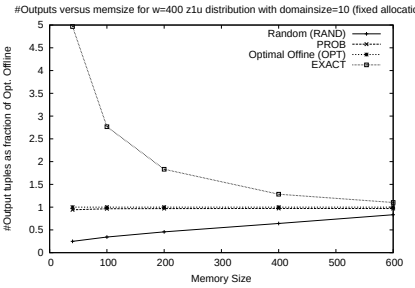

Figure 17: Domain size 10

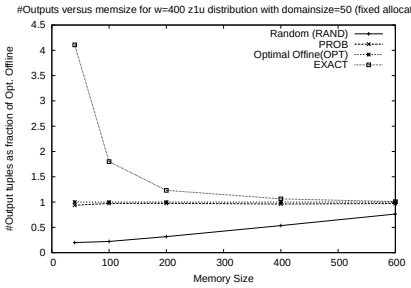

Figure 18: Domain size 50

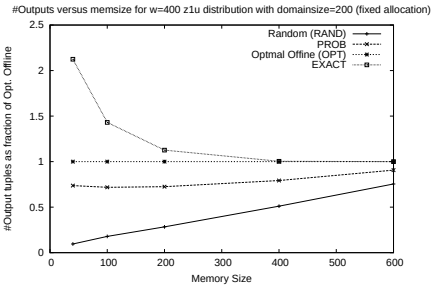

Figure 19: Domain size 200

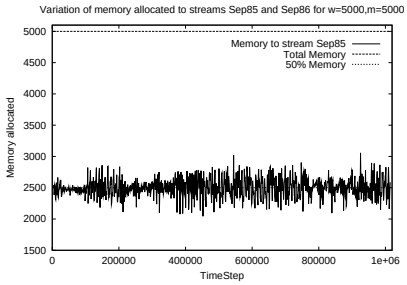

Figure 20: Weather data: Memory allocation

years by thousands of sensors located all over the globe, in land and water. The data sets contain measurements such as the time the reading was taken, sensor location, sky brightness, 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. We 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 examine correlations between values measured by sensors in the same region, with the join window enforcing that the matched readings are taken at nearby points in time. For PROB the frequency table of the data values in the *whole* dataset was used to estimate the probabilities of the next incoming tuple.

The size of the join window was set to 5000, and a plot of the number of tuples output by the various join approximation methods with varying memory size is shown in Figure 12. This graph closely resembles those obtained for smaller stream lengths and window and domain sizes (see Figures 7, 8). 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 20 which indicates that the memory allocation remained more or less at the 50-50 mark (2500) for 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.

5.2.6 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 eviction, and it can perform 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. 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.

The question of how to split the available memory between the buffer space for join processing and any summary structures is an important and complex one that is beyond the scope of this article. However, we would like to note that summary statistics about the frequencies of the various domain values occurring in each stream are usually a basic primitive required for answering and optimizing virtually any type of query (not just joins) over the corresponding streams. Hence this summary space can be *shared* by several queries, similar to the summary statistics stored in a relational database system.

6 Generalization of the Dynamic Window Join Approximation

In Section 4 we introduced an optimal offline algorithm and proposed two online heuristics. All three were developed for standard equi-joins, i.e., where two tuples join if the values of the join attribute(s) are equal. Furthermore we assumed that the tuples of streams R and S arrive in synchrony, one per time unit. The window size w and available memory M were fixed. These assumptions obviously will not be satisfied by most applications. In this section we show how to generalize the approaches.

6.1 Extensions of OPT-Offline

The flow network model for OPT-offline (cf. Section 4.1.1) has the great advantage of enabling the efficient computation of a baseline for evaluating the approximation quality of *any real online algorithm*. In addition to that it is fairly flexible.

6.1.1 More General Application Parameters

Varying window size. Variations of the window size w over time can easily be incorporated into the flow graph. Consider the wide box in Figure 3 which highlights the lifetime of tuple $s(1)$. Instead of all such boxes having the same number of horizontal arcs (2 in the example), varying window size can be reflected by a correspondingly larger or smaller number of these arcs. For instance, if the window size temporarily shrinks to $w = 2$ at time $t = 1$, then node $s(1) : 3$ would not exist and $s(1) : 2$ would only have a single outgoing edge to $s(2) : 2$. It is interesting to note that we can even model a different lifetime for each single tuple.

As an extreme case, we can also model *unbounded* joins, i.e., joins with no window restrictions. This can be done by adding the corresponding horizontal arcs for each tuple and each time instant after this tuple has arrived. For instance in Figure 3 there would be horizontal arcs to new nodes $r(0) : 3$ and $r(0) : 4$, indicating that $r(0)$ never expires.

Varying amount of memory. Dealing with resource fluctuations, in this case memory, is of paramount importance for applications. We show how such fluctuations can be modeled by the flow graph. Recall that the source node s has M outgoing arcs (ignoring the path for output generated by matching input tuples which arrive at the same time, e.g., the top path through node $t = 2$ in Figure 3). Through these arcs a total flow of M is pushed, modeling the amount of available memory. Allowing variable memory implies that memory is added or removed at certain time instants. The flow graph can reflect these changes by introducing additional source and sink nodes that adjust the flow for these time instants.

Consider time instant $t = 3$ in Figure 3 (tall box). Suppose that at this time not only the new tuples $r(3)$ and $s(3)$ arrive, but also the memory is reduced by m units. Figure 21 shows the modified section of the flow graph. We model the event by adding a new *sink* node sink(3) with demand m. Furthermore, nodes $r(1) : 3$, $r(2) : 3$, $s(1) : 3$, and $s(2) : 3$ now each have an additional outgoing arc to sink(3). Hence at time $t = 3$ exactly a flow of m will be redirected from the tuples which are in memory at that time to the sink node, reducing the flow and hence the modeled amount of memory as desired.

Increasing amounts of memory can be modeled similarly, but requires more care. If m more memory slots are available beginning at time $t = 3$, we add a new *source* node source(3) with supply m. This node has outgoing arcs to the next $\lceil m/2 \rceil$ incoming pairs of R- and S-tuples with the next higher arrival times. In the example for $m \leq 2$, node source(3) is connected to $r(3) : 3$ and $s(3) : 3$. For $2 < m \leq 4$ it is connected to $r(3) : 3$, $s(3) : 3$, $r(4) : 4$, and $s(4) : 4$, and so on. This way of modeling larger memory increments by more than two slots might appear artificial, but it

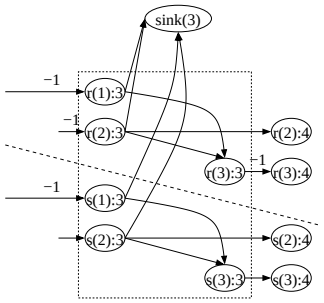

Figure 21: Memory shrinks at time 3

perfectly models reality. Even if more than 2 memory slots are added at a certain time, there are only two newly arriving tuples to fill them. Hence the $m-2$ remaining additional memory slots are essentially irrelevant in that moment and can only be filled at later times, two tuples at each time instant.

There is one more subtlety to be considered when modeling increasing memory. We illustrate it with an example. Assume at time t the memory increases by 10 slots, followed by a decrease by 16 slots at time $t+2$. In this case the increment by 10 would be distributed over 5 consecutive time instants, hence overlapping with the decrement at time $t+2$. In such cases of artificial overlap we simply compute the aggregate change in memory and only add the corresponding nodes. In the example there would be a source node with supply 4 and four outgoing arcs to the tuples arriving at times t and $t+1$. For time $t+2$ we add a sink node with demand 10 (=16-10+4). Notice that this accurately reflects the real situation. At times t and $t+1$ the effect of the added 4 memory slots is equivalent to the effect of adding 10 new slots at time t (recall that only at most 2 slots can be filled per time unit). From time $t+2$ we have ensured that the memory reflects the overall loss of 6 memory slots (compared to time $t-1$). Interestingly, if we have several phases of memory increases overlapping each other, this simply leads to a cascading effect of adding arcs from source nodes to tuples at later time instants.

Asynchronous tuple arrival. Instead of exactly one tuple per time unit, we allow any number of tuples to arrive at a given time unit (including zero tuples). Notice that a sliding window defined based on time units might contain a varying number of tuples over time.

This case can be incorporated into the flow graph very easily. Consider the tall box in Figure 3 which highlights the window contents at time $t=3$. Tuples $r(3):3$ and $s(3):3$ are the new tuples that arrive at that time. If asynchronous arrival is possible, there might be no such tuple, or several of them. We can model this by adding the appropriate number of nodes for each stream and by connecting the nodes from previous tuples to them. In the example $r(1):3$, $r(2):3$, $s(1):3$, and $s(2):3$ would have additional outgoing arcs to these new tuples, or would directly point to the tuples arriving at time $t=4$.

Notice that we can even model continuous time domains. All we need for our model is the arrival order of the tuples. As before we can model both fixed and variable memory allocation between R and S.

6.1.2 Other Join Operators

The equi-join is arguably the most commonly used join operator in database systems. In general any predicate over the schemas' attributes could be used to match the tuples of streams R and S. Common examples are join conditions which are conjunctions of terms of the form $r.J_1\theta s.J_1$, $r.J_1\theta c$, and $s.J_1\theta c$ where r and s are tuples arriving in streams R and S, respectively; c denotes a constant, and $\theta \in \{<, \leq, =, \neq, \geq, >\}$. Another popular join operator from spatial applications is the spatial- or ε-join that matches tuples r and s if their distance is less than or equal to ε.

Our flow model can handle *all* these join operators. In fact, it can handle any subset of the cross-product (including the cross product itself) of two data streams. Hence our model can by definition handle any join [32]. Notice that for each element of the cross-product of R and S, limited to the contents of a window of size w, there is a corresponding horizontal arc in the flow graph. In Figure 3, arc $r(2):2 \rightarrow r(2):3$ corresponds to the pair $(r(2), s(3)$, because it models that $r(2)$ remained in memory until time 3 when tuple $s(3):3$ arrives. In general, arc $r(i):j \rightarrow r(i):j+1$ corresponds to the pair $(r(i), s(j+1))$, similarly for the corresponding horizontal S-arcs. If the tuple pair satisfies the join condition, the arc has cost factor -1, otherwise zero.

6.1.3 Other Approximation Error Measures

The MAX-subset measure assigns the same benefit value to all output tuples of the join. In practice certain tuples might be more valuable than others. For instance, in network monitoring systems, packages that might indicate a denial of service attack are of higher importance than others. Our model is general enough to assign a different priority to *each single* output tuple. Hence we can not only handle applications where tuples with the same value have the same priority, but also cases where the priority of an output tuple is defined by an arbitrary function $\text{val}: R \times S \rightarrow \mathbb{R}$, i.e., a function that assigns a real number independently to each possible output tuple. In the model we simply add the cost factor $-\text{val}((r(i), s(j+1))$ to arc $r(i):j \rightarrow r(i):j+1$, similarly for the horizontal S-arcs.

The measure we are optimizing now is to approximate the sliding window join with limited memory such that the sum of the priorities of the join tuples output by the approximation is as large as possible. This enables our model to support value-based QoS specifications [7].

6.1.4 Discussion of the OPT-Offline Extensions

With the above extensions we can model any application with synchronous or asynchronous tuple arrival, discrete or continuous time, fixed or varying window size, windows defined based on time or number of tuples, fixed or varying memory size, fixed or varying memory allocation between R and S, any type of join condition, and any approximation quality measures that assign weights (priorities) to output tuples. In fact the only relevant property we could think of, which can not be modeled by the flow graph are tuple priorities that depend on the output history, i.e., if certain previous output tuples have been generated or not.

All generalizations we proposed for the flow graph model retain the polynomial complexity and hence enable efficient computation of the offline benchmark.

6.2 Extensions of the Online Heuristics

Recall that the PROB and LIFE heuristics assign priorities to input tuples in the current window in order to decide which tuples to drop in case of resource shortage. Both heuristics generalize to **varying window size**, **varying amount of memory**, and **asynchronous tuple arrival** (including a continuous time domain) in a straightforward manner. For example, if the window size changes, the behavior of PROB does not change (except for expiring tuples at the appropriate time), while LIFE modifies the priority estimation calculation where the remaining lifetime is computed by taking into account the new window size. For *unbounded* joins, i.e., joins with no window restriction, LIFE would be meaningless (since no tuple ever expires), but PROB would just work in the same manner as before. Note that over time all memory slots will fill up with high-priority tuples.

If other join operators or other approximation error measures are used, one simply needs to change the priority computation accordingly. This is identical to the way we change the cost factor of horizontal edges in the flow model. The only difference is that here we do not know the future, hence the benefit of an input tuple is weighted by the probability of seeing a matching partner in the other stream.

Interestingly, the heuristics can even handle the case when priorities of current tuples depend on past output (or drops). All we need is some compact synopsis data structure that summarizes the relevant properties of the previous output. Here we can select from a variety of efficient stream synopsis techniques (see discussion in Section 7).

7 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 [3, 4] (Stanford's STREAM) and [7] (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. As such, our techniques can well be integrated into the Aurora query processor.

In a recent paper Kang et al. [28] propose a unit-time based cost model for selecting the appropriate implementation and memory allocation for the join of 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 join approximation introduced in this paper). In that respect our work is also related to uniform sampling over joins [9]. However, our goal is to maximize the *accuracy* of the output, not its statistical properties (e.g., being a uniform sample).

In a recent paper Tatbul et al. [36] propose several load shedding approaches for data stream processing. The goal is to find the optimal position and drop rate of operators which are inserted into the query plan of the Aurora query network. This work is complementary to ours in the sense that our techniques specifically target load shedding for joins and provide results about the theoretical foundations of approximating static and dynamic joins.

Hammad et al. [23] propose new techniques for scheduling for shared window joins over data streams. In other recent work on joins over data streams Viglas et al. [40] propose multi-way join operators to speed up the generation of a prefix of the overall join output. Shah et al. [35] examine how to process stream queries in parallel on a cluster.

Adaptive query processing systems like Telegraph [25], NiagaraCQ [10], sensor database systems [6] and adaptive techniques as proposed in [8, 27, 31] 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. [2] 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 [13, 14, 18, 20, 21, 38].

8 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 join approximation*, i.e., adapting to resource shortages by dropping tuples based on their values, is clearly superior to random load shedding at the cost of a small overhead for maintaining simple stream statistics.

We showed how our techniques for sliding window join approximation, both the optimal offline benchmark algorithm and the online heuristics, can be extended to capture almost any possible application scenario, including a general class of approximation quality measures, asynchronous tuple arrival, continuous time domain, any join operator, windows defined based on tuples or time, and variations in window size and resources. For the static join case we provided hardness results and optimal and approximate algorithms for the MAX-subset measure.

Nevertheless this work only examined part of the overall problem space, and many problems remain open, e.g., developing efficient algorithms for the Archive-metric and examining 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 join approximation with the join implementation selection in [28]. Also, for complex queries which involve joins and other operators, new approximation techniques are required. We could easily integrate aggregate operators like `SUM` on the join output, and selections on the join inputs. On the other hand, for instance optimizing MAX-subset for each single node of a join tree will not optimize MAX-subset overall. Examining complex queries therefore is another challenging problem which will be addressed as part of our future work.

Acknowledgement

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

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 623, 2002.

[12] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 40–51, 2003.

[13] 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.

[14] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–72, 2002.

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

[16] M. N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[17] J. Gehrke, editor. *Special Issue on Data Stream Processing*, volume 26 of *IEEE Data Engineering Bulletin*, 2003.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 297–308, 2003.

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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2003.

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

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

[35] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 25–36, 2003.

[36] N. Tatbul, U. Çetintemel, S. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 309–320, 2003.

[37] Stanford STREAM Team. Stream query repository. http://www-db.stanford.edu/stream/sqr.

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

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

[40] S. D. Viglas and J. Burger J. F. Naughton. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 285–296, 2003.

A Proof of Lemma 1

We show the following. 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. Then $C_{m,n}(p)$ is given by: (w.l.o.g., assume $m \geq n$)

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

Proof. 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. 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. The formula for $C_{m,n}(p)$ follows immediately. □

B Proof of Theorem 1

We show that the node degree greedy algorithm provides a 2-approximation to the k_A, k_B-truncated join approximation problem at a cost of $O(c \log c)$ and $O(c)$ for running time and space, respectively.

Proof. Let p_1 and p_2 denote the sum of the degrees of the k_A and k_B lowest degree nodes from the A- and B-partitions in the original graph, respectively. The NDG algorithm deletes at most $p_1 + p_2$ edges. On the other hand, any algorithm that deletes k_A and k_B nodes from the A- and B-partitions, respectively, (and hence also the optimal algorithm) must delete *at least* $\max\{p_1, p_2\}$ edges.

A similar argument shows that the solutions obtained using the above approaches are also 2-approximations to the dual version (i.e., the number of edges retained is at least half the number of edges retained by an optimal solution retaining $m-k_A$, $n-k_B$ nodes, where m and n are the number of tuples in relations A and B respectively).

The running time of NDG is dominated by the cost of sorting the components, i.e., $O(c \log c)$. The algorithm can be implemented 'in-place', and thus needs only $O(c)$ space, viz. the space needed to store the inputs. □

C Proof of Theorem 2

The running time of the ADG algorithm is dominated by the cost of sorting the components, i.e., $O(c \log c)$. The algorithm can be implemented 'in-place', and thus needs only $O(c)$ space.

We show now that the average degree greedy algorithm provides a 2-approximation to both the primal and dual versions simultaneously. We give separate proofs of 2-approximations to the primal and dual versions respectively. Before presenting the actual proof, we first state some useful observations.

Lemma 2. *There exists an optimal solution for the primal and dual k-truncated join approximation problem which is of the following form: For all but at most one Kurotowski component, either* all *the nodes in the Kurotwoski component are retained or* none *of the nodes in the Kurotwoski component is retained.*

Proof. (Sketch) The main idea is to show that any optimal solution to the primal or dual k-truncated join approximation problem can be 'converted' to the required form without reducing the number of retained edges in the resulting graph. Assume there is an optimal solution with two or more *partially selected components*, i.e., components where some of their nodes are retained and others are not. Pick any two such partially selected components $K_i(m_i, n_i)$ and $K_j(m_j, n_j)$ (let m_i', n_i', m_j', n_j' denote the respective numbers of retained nodes) and progressively increase the number of nodes to be retained from one and correspondingly decrease the number of nodes to be retained from the other until either the former is completely retained or the latter has no node left. A standard case-by-case analysis (based on the relative sizes of m_i', n_i', m_j', n_j') shows that there is always a way to 'transfer' the nodes chosen for retention from one partially selected component to another such that none of the node-transfer operations decreases the number of retained edges and at the end, either the former component has no nodes selected for retention, or the latter component is completely selected. The whole process is repeated until there is at most one partially selected component left. Since none of the node-passing operations from component to component ever decreases the number of retained edges, the proof is complete. □

Lemma 3. *If the solution found by the ADG algorithm has no partially selected component, then this solution is optimal.*

Proof. (Sketch) For a given solution of the dual k-truncated join approximation problem, assign to each node of this solution an *amortized value* equal to the average degree of the component it belongs to. Then the total number of edges retained is the sum of the amortized values assigned to all retained nodes. We argue that the k nodes retained by ADG have the highest

possible amortized values amongst all solutions retaining k nodes. Notice that a retained node in a partially selected component $K(m,n)$ with m', n' nodes retained from the two partitions of the component respectively is assigned an amortized value of $m' \cdot n'/(m'+n')$. It can be shown that $m' \cdot n'/(m'+n') \leq m \cdot n/(m+n)$. Hence nodes of components which are not selected by ADG can never achieve a higher amortized value than nodes selected by ADG. It follows that retaining nodes from any Kurotowski component which is not selected by ADG would result in introducing nodes with lower amortized values, and hence a lower number of retained edges than the one achieved by ADG. The argument for the primal version is similar, with the amortized value for a deleted node in a partially deleted component being defined as the ratio of the number of edges deleted from the component to the number of nodes deleted from the component. □

We are now ready to state the proof for Theorem 2:

Proof. Based on Lemmas 2 and 3 we only need to perform the proof for the case that the solution given by the ADG algorithm has exactly one partially selected component. For a Kurotwoski component $K(m,n)$, let its average degree be $d = mn/(m+n)$. W.l.o.g. assume that $m \leq n$. Then it is easy to see that

$$m/2 \leq d \leq min(m, n/2) \tag{1}$$

Consider a Kurotwoski component $K(m,n)$ with average degree d. If p nodes are deleted from such a component, then it is can be shown that the number of nodes deleted is always *at least* $p * d$ and if p nodes are retained from this component, then the number of nodes retained is always *at most* $p*d$. We first show the 2-approximation for the primal version of the problem:

Lemma 4. *Primal version: The number of nodes* deleted *by the ADG algorithm is at most twice the number of nodes deleted by an optimal algorithm.*

Proof. Let $\text{del}_{\text{nodes}}$ be the number of nodes deleted in the partially selected component $K(m,n)$ by ADG. Let $\text{del}_{\text{edges}}^{\text{complete}}$ be the number of edges deleted in the (zero or more) components completely deleted by the ADG algorithm. Notice that $\text{del}_{\text{edges}}^{\text{complete}}$ may be 0 if no component is completely deleted. Let $d = m \cdot n/(m+n)$ be the average degree of the partially selected component, and let $\hat{d}$ be the amortized value assigned to *deleted* nodes of the partially selected component, as described above. Thus if $\text{del}_{\text{edges}}^{\text{partial}}$ edges are deleted by the ADG algorithm from the partially selected component, then $\hat{d} = \text{del}_{\text{edges}}^{\text{partial}}/\text{del}_{\text{nodes}}$.

Let C(Greedy) denote the number of edges deleted by ADG and let C(OPT) denote the number of edges deleted by an optimal algorithm. With the above notation we obtain:

$$\begin{aligned} C(\text{Greedy}) &= \text{del}_{\text{edges}}^{\text{complete}} + \text{del}_{\text{edges}}^{\text{partial}} \\ &= \text{del}_{\text{edges}}^{\text{complete}} + \hat{d} \cdot \text{del}_{\text{nodes}} \\ &\leq \text{del}_{\text{edges}}^{\text{complete}} + m \cdot \text{del}_{\text{nodes}} \end{aligned}$$

The last inequality follows from the fact that if nodes are deleted optimally from a *single* Kurotwoski component then each deleted node gets an amortized value of at most m and hence $\hat{d} \leq m$. From Equation 1 follows that

$$\text{del}_{\text{edges}}^{\text{complete}} + m \cdot \text{del}_{\text{nodes}} \leq \text{del}_{\text{edges}}^{\text{complete}} + 2 \cdot d \cdot \text{del}_{\text{nodes}}$$

Hence,

$$C(\text{Greedy}) \leq \text{del}_{\text{edges}}^{\text{complete}} + 2 \cdot d \cdot \text{del}_{\text{nodes}} \tag{2}$$

Note that if $\text{del}_{\text{nodes}}$ nodes are deleted from a component $K(m,n)$ with average degree d, then the amortized value of each deleted node is always *at least* d, since the number of edges deleted is always at least $\text{del}_{\text{nodes}} \cdot d$. Using this, and comparing the amortized values of the deleted nodes in decreasing order in the greedy and OPT solutions, we can argue along the lines of the amortized value argument used above for Lemma 3, hence:

$$C(\text{OPT}) \geq \text{del}_{\text{edges}}^{\text{complete}} + d \cdot \text{del}_{\text{nodes}} \tag{3}$$

The conclusion that the ADG algorithm is a 2-approximation to the primal follows immediately from Equations 2 and 3. In fact, as can be seen, if the number of edges deleted from the *completely* deleted components in the greedy algorithm is $\text{del}_{\text{edges}}^{\text{complete}} > 0$, we have a $1 + d \cdot \text{del}_{\text{nodes}}/(\text{del}_{\text{edges}}^{\text{complete}} + d \cdot \text{del}_{\text{nodes}}) < 2$ approximation scheme to the primal. □

With similar arguments as for the primal we will now show the following lemma for the dual formulation of the k-truncated join approximation problem:

Lemma 5. *Dual version: The number of nodes* retained *by the optimal algorithm is at most twice the number of nodes retained by the ADG algorithm.*

Proof. The proof of the dual version is similar in spirit to the primal version, except that there are two cases to consider. Suppose that ADG retains nodes from j Kurotowski components and let $K(m_i, n_i)$ and d_i $(i = 1, \ldots, j-1)$ be the completely retained components and their corresponding average degrees in non-increasing order of average degree. Furthermore let $K(m_j, n_j)$ be the component *partially* retained by the greedy solution and let d_j be its average degree. Thus we have

$$d_1 \geq d_2 \geq \ldots \geq d_{j-1} \geq d_j \ . \tag{4}$$

For the following discussion assume w.l.o.g. that $m_j \leq n_j$. Using notation analogous to the primal case, let $\text{ret}_{\text{edges}}^{\text{complete}}$ be the number of edges in the components $K(m_1, n_1), \ldots, K(m_{j-1}, n_{j-1})$, and let $\text{ret}_{\text{nodes}}$ be the number of nodes *retained* in the component $K(m_j, n_j)$

which is partially selected by the ADG algorithm. We distinguish between the following two cases: (1) $\text{ret}_{\text{nodes}} < 2m_j$ and (2) $\text{ret}_{\text{nodes}} \geq 2m_j$. Analogous to the notation introduced earlier, let $\hat{d}_j$ be the amortized value assigned to *retained* nodes of the partially selected component, as described above. Thus if $\text{ret}_{\text{edges}}^{\text{partial}}$ edges are retained by ADG from the partially selected component $K(m_j, n_j)$, then $\hat{d}_j = \text{ret}_{\text{edges}}^{\text{partial}}/\text{ret}_{\text{nodes}}$. We will use $C(\text{Greedy})$ to denote the number of edges retained by ADG and $C(\text{OPT})$ to denote the number of edges retained by an optimal algorithm. **Case (1):** $\text{ret}_{\text{nodes}} \geq 2m_j$. The proof of this case is very similar to the primal version:

$$\begin{aligned} C(\text{Greedy}) &= \text{ret}_{\text{edges}}^{\text{complete}} + \text{ret}_{\text{edges}}^{\text{partial}} \\ &= \text{ret}_{\text{edges}}^{\text{complete}} + \hat{d}_j \cdot \text{ret}_{\text{nodes}} \\ &\geq \text{ret}_{\text{edges}}^{\text{complete}} + (m_j/2) \cdot \text{ret}_{\text{nodes}} \end{aligned}$$

The last inequality makes use of the following fact: If $2m_j$ nodes are optimally retained from $K(m_j, n_j)$, the amortized value assigned to each retained node is exactly $m_j/2$ (see Lemma 1). Thereafter each additional node retained in the component results in the retention of m_j additional edges, so that $\hat{d}_j \geq m_j/2$. From Equation 1 follows that

$$\text{ret}_{\text{edges}}^{\text{complete}} + (m_j/2) \cdot \text{ret}_{\text{nodes}} \geq \text{ret}_{\text{edges}}^{\text{complete}} + (d_j/2) \cdot \text{ret}_{\text{nodes}}$$

Hence,

$$C(\text{Greedy}) \geq \text{ret}_{\text{edges}}^{\text{complete}} + (d_j/2) \cdot \text{ret}_{\text{nodes}} \quad (5)$$

Similar to the primal case, it is easy to argue using amortized values that

$$C(\text{OPT}) \leq \text{ret}_{\text{edges}}^{\text{complete}} + d_j \cdot \text{ret}_{\text{nodes}} \quad (6)$$

The 2-approximation property of the ADG algorithm follows from Equations 5 and 6.

Case (2): $\text{ret}_{\text{nodes}} < 2m_j$. We consider the case when $\text{ret}_{\text{nodes}}$ is even. A similar argument can be used for the case when $\text{ret}_{\text{nodes}}$ is odd. The cost of ADG is

$$\begin{aligned} C(\text{Greedy}) &= \text{ret}_{\text{edges}}^{\text{complete}} + \text{ret}_{\text{edges}}^{\text{partial}} \\ &= \text{ret}_{\text{edges}}^{\text{complete}} + (\text{ret}_{\text{nodes}}/2)^2 \ . \end{aligned}$$

This follows from Lemma 1. As before we have

$$C(\text{OPT}) \leq \text{ret}_{\text{edges}}^{\text{complete}} + d_j \cdot \text{ret}_{\text{nodes}}$$

If $\text{ret}_{\text{edges}}^{\text{complete}} = 0$ (no component is completely retained by ADG) it can be easily shown that the ADG algorithm in fact finds the optimal solution. This is based on the observation that for $m = m_1 + m_2$ and $n = n_1 + n_2$ the Kurotwoski component $K(m, n)$ always has at least as many edges as $K(m_1, n_1)$ and $K(m_2, n_2)$ together. Hence assume $\text{ret}_{\text{edges}}^{\text{complete}} > 0$. Clearly, if $d_j \cdot \text{ret}_{\text{nodes}} \leq \text{ret}_{\text{edges}}^{\text{complete}}$, we have a 2-approximation. Hence suppose, for the sake of contradiction, that $d_j \cdot \text{ret}_{\text{nodes}} > \text{ret}_{\text{edges}}^{\text{complete}}$. Then,

$$\begin{aligned} d_j \cdot \text{ret}_{\text{nodes}} > \text{ret}_{\text{edges}}^{\text{complete}} &= \sum_{i=1}^{j-1} (m_i + n_i) \cdot d_i \\ &\geq \sum_{i=1}^{j-1} (m_i + n_i) \cdot d_j \end{aligned}$$

The last inequality follows from Equation 4. Thus if the ADG solution is not a 2-approximation, then we have

$$\text{ret}_{\text{nodes}} > \sum_{i=1}^{j-1} (m_i + n_i)$$

Let $s_i = (m_i + n_i)$ for $i = 1, \ldots, j-1$. For any $i \in \{1, .., j-1\}$ we have:

$$\text{ret}_{\text{nodes}} > m_i + n_i = s_i$$

For a given s_i (and hence a fixed $m_i + n_i$) we can obtain an upper bound on d_i and show that this bound leads to a contradiction. Formally:

$$\begin{aligned} d_i &\leq \max_{\{m,n \in \mathbb{Z}: m+n=s_i\}} \frac{m \cdot n}{s_i} \\ &\leq \frac{(s_i/2)^2}{s_i} \\ &= s_i/4 \\ &< \text{ret}_{\text{nodes}}/4 \\ &< m_j/2 \qquad \text{Condition of case (2)} \\ &\leq d_j \qquad \text{Equation 1} \end{aligned}$$

This contradicts Equation 4. □

Theorem 2 follows immediately from Lemmas 4 and 5. □

D Proof of Theorem 3

We show that the 3-relation static join approximation problem is NP-Hard.

Proof. We model a 3-relation join using a tripartite graph and then use a reduction from the *balanced biclique problem* [15].

The tripartite join graph representing a 3-way join is constructed exactly as the bipartite join graph for a two relation join. In this case, we have three partitions, one for each relation with each partition having a node for every tuple of the corresponding relation. Suppose that the 3-way join involved is $A \bowtie B \bowtie C$. W.l.o.g. we assume that the joins are on different join attributes J_1 (for $A \bowtie B$ and J_2 (for $B \bowtie C$). We slightly overload notation and also refer to the partitions corresponding

to relations A, B and C as partitions A, B, C. There is an edge between a node in partition A and a node in partition B iff the corresponding tuples produce at least one output tuple in the 3-way join. Similarly, we add edges between nodes in partition B and partition C. However, for concise representation and to avoid redundancy, *we do not have any edges between nodes in partition A and nodes in partition C.* Notice that we can figure out whether or not a given tuple in partition A joins with a given tuple in partition C to produce output for the 3-way join by checking if there is a path from the node in partition A to the node in partition C or not. Note that a tuple from A may join with a tuple from B based on the 3-way join conditions, but the tuple from B may not join with any tuple from C. In that case this A-tuple joining with the B-tuple produces no output for the 3-way join and hence there is no edge between them.

With the formulation as a graph problem, we are now ready to show the reduction from the balanced biclique problem (decision version).

Problem 1 (Balanced Biclique Problem). *Given an arbitrary bipartite graph G and a an integer k, does G contain a $K(k,k)$ subgraph (i.e. a biclique with k nodes in each partition)?*

Given an arbitrary bipartite graph, we construct a corresponding tripartite graph representing the join of three relations as follows (see Figure 22). The A and C partitions of the tripartite graph are the same as those of the given bipartite graph. The third partition is constructed by inserting nodes on every edge in the original bipartite graph and putting all the newly inserted nodes into the same partition, i.e., partition B. Each of the nodes of partition B has degree 2, and is connected to the two nodes in partitions A and C that were originally connected by the edge onto which the newly added node of partition B was inserted.

Clearly the transformation of the given bipartite graph to the corresponding tripartite graph as explained above is polynomial in the number of edges in the original bipartite graph. To complete the hardness proof, we shall show that the decision version of the primal k_A, k_B, k_C-truncated join problem is NP-Hard. Thus, we need to show the following lemmata:

Lemma 6 (Correctness of Construction). *The tripartite graph obtained by using the above construction represents a 3-way join graph for any arbitrary bipartite graph.*

Lemma 7 (Reduction for k_A, k_B, k_C version). *The given bipartite graph has a $K(k,k)$ subgraph iff the tripartite join graph produces exactly k^2 join tuples when optimally (w.r.t.* MAX-subset *measure) retaining k, k^2, and k nodes from partitions A, B and C respectively.*

Strictly speaking, the decision version asks if there *exists* a solution which when retaining k, k^2, and k nodes from the three partitions respectively produces *at least* k^2 join tuples. But as we shall soon see, this is equivalent to the *optimal* solution producing *exactly* k^2 join tuples for join graphs where the B partition has only nodes of degree 2. For the variant of the problem where "k" nodes *overall* are retained, we will show the following lemma:

Lemma 8 (Reduction for k nodes overall version). *The given bipartite graph has a $K(k,k)$ subgraph iff optimally retaining $2k + k^2$ nodes (overall) from the tripartite graph produces k^2 join tuples.*

Lemma 6. To see why Lemma 6 is true, first note that every node in partition B has degree two, and hence imposes no structural constraints on the tripartite join graph (i.e., no set of edges implies the existence of another edge). More concretely, we can assume the join represented by the tripartite graph to be an equality join with the predicates $A.J_1 = B.J_1 \wedge B.J_2 = C.J_2$ where the schemas of relations A, B and C are $A(J_1)$, $B(J_1, J_2)$, $C(J_2)$ respectively. For each tuple of relation A, we associate the value a_i, $i \in \{1, \ldots, |A|\}$ such that $i \neq j \Rightarrow a_i \neq a_j$. Similarly, with each tuple of relation C, we associate the value c_i, $i \in \{1, \ldots, |C|\}$ such that $i \neq j \Rightarrow c_i \neq c_j$. For each edge which connects an a_i with a c_j in the original bipartite graph, we have a tuple in partition B with join attribute values (a_i, c_j) (cf. Figure 22). Since every tuple in partition B has degree exactly 2, its value is *uniquely* determined by the above construction. Thus the tripartite graph obtained by the transformation described above represents a 3-way join graph. □

Lemma 7. To show Lemma 7, first suppose that the given bipartite graph had a (k,k) subgraph. Suppose we need to optimally retain k, k^2, k nodes from partitions A, B, C respectively of the corresponding tripartite 3-way join graph. (This is the dual version of the k_A, k_B, k_C-truncated join problem.) We shall show that an optimal solution retains k^2 join tuples in the k, k^2, k truncated 3-way join. Consider the k, k nodes from partitions A,C respectively of the tripartite graph which correspond to the $K(k,k)$ subgraph in the original bipartite graph. These k, k nodes are, by construction, inter-connected via k^2 nodes in partition B. (Each of the k nodes in partition A is connected to each of the k nodes in partition C via a unique node in partition B). Then, if we retain this set of k, k^2, and k nodes from the partitions A, B and C respectively, the retained subgraph produces k^2 join tuples. Thus we have a solution which produces exactly k^2 join tuples. To see that this is an optimal solution, we only need to note that since the nodes in partition B all have degree exactly two, no solution to a k_A, k_B, k_C-truncated join on such a graph could ever produce more than k_B join tuples in the truncated result. This is because every

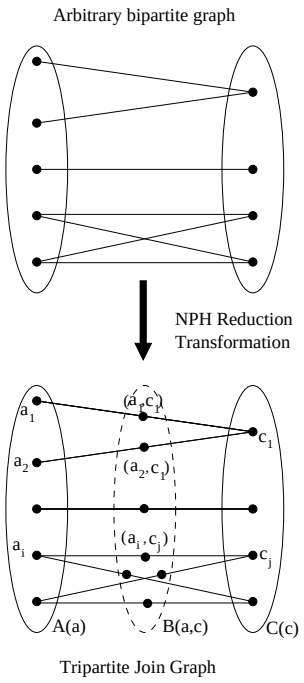

Figure 22: Example construction

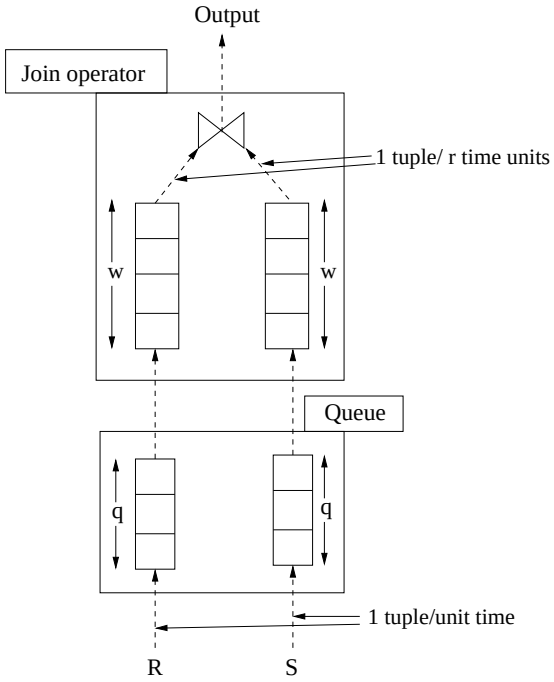

Figure 23: Slow CPU model

tuple in partition B can produce at most one output tuple in the 3-way (non truncated) join. Thus we have shown the "only if" direction of Lemma 7.

For the "if" direction, suppose that an optimal solution to the dual version of the k, k^2, k-truncated join approximation problem for the tripartite graph obtained from the given bipartite graph retains exactly k^2 join tuples. We claim that the original bipartite graph must contain a $K(k,k)$ subgraph. To see this, first note that the number of join tuples produced is exactly equal to the number of nodes retained from partition B. Since every node in B has degree exactly 2, this implies that every node in B produced exactly one output tuple. However, each node in partition B is connected to a distinct *pair* of nodes from relations A and C. Since we have exactly k nodes retained from each of the partitions A and C and these k nodes can correspond to at most k^2 distinct pairs, it must follow that each of these pairs appears in some join tuple in the truncated result. By construction, for a pair of tuples from A, C to appear in a join output, the nodes corresponding to the pair in the original bipartite graph must have been connected. Thus the k, k nodes in the original bipartite graph which correspond to the retained k, k nodes from partitions A and C in the optimal solution to the dual k, k^2, k-truncated join problem must form a $K(k,k)$ subgraph. Hence the original bipartite graph has a $K(k,k)$ subgraph. □

Lemma 8. For the variant where a total of "k" nodes need to be retained *overall*, we use the same transformation to a tripartite join graph as before. Suppose the original bipartite graph has a $K(k,k)$ subgraph. Arguing as before, it can be easily shown that there exists a solution retaining $2k+k^2$ nodes overall from partitions A, B and C (respectively retain k, k^2, k nodes from the 3 partitions as before) that produces k^2 join tuples.

Conversely, suppose there exists a solution retaining $2k + k^2$ nodes from partitions A, B and C which retains k^2 join tuples. We need to show that the original bipartite graph must have a $K(k,k)$ subgraph. We do this by proving that any solution retaining $2k + k^2$ nodes which produces k^2 join tuples must retain k, k^2, k nodes from partitions A, B and C respectively. Then, arguing as in the case of the k_A, k_B, k_C variant, we can conclude that the original bipartite graph must contain a $K(k,k)$ subgraph.

Consider any solution that retains p, $2k+k^2-p-q$, and q nodes from the partitions A, B and C respectively, hence a total of $2k + k^2$ nodes overall. Clearly, the number of join tuples produced can not exceed $p \cdot q$ and $2k+k^2-p-q$. This follows from the fact that the nodes of partition B each have degree 2, hence for each pair (a_i, c_j) there is at most one join result tuple, and each tuple in B can generate at most one join tuple. Since k^2 join tuples are retained in total, we obtain the following inequalities:

$$k^2 \leq p \cdot q \quad (7)$$

$$k^2 \leq 2k + k^2 - p - q \quad (8)$$

From Equation 8 we obtain $q \leq 2k-p$, therefore $p \cdot q \leq 2kp-p^2$. With Equation 7 it follows that $k^2 \leq 2kp-p^2$ which is equivalent to $(p-k)^2 \leq 0$. This can only be satisfied for $p = k$, which implies $q = k$, completing the proof of the lemma. □

With Lemmata 6, 7, and 8 the proof of Theorem 3 for the NP-Hardness of the 3-relation static join approximation problem is complete. □

E Static Case as a Special Instance of the Slow CPU Offline Case

Consider an instantiation of the slow CPU case where the join operator has sufficient memory to store all the tuples in a join window. As shown in Figure 23, let R and S be the joining streams, and let w be the size of the join window. Thus, we assume that the join operator has a memory size of $2w$ tuples.

Recall that in the slow CPU case, the join operator processes input tuples slower than they arrive. W.l.o.g., let us normalize the arrival rates as follows: We assume that input tuples arrive on streams R and S at the rate of 1 tuple per time unit, while the join operator processes 1 tuple every r time units ($r > 1, r \in \mathbb{R}$). Thus, as shown in Figure 23, the system needs to have input queues to buffer arriving tuples before they enter the join memory for processing. Let q be the total queue size in number of tuples. (Figure 23 shows a queue size of $2q$ with q being allocated to each stream, but in general the allocation of queue space amongst the streams may not be equal and may vary with time.) Note that since we are assuming that there is sufficient join memory available, once a tuple moves from the queue to the join memory, it is retained in the join memory until it is joined with all its arriving partner tuples which also made it to the join memory. Since the queue size is bounded, whereas the joining streams are potentially infinite or very large so as to make it infeasible to buffer all input tuples before they enter the join memory, our aim is to decide which tuples to retain in the queues so as to maximize some objective function. In our case, we shall be looking at the MAX-subset metric, where the goal is to maximize the number of join tuples produced.

Consider the offline version of the slow CPU case, where an algorithm knows the entire streams in advance, and must decide which tuples to retain in the join queues so as the maximize the number of join tuples produced. We claim that the static case discussed in Section 3 is a special instance of the slow CPU offline case. This can be seen as follows. Let n be the length of the joining streams ($n > w + q$). Consider the special case where $r > n$. In this case, the first $2w$ tuples (w from each stream) enter the join memory initially, but since the join operator takes $r > n$ time to process each input tuple, the entire input streams R and S arrive before a single tuple from the first $2w$ tuples has been completely processed. Hence an optimal offline algorithm which knows the sequence of input tuples on either stream must decide which q of the $2n - 2w$ tuples to retain so as to maximize the size of the join output. Also suppose that none of the first w tuples in each stream joins with any of the tuples in the other stream which arrive after time w. Hence none of the tuples which need to be buffered in the queue join with the tuples already in the join memory. Further consider the case where the input is such that none of the tuples which have the same join attribute value and appear in different streams arrive more than w time units apart. For this setup the problem that the optimal offline algorithm needs to solve is exactly the static case: Retaining q out of $2n - 2w$ nodes of a bipartite equi-join graph so as to maximize the number of retained edges.

Note that, more generally, we can replace the requirement $r > n$ by $r > w + q$ and still have a scenario at time r in which more than q tuples have arrived and need to be buffered in a queue of size q so as to produce maximum join output.

Thus any optimal solution to the slow CPU case must also be able solve the static case optimally.