Window Join Approximation over Data Streams with Importance Semantics*

Adegoke Ojewole
Dept. of Computer & Info. Science
University of Michigan – Dearborn
Dearborn, MI 48128, USA
ojewolea@umich.edu

Qiang Zhu
Dept. of Computer & Info. Science
University of Michigan – Dearborn
Dearborn, MI 48128, USA
qzhu@umich.edu

Wen-Chi Hou
Dept. of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
hou@cs.siu.edu

ABSTRACT

Load shedding techniques generate approximate sliding window join results when memory constraints prevent exact computation. The previously proposed random load shedding method drops input tuples without consideration for the number of outputs created, while the recently proposed semantic load shedding technique aims to produce the largest possible result set. We consider a new model in which data stream tuples contain numerical importance values relevant to the query source and seek to maximize the importance of the approximate join result. We show that both random load shedding and semantic load shedding are sub-optimal in this situation, while the techniques presented in this paper satisfy the objective function by considering both tuple importance and join attribute distributions. We extend the existing offline semantic approximation technique to make it compatible with our objective function and show that it is less space and time efficient than our new optimal offline algorithm for small and large join memory allotments. We also introduce four efficient online algorithms, which are quite promising in maximizing the importance of the approximate join result without foreknowledge of input streams.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Query Processing

General Terms
Algorithms, Management, Performance

Keywords
Data streams, importance semantics, sliding window join, approximation algorithms, load shedding

1. INTRODUCTION

Research in data stream processing is motivated by the important application domains in which data naturally occurs in the form continuous streams. Examples of these applications include weather monitoring via sensor networks [4], life signs monitoring in hospitals, vehicle tracking via a global positioning system (GPS) or through a digital radio service, internet traffic monitoring [12], or transaction log analysis [7].

Data stream processing poses challenges which cannot be overcome by directly applying traditional DBMS techniques. Firstly, it is impossible to store unbounded streams in their entirety. Secondly, recently arrived data stream elements may be more relevant than older data. Thirdly, standard blocking operators cannot be used, as they may indefinitely delay output production. Further, data stream management systems (DSMSs) are subject to real time processing constraints. To satisfy these requirements, DSMSs may produce approximate results.

We consider these problems in the context of the join operator. The familiar blocking join operator must be adapted to operate in a streaming environment because it would require infinite time and storage to compute the join result over a pair of unbounded streams.

Data stream joins are computed incrementally and continuously, with new result tuples being generated and streamed away as matching input tuples arrive [17]. To bound a streaming join's memory requirement, the criterion of exactness is altered to require that the join operates on a finite prefix of the input streams. While several variations are possible, such as fixed and landmark windows [13], we consider sliding windows (which we refer to simply as windows), where both endpoints conceptually move over the input stream, allowing in the newest element and displacing the oldest. Sliding windows may be time (i.e. holding tuples from the last 20 minutes) or count-based. We consider the time-based windows in this paper.

Although incremental computation and windowing unblock the join operator and bound its processing and memory requirements, operating conditions may still overwhelm a DSMS' resources. In reality, stream arrival rates fluctuate over time and may exceed the DSMS' processing capability. In addition, the DSMS' resources may become constrained when it simultaneously performs multiple continuous queries. In these situations, there is no recourse to generating approximate query results. In this paper, we consider approximation by load shedding, where tuples are prematurely dropped either before or after entry into a sliding window.

Our work is motivated by the following example. Consider a network of battery-powered sensors monitoring environmental conditions. Sensors, which have limited processing, storage, and communications capabilities, transmit gathered data to proxies, which are terminals to which users pose queries. Rather than sending raw tuple data to proxies as in [8], sensors append importance metadata before transmitting tuples. A tuple's importance metadata may be a function of its frequency, its presence in a predetermined range, its degree of statistical aberrance, or its distance to a cluster point [7], [18], for instance. A proxy accepts input streams and streams output tuples to the query source. The importance of an output tuple, which provides

*This work was partially supported by the NSF under grant # CNS-0521142 and The University of Michigan – Dearborn under a CEEP grant.

domain information to the query source beyond the raw tuple data, is a function of the importance of the input tuples which compose it.

In this paradigm, an approximation may occur at a sensor or at a proxy, and the objective is to minimize the approximation error to the greatest extent possible. In the former case, an approximation arises if power constraints prevent the sensor from transmitting all its tuples. In this event, the sensor aims to transmit the most relevant tuples to the proxies. Approximations in the latter case occur when the number of queries being posed or the volumes of incoming data exceed a proxy's computational resources.

This paper addresses the problem of load shedding at a sliding window join operator where such a system has sufficiently fast CPUs but lacks enough memory to compute the exact join result. Since all load shedding approximations are subsets of the exact result [8], our techniques are aimed at producing the approximation with the least error. Random load shedding techniques [10], [5], [17], which drop tuples randomly with respect to join attribute values, are known to produce sub-optimal approximations because the join result is composed of pairs of matching tuples from the input streams. In the extreme, a bad load shedding strategy can produce few or no output tuples, even though the largest possible approximate result is large. Semantic load shedding [8] improves upon random load shedding by taking into account join attribute value correlations between the two input streams with the aim of maximizing the size of the approximate join result under the memory constraint. However, semantic approximation depends only upon join attribute distributions and not importance semantics.

The presence of embedded per-tuple importance semantics relevant to the query source differentiates our work from previous load shedding techniques. Our objective is to compute the approximate result having the largest aggregate *importance*, given the memory constraint. Unlike those previously mentioned, our techniques process data streams whose tuples have different explicit importance values and evaluate these tuples on two independent criteria: importance *and* the expected number of matches.

Firstly, highly important output tuples provide more domain information to the query source than output tuples with lower importance. In this way, tuple importance semantics assist in QoS-based approximation. Dropping an input tuple randomly or because of its relatively low number of matches ignores the fact that the tuple may provide valuable information to the query source if it creates an output. For example, semantic approximation techniques will likely drop a highly anomalous tuple prematurely because its join attribute value occurs infrequently. Yet, despite its infrequent occurrence, this anomalous value might be actually be a "needle in the haystack" that provides the application with valuable information whenever it joins, especially in the case of an approximation. It may be desirable to retain this tuple in the join memory for as long as possible with the expectation that it will produce at least one important output tuple.

Secondly, assuming equal importance, an input tuple with many matches creates more aggregate importance in the join result than a tuple with fewer matches. Consequently, retaining input tuples solely because of their high importance may run counter to the objective function. Because it, in effect, regards all input and output tuples as having equal explicit importance, semantic load shedding deals with a special case of the problem we consider. Indeed, domain importance together with join semantics constitute a novel, non-trivial extension to the problem of load shedding in sliding window join approximation.

In this paper, we consider a new model in which data stream elements contain numerical importance values relevant to the query source. We propose different strategies of incorporating tuples' importance attributes into the priority formulation of online algorithms. In addition, we propose various methods of formulating the importance of output tuples from matching inputs and establish the objective function of maximizing the approximate join result's aggregate importance. We demonstrate that this new objective function is a non-trivial extension of load shedding by showing that random and semantic load shedding solutions previously proposed in the literature yield sup-optimal approximations. We extend the existing offline semantic load shedding algorithm [8] to compute the maximum importance under memory constraints and also present a new optimal offline join approximation algorithm with superior space efficiency and time efficiency at small and large join memory allotments. Furthermore, we propose four efficient online join approximation algorithms with superior efficacy to online random and semantic load shedding techniques.

The remainder of this paper is organized as follows. We review related work in Section 2. In Section 3, we formally state our problem and describe different methods of formulating join memory priorities and output tuple importance. We present our offline and online algorithms in Section 4 and experimentally evaluate them in Section 5. Finally, we conclude the paper and outline directions for future work in Section 6.

2. RELATED WORK

Golab *et al.* [14] investigate the problem of join ordering in queries with multiple sliding window joins and explore the tradeoffs of eager and lazy re-evaluation and expiration. Data stream summary structures [3] produce approximate results when a sliding window is too large to fit in available memory or when a streaming version of a blocking operator does not exist or is too inefficient. These techniques also provide an efficient way to maintain data stream statistics used in computing tuple priorities for online join memory maintenance.

While the symmetric hash join [2] was the first algorithm to process joins over unbounded streams, XJoin [10] was the first to shed load when stream arrival rates exceed the available join processing capacity. Processing two streams, XJoin randomly sheds load by spilling tuples to disk and processes backlogged inputs when the DSMS is once again able to cope with stream arrival rates. Viglas *et al.* apply their optimization framework for maximizing output production rate [17] to MJoin [5], which extends XJoin by aggregating or decomposing query plans containing multiple join operators. We consider the load shedding scenario in which tuples are dropped permanently.

Given CPU or memory constraints, Kang *et al.* [16] study the problem of optimal resource allocation with the aim of maximizing window join processing efficiency or output size and propose a per-unit-time cost model to evaluate their random load shedding techniques. Das *et al.* [8] show that random load shedding in general yields join approximations of sub-optimal size. Their semantic load shedding techniques aim to compute the join approximation of maximum *size* over input streams without the importance semantics that we consider here.

Stream semantics, known as punctuations [9], have been proposed in the literature. Punctuations provide information about the remainder of an input stream which can be used to unblock an operator. However, work in this area does not address load shedding or sliding window join approximation.

Eddies [15] and NiagaraCQ [5] are adaptive continuous query processing systems which deal with resource fluctuations by dynamically re-ordering operators. Though promising for providing reliable query performance in changing environments, they do not address tuple importance semantics or join approximation via load shedding.

The Aurora query processor [1] employs a different approach to load shedding in continuous queries. Aurora continuously monitors the frequency distribution of output tuples from streaming operators and compares it to the distributions of tuples in the input streams. It augments input tuples with QoS values from its monitoring statistics to ensure that load shedding reasonably preserves the distributions of the input tuples in the outputs. However, its techniques do not consider optimizations taking into account importance semantics embedded within data stream tuples at their source. To our knowledge, our work is the first to consider offline and online optimization of window join approximation in this context.

3. PROCESSING TUPLE IMPORTANCE

In this section, we define the problem space and discuss the formulation of join memory priorities in online approximation algorithms and the importance of output tuples.

3.1 Problem Definition

We process a sliding window equi-join between two data streams, R and S. Tuples in a stream are identified as <*ts*, *sch*, *imp*>, where $ts \in \mathbf{N}$, the set of natural numbers, is the tuple's arrival timestamp; *sch* is the conventional schema of the stream; $imp \in \{x \mid x \in \mathbf{R} \text{ and } 0 < x \leq U\}$ is the tuple's importance, where **R** is the set of real numbers and U is an upper bound for numerical importance. We employ time-based windows where one tuple arrives in each input stream per time instant, though our discussion extends to count-based windows or asynchronous tuple arrival.

We adopt the notation in [8]. Let R be a sliding window of size w over stream R, where w is also the lifetime of tuples in window R. Let $r(i)$ be a tuple that arrives in stream R at time i. For convenience, this tuple's join attribute also has a value $r(i)$. At the end of time t, window R contains tuples $r(i)$ such that $0 \leq t{-}w{+}1 < i \leq t$. The description of tuple $s(i)$ arriving in window S over stream S is analogous.

We employ the Fast CPU approximation model [8], where tuples are not dropped before reaching the join operator, with eager re-evaluation and expiration [14]. The join memory, M, which is fixed, is bounded above by $2w$, the amount required for exact computation. When input tuples $r(t)$ and $s(i)$ join at time t to create output tuple $o(t)$, the importance of $o(t)$ is a function of $r(t).imp$ and $s(i).imp$. Given that streams R and S begin at $t{=}0$ and end at $t{=}N$, the result multiset of the sliding window join is

$$\bigcup_{t=0}^{t=N} \; \bigcup_{i=t-w+1\geq 0}^{t} \{(r(t) \bowtie s(i))\} \cup \{(s(t) \bowtie r(i))\}$$

and has importance

$$\sum_{t=0}^{t=N} \sum_{y \in Y(t)} y.imp\,,$$

where the inner summation is the total importance of the set of output tuples, $Y(t) = \{o(t)\}$, created at time t. Any load shedding strategy produces a subset of the tuples in the exact answer whose importance is, consequently, no larger than that of the exact result. Because memory is constrained (i.e. $0 < M < 2w$), exact computation is not possible. We seek to compute the approximate result with the maximum importance.

3.2 Priorities and Output Tuple Importance

We first consider the formulation of tuple priorities, the basis of join memory retention and eviction in online algorithms. Assigned by the join algorithm, a tuple's priority is its estimated worth with respect to the objective function relative to the other tuples in the constrained join memory. The tuple priority is determined by importance, *imp*, the (expected) number of matches, *m*, and the arrival timestamp (i.e. age), *ts*. There is no explicit or implied correlation among these three parameters. Note that semantic approximation [8] does not factor importance into its priority formulation. In our case, tuple $r(i)$ has priority

$$P(r(i)) = f(r(i).imp, r(i).m, r(i).ts),$$

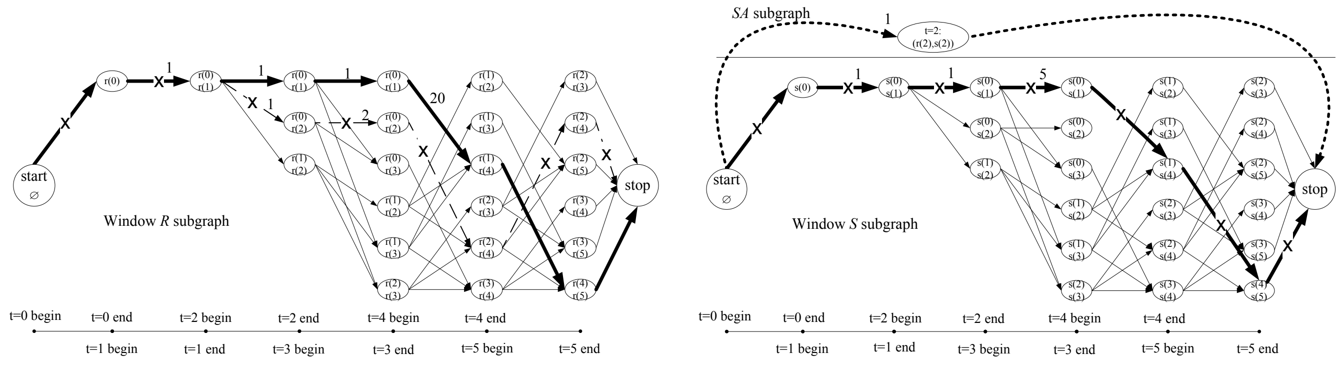
for some function f. The priority of $s(i) \in S$ is defined similarly. A tuple's priority is either assigned once when it arrives or is updated at each re-evaluation interval. The following are some general possibilities for priority formulation.

1. *Additive*: $r(i)$'s priority may be a linear combination of its importance, matches, and arrival time as follows:
   $$f(r(i).imp, r(i).m, r(i).ts) = \alpha * r(i).imp + \beta * r(i).m + \delta * r(i).ts,$$
   where $\alpha, \beta, \delta \geq 0$.
2. *Multiplicative*: alternatively, $P(r(i))$ can be formulated multiplicatively as
   $$f(r(i).imp, r(i).m, r(i).ts) = (r(i).imp)^{\alpha}\,(r(i).m)^{\beta}\,(r(i).ts)^{\delta},$$
   where $\alpha, \beta, \delta \geq 0$.
3. *Cumulative*: in this case, a $r(i)$'s priority is a function of its importance, matches, and arrival time as described in option 1 or 2, in combination with its previous priority:
   $$P(r(i)) = g(P'(r(i)), r(i).imp, r(i).m, r(i).ts),$$
   where g is some function and $P'(r(i))$ is the priority previously assigned to $r(i)$. When employing a cumulative scheme, the tuple's priority is re-calculated during every time interval, similarly to a rolling average.

We will evaluate special cases of all three priority formulations in our experiments.

We next consider how to determine an output tuple's importance from the importance of the input tuples that join to create it. Specifically, we consider how to formulate $o(t).imp$, created at time t, from joining $r(i)$ and $s(t)$ (or $r(t)$ and $s(i)$). We must address the issue of output tuple importance because matching input tuples convey their importance to the monitoring application or query source through $o(t).imp$. Moreover, $o(t)$ can conceivably be an input to another join operator, where its importance is once again used to determine its priority within that join memory. Intuitive ways to derive an output tuple's importance from the inputs are additive (i.e. $o(t).imp = r(i).imp + s(t).imp$), multiplicative, maximal, minimal, and average. Without loss of generality, we use the "minimum" output importance formulation, where

$$r(i) \bowtie s(t) \Rightarrow o(t).imp = \min\{r(i).imp, s(t).imp\}.$$



Exact Result: *R* subgraph

Output tuple
(*r(0),s(1)*)=<1,1,1>
(*r(0),s(2)*)=<2,1,1>
(*r(0),s(3)*)=<3,1,1>
(*r(2),s(3)*)=<3,1,1>
(*r(1),s(4)*)=<4,9,20>
(*r(2),s(5)*)=<5,1,1>

Exact Result: *S* subgraph

Output tuple
(*r(2),s(1)*)=<2,1,1>
(*r(3),s(0)*)=<3,3,5>

SA subgraph

Output tuple
(*r(2),s(2)*)=<2,1,1>

Optimal approximation for *M*=4:
7 output tuples, importance = 30

Output tuple
(*r(0),s(1)*)=<1,1,1>

(*r(0),s(2)*)=<2,1,1>
(*r(2),s(1)*)=<2,1,1>
(*r(2),s(2)*)=<2,1,1>

Output tuple
(*r(0),s(3)*)=<3,1,1>
(*r(3),s(0)*)=<3,3,5>
(*r(1),s(4)*)=<4,9,20>

Legend
━▶ Path 0
—X▶ Path 1

Figure 1. Join memory state graph

In practice, the output importance formulation is application-dependent. Nevertheless, the real world meaning of importance semantics is irrelevant to our priority and output importance formulations beyond the ability to order priorities and to propagate importance values from input tuples to the outputs.

4. APPROXIMATION ALGORITHMS

We first introduce our offline approximation algorithm, which employs its foreknowledge of the input streams to generate an approximate join result with the maximum importance. This establishes the baseline against which we measure the efficacy of online algorithms, which have no knowledge of future inputs.

4.1 Join Memory State Graph

We formulate the offline window join approximation as a directed graph. Vertices correspond to snapshots of the join memory at different points in time, while edges, which model transitions between sequential memory states, represent choices to retain or drop tuples. The join memory state graph, which can be constructed for arbitrary combinations of tuple lifetime (*w*), join memory size (*M*), and input stream length (*N*), models all possible ways to retain and evict tuples. Our goal is to determine the tuple eviction and retention strategy corresponding to the approximate join result with the largest importance.

Figure 1 illustrates the semantics of the join memory state graph. In this example, both data streams have length *N*=6. From Subsection 3.1, data stream tuples are of the form <*ts*, *sch*, *imp*>. Stream R is <0,1,1>, <1,9,20>, <2,1,1>, <3,3,5>, <4,4,5>, <5,2,1>; stream S is <0,3,5>, <1,1,1>, <2,1,1>, <3,1,1>, <4,9,20>, <5,1,1>. Only the join attributes of *sch* are shown.

The modeling of time is discrete, so the first tuple in each stream arrives at time *t*=0, the second tuple arrives at *t*=1, and so on. Tuples are identified by a combination of their stream and arrival time. For example, tuple <1,9,20> in stream R is referred to as *r(1)*; it arrives in stream R at *t*=1, and its join attribute and importance values are 9 and 20, respectively. The importance of an output tuple is the minimum of those of the matching tuples that create it (see Subsection 3.2). For instance, *r(2)* and *s(2)* join at *t*=2 to create (*r(2),s(2)*), which has a join attribute value of 1 and an importance of 1.

In this problem, the tuple lifetime and window size are *w*=4. Assuming that each tuple occupies one unit of memory, 2*w*=8 memory units are required to compute the exact result. The available join memory, *M*=4, is half of this amount. In this example, *M* is divided evenly between windows *R* and *S*.

Events in the join approximation are modeled in three subgraphs: the *R*, *S*, and *SA* subgraphs. An output tuple in the *SA* subgraph is created when two matching tuples in opposite streams arrive at the same time. In subgraphs *R* and *S*, however, an output tuple is created when a tuple in a memory state joins with the new tuple arriving in the opposite stream. Because of symmetry, subgraphs *R* and *S* share the same properties.

Memory states (i.e. vertices) are represented by the set of tuples they contain. The start vertex represents the join memory before any data stream tuple arrives, while the stop vertex represents the time instant after the final tuple has arrived. An edge's vertex of origin represents the join memory state at the instant when a new tuple arrives. When it arrives, the tuple is either (1) admitted into the join memory which is not full, or (2) admitted at the expense of an expired tuple, or (3) admitted at the expense of an active tuple in the join memory, or (4) dropped before entering the join memory. An edge is created from the current memory state to a subsequent state in the next time instant for each of these events. For example, *r(0)* and *r(1)* are admitted into the join memory which is not yet full. When *r(2)* arrives, to a full join memory (*M*/2 tuples in this example), since neither *r(0)* nor *r(1)* has expired, new memory states must be created from vertex {*r(0),r(1)*} at *t*=2, each representing a decision to drop *r(0)*, *r(1)* or *r(2)* prematurely.

An edge's weight is the importance of the output tuples resulting from the decision to admit, expire, retain, or drop tuples. For example, if *r(0)* is in memory at *t*=2, it joins with *s(2)* to create output tuple (*r(0),s(2)*) which has importance 1. Thus, the two memory states at *t*=2 in which *r(0)* survive have incident edges with weight 1. On the other hand, edge {*r(0),r(1)*}→{*r(1),r(2)*} represents the decision to drop *r(0)* for *r(2)* at *t*=2, so its weight is 0. (For clarity, edge weights of 0 are not shown.) Note that an edge's weight represents the *total* importance from the output tuples from the originating vertex. If multiple tuples in the join memory state match the new tuple in S, the edge weight is the sum of the importance of the created output tuples. For instance, edge *{r(0),r(2)}*→*{r(0),r(2)}* at *t*=3 has a weight of 2 because both *r(0)* and *r(2)* match with *s(3)* to create two output tuples of importance 1.

A sequence of vertices from the start vertex to the stop vertex represents a complete path. Each path represents a set of decisions to retain or evict tuples and the states resulting from those decisions. Paths respect correct temporal modeling because they represent a set of transitions from one time instant to the next. In addition, paths in a subgraph respect the memory constraint because vertices within all paths do not accommodate more than the allotted number of tuples. For example, each vertex in the *R* and *S* subgraphs of Figure 1 does not accommodate more than *M*/2 tuples. Moreover, paths only contain valid transitions between memory states, meaning that a tuple that is dropped within a path cannot re-enter a join memory state. For example, no path in Figure 1 contains the edge *{r(0),r(1)}*→*{r(1),r(2)}* at *t*=4. For any path, *r(2)* can enter a join memory state only at *t*=2 and, once dropped, cannot possibly re-enter a join memory state at a future time. Thus, each path from the start vertex to the stop vertex represents a valid sequence of join memory states adhering to the memory constraint. In other words, each complete path from the start vertex to the stop vertex represents a join approximation. The optimization goal is to find a path from the start vertex to the stop vertex such that the sum of the edge weights is maximal.

States and transitions in the *R* subgraph are independent of those in the *S* subgraph because join memory states in *R* are determined solely by tuples arriving in *R*. Furthermore, edge weights in the *R* subgraph depend on the arrival of tuples in stream S and *not* on the contents of window *S*'s join memory. The same is true of the corresponding structures in the *S* subgraph in relation to those in the *R* subgraph. Subgraph *SA* is independent of both *R* and *S* because it alone captures output tuples from matching input tuples that arrive at the same time. Because subgraphs *R*, *S*, and *SA* are independent, the optimal approximation consists of the output tuples in the *SA* subgraph in union with the optimal paths in subgraphs *R* and *S*.

For our example, the exact join result (i.e. *M*=8, see Figure 1) contains nine output tuples and has an importance of 32. For clarity, the only non-zero edge weights shown are those of paths 0 (bold) and 1 ('x'). Path 0 corresponds to the join approximation for *M*=4 with the maximum importance. This approximation contains seven output tuples and has an importance of 30. For comparison, the *largest* approximate join result for *M*=4 (i.e. the optimal semantic load shedding approximation [8]), represented by Path 1, contains eight output tuples but has an importance of only 12.

The join memory state graph has two more useful properties. First, different paths have overlapping subpaths. For example, three different subpaths starting at *t*=4 contain subpath *{r(1),r(4)}*→*{r(4),r(5)}*→stop, which begins at *t*=5. Second, the path corresponding to the optimal approximate join result necessarily contains within it optimal subpaths, which means that the graph formulation possesses optimal substructure. The presence of both properties, which will be discussed in more detail in Subsection 4.2.2, leads us to a dynamic programming optimal offline algorithm, which we present next.

4.2 Optimal Offline Approximation

Our optimal offline approximation algorithm consists of several procedures, which are described in the following subsection.

4.2.1 Optimal Approximation Construction

In the following procedures for constructing the join memory state graph and determining the optimal approximation, variable *OJI* represents the importance of the optimal join approximation. Variable MS[*t*] contains all the join memory states (i.e. vertices) at time *t*. A memory state's *MI* field represents the maximum importance of all paths from the start vertex to that memory state, and a state's *prev* field is the immediate predecessor vertex in this optimal path. Both are initialized to zero for new vertices. Function *Importance()* returns the sum of the importance of the output tuples produced by the joined inputs for a given memory state at a time instant.

```
// Construct subgraph R, compute optimal path and importance
// Repeat this procedure for subgraph S
Procedure ComputeMaxImportance( stream R, stream S, $N$ )
  OJI = 0, $t$ = 0
  MS[$t$] = ∅, initialize and insert StartVertex into MS[$t$]
  do
    MS[$t$+1] = ∅
    ∀ memory state x ∈ MS[$t$]
      // Generate memory states for MS[$t$+1] using r($t$)
      ∀ memory state y from x
        if( y ∉ MS[$t$+1] )
          Insert y into MS[$t$+1]
          EdgeWeight( x → y ) = Importance( y, s($t$+1) )
          if( x.MI + EdgeWeight( x → y ) ≥ y.MI )
            y.MI = x.MI + EdgeWeight( x → y )
            y.prev = x
    $t$ = $t$+1
  while( $t$ < $N$ )
  MS[$t$] = ∅, initialize and insert StopVertex into MS[$t$]
  ∀ memory state x ∈ MS[$t$–1]
    if( x.MI ≥ StopVertex.MI )
      StopVertex.MI = x.MI
      StopVertex.prev = x
  OJI =  OJI + StopVertex.MI.
// Construct SA subgraph, compute importance
Procedure ComputeSAImportance( stream R, stream S, $N$ )
  vertex x = StartVertex
  for( $t$ = 0 to $N$–1 )
    if( Importance( r($t$), s($t$) ) > 0 )
      Create vertex y = ( r($t$), s($t$) ) for SA subgraph
      EdgeWeight( x → y ) = Importance( r($t$), s($t$) )
      x = y
      OJI = OJI + Importance( r($t$), s($t$) )
  EdgeWeight( x → StopVertex ) = 0.
```

Using the join memory state graph and the information produced by the above procedures, we can print the input tuples in the states corresponding to the optimal join approximation in the reverse order of time. The procedure starts from the stop vertex and iteratively uses the *prev* field to traverse the predecessor in the optimal path.

```
// Print the tuples of optimal subgraph R approximation
// Repeat the procedure for subgraph S
Procedue ReconstructContents( subgraph R )
  vertex x = StopVertex
  do
    x = Locate( x.prev )
    printVertex( x )
  while( x ≠ StartVertex )
```

Through simple modifications, the algorithm can satisfy different objective functions. For example, modifying *Importance()* to return the number of output tuples from a memory state would cause the algorithm to produce the optimal semantic approximation [8]. In addition, changing the procedure so that the vertices' *MI* fields store the smallest importance of all the paths leading to them allows the algorithm to compute the approximation with the lowest importance. Alternatively, the algorithm can support a combination of positive and negative edge weights, where desirable output tuples have positive importance and undesirable outputs tuples have negative importance.

4.2.2 Correctness and Complexity Analysis

The correctness of the algorithm follows from the preservation of the following invariant: at the end of each time step, every vertex contains information to determine the maximum importance from the start vertex to itself. Thus, the stop vertex in the *R* and *S* subgraphs contains the maximum importance from the start vertex to itself. The optimal substructure and overlapping subpath properties present in the join memory state graph formulation allow for the preservation of this invariant.

We first show that the problem exhibits optimal substructure. Let G(V,E) represent a join memory state graph. *Importance*: E→**R** is the weight function for edges as described in Subsection 4.1. Let P be a path in the join memory state graph from the start vertex to the stop vertex which yields the join result with the greatest importance. P consists of vertices $\{v_0,v_1,\ldots,v_N\}$ and edges $\{e_{0,1},e_{1,2},\ldots,e_{N-1,N}\}$, where edge $e_{i,j}$ connects vertices v_i and v_j for $0 \le i < j \le N$. If $P_{i,j}$ is a subpath in P starting at vertex v_i and ending at vertex v_j, then $P_{i,j}$ yields the greatest importance between v_i and v_j. If not, then decompose P as $P_{0,i} \to P_{i,j} \to P_{j,N}$, which has importance *Importance*(P) = *Importance*($P_{0,i}$) + *Importance*($P_{i,j}$) + *Importance*($P_{j,N}$). Since there is an alternate path $P'_{i,j} \in$ G(V,E) such that *Importance*($P'_{i,j}$) > *Importance*($P_{i,j}$), we substitute this path into P in place of $P_{i,j}$ to form path P′, whose importance, *Importance*($P_{0,i}$) + *Importance*($P'_{i,j}$) + *Importance*($P_{j,N}$), is greater than *Importance*(P). This contradicts the assumption that P has the greatest importance from the start to the stop vertex.

Next, we explain why the graph formulation possesses property of overlapping subpaths and how the algorithm uses the property to generate the optimal approximation. Let v_i be a memory state in MS[*t*]. As a result of *r(t)*'s arrival, v_i generates between one and *M*/2+1 subsequent states (inclusive) in MS[*t*+1] corresponding to each of the following events:

- Join memory is not full. *r(t)* is admitted.
- Join memory is full. *r(t)* displaces the expired tuple in v_i.
- Join memory is full. *r(t)* displaces *any* active tuple in v_i.
- Join memory is full. *r(t)* is dropped.

Let v_{i+1} be a subsequent state of v_i. v_{i+1} may also be reached independently through memory state v_h in MS[*t*]. Updating v_{i+1}'s *MI* and *prev* fields enables the algorithm to select which of the paths from the start vertex ending at v_{i+1} maximizes the importance measure, while storing only one copy of v_{i+1}. In Figure 1, for example, vertex *{r(1),r(4)}* at *t*=4 can be reached from *{r(0),r(1)}*, *{r(1),r(2)}*, or *{r(1),r(3)}* at *t*=3. By maintaining non-duplicate vertices, the algorithm allows these paths to share subpath *{r(1),r(4)}*→*{r(4),r(5)}*→stop and decides which of them maximizes the importance measure without the need to store a separate copy of the subpath for each.

While our methods also work for uneven memory allocations, the following time complexity analysis assumes that each window holds at most *M*/2 tuples. Each subgraph contains *N* time instants, and there are at most $\binom{w}{M/2}$ memory states in a time instant. Each state generates at most *M*/2+1 new states of size *M*/2. Each new memory state is generated in O(*M*/2) [19]. The calculation of *Importance()* for each output tuple is absorbed into this step. The algorithm uses the overlapping subpaths property by maintaining non-duplicate memory states in a time instant. Duplicate avoidance is performed in O(*M*/2). Each *MI* and *prev* update is performed in O(1). Thus, the algorithm determines the sliding window join approximation of maximum importance in $O\left(N\binom{w}{M/2}(M/2)^3\right)$. The time complexity approaches O(*Nw*) as *M*/2→1 and approaches O(Nw^4) as *M*/2→*w*–1. The printing procedure runs in θ(*N*), since there are *N* time instants and the *Locate()* step takes constant time.

The algorithm requires MS[*t*] and MS[*t*+1] of the join memory graph to produce an optimal approximation. Thus, its space complexity, $\theta\left(2\binom{w}{M/2}\right)$, is independent of *N*. The printing procedure only requires O(1) space since only vertex v is stored.

4.2.3 Extending the Linear Flow Solution

Semantic approximation [8] is sub-optimal for maximizing the importance measure because it only computes the largest approximate join result. We extend the original semantic approximation technique to make it compatible with our objective function (omitted due to space constraints). We modify the flow graph creation algorithm to accept tuple importance semantics by allowing arc weights of arbitrary (absolute) magnitude and employ an efficient implementation of the minimum cost linear flow algorithm [11] to calculate the optimal join approximation's *numerical* importance in O($(wN)^2(wN+M)\log(wN)$). Unlike our offline algorithm, the space efficiency of this algorithm, O(*wN*+*M*), is a function of *M*, *w*, and *N*. The extended semantic approximation technique and our optimal offline algorithm will be evaluated empirically in Section 5.

4.3 Online Approximation

Unlike the optimal offline algorithm, online approximation algorithms have no knowledge of future input tuples. Therefore, they have no recourse but to greedily hold on to tuples they consider valuable for the objective function according to pre-determined heuristics, which include:

1. Using past usefulness (i.e the importance of the outputs generated by the tuple) to estimate future worth.
2. Using current or past join attribute distributions to estimate future match probabilities.

Such statistical and historical information is maintained automatically by DSMSs as histograms and sketches [3]. Our techniques employ these general guidelines while also incorporating tuple importance. We present four efficient online algorithms in the Fast CPU framework [8] with either static or dynamic priorities.

4.3.1 SIMP Heuristic

The Static IMPortance heuristic (SIMP) fixes the priority of each tuple as its importance. This is a special case of the additive (or multiplicative) priority formulation in Subsection 3.2. SIMP prefers high importance tuples over those with lower importance, regardless of estimated match probabilities or the relative ages. The tuple of lowest priority (including the new tuple) is dropped whenever a tuple arrives at a full window. If a tie in priorities occurs, the oldest tuple is dropped. For simplicity, suppose that the join memory is maintained as two priority queues, each holding *M*/2 tuples. Using the cost model in [17], SIMP's per-tuple processing cost is evaluated as

$$M/2(\mathit{probe} + \mathit{invalidate}) + 1(\mathit{insert}).$$

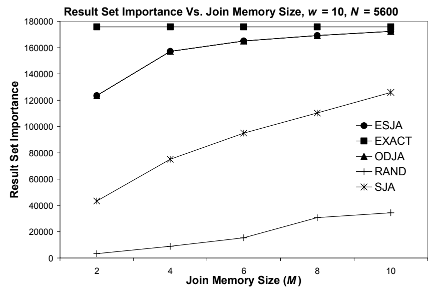


Figure 2. Importance vs. memory size

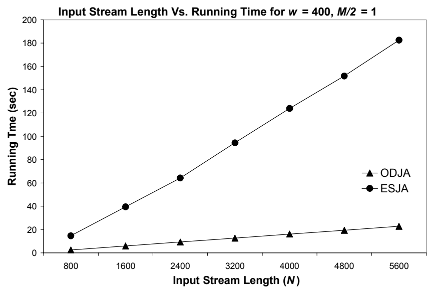


Figure 3. Running times for *M/2* = 1

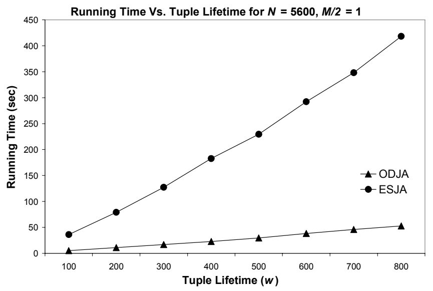


Figure 4. Running times for increasing *w*

4.3.2 SIMPPROB Heuristic

When a new tuple arrives, the Static IMPortance PROBability heuristic (SIMPPROB) fixes its priority as the product of its importance and the number of matches in the opposite window, the latter number being the new tuple's estimated match probability. This is a special case of the multiplicative priority formulation. Since a new tuple may have no matches, any tuple can be prioritized to zero, regardless of its importance. Ties are resolved by dropping the tuple with the lowest importance, then the tuple with the fewest number of matches, followed by the oldest tuple. Because priorities are assigned once, SIMPPROB's per-tuple processing cost is the same as that of SIMP. Note that the online semantic approximation heuristic [8] also has the same per-tuple processing cost.

4.3.3 DIMPPROB Heuristic

The Dynamic IMPortance PROBability (DIMPPROB) heuristic also calculates a new tuple's priority as the product of its importance and the number of matches. However, expected match probabilities are updated at each time step. Ties are resolved exactly as they are in SIMPPROB. Assuming $M/2$ tuples per window, the per-tuple processing cost,

$$M/2(probe + invalidate + priority\_uddate) + 1(insert),$$

is greater than SIMPPROB's, where match probability estimates become inaccurate over time as the opposite window changes.

4.3.4 DGL Heuristic

The Dynamic Gain Loss heuristic (DGL) employs the cumulative priority formulation. At arrival, a tuple's priority is its importance. If it produces an output during a time instant, the tuple's priority is increased by an amount proportional to the product of its importance, its current expected number of matches, and its remaining lifetime. A tuple's priority is decreased by a constant factor during time instants when it does not produce an output. Thus, high importance tuples generally make priority gains more quickly with each matching partner than low importance tuples, though gains are reduced as tuples age. DGL's analytical per-tuple processing cost is the same as DIMPPROB's, as all priorities are modified at every time step.

5. EXPERIMENTS

We verify the feasibility of tuple importance semantics in window join approximation and evaluate the efficacy and efficiency of our techniques under different conditions using independently generated synthetic input streams. Normal importance tuples dominate the input streams, while tuples with high importance, which correspond to aberrant or special values, occur infrequently. M is divided evenly between windows R and S, though our techniques work for uneven allocations. Experiments comparing approximation quality begin counting outputs at t=2w, since all algorithms accept the first $M/2$ input tuples from each input stream, the last of which may still be in memory at t=$M/2$+w. The starting time for counting is adjusted to reflect the maximum w or $M/2$ in experiments where these parameters are varied. The platform is a 2.0 GHz AthlonXP machine with 1.0 GB of RAM running SUSE Linux 9.3. Each reported running time is the average of three trials.

5.1 Offline Join Approximation

We compare our offline algorithms to semantic and random load shedding with respect to efficacy, and time and space efficiency.

5.1.1 Approximation Quality

Figure 2 depicts the approximation qualities of the aforementioned load shedding strategies as a function of increasing join memory size (M) given a fixed tuple lifetime, w=10, and input stream length, N=5600. Henceforth, EXACT is the exact result, RAND is random load shedding, and SJA and ESJA, respectively, represent the semantic and extended semantic (see Subsection 4.2.3) load shedding techniques. ODJA is our dynamic programming approximation technique.

All techniques show improved approximation quality relative to the exact result (EXACT) as join memory increases. For example, at $M/2$=1, SJA produces a 75.3% approximation error, while ODJA/ESJA's error is 29.7%. At $M/2$=5, the respective approximation errors shrink to 28% and 1%. RAND's approximation error decreases from 98% at $M/2$=1 to 80.4% at $M/2$=5. RAND is least effective because it drops input tuples without regard for importance or the number of output tuples produced. ODJA and ESJA produce the maximum possible importance under the memory constraint and converge upon EXACT more quickly than other methods. This result is expected because both methods recognize important input tuples *and* those which produce many outputs. SJA generates the largest result set but is consistently sub-optimal because it only favors input tuples for their ability to produce many output tuples. At $M/2$=1 and $M/2$=2, for example, SJA generates 164 and 97 more output tuples than ODJA/ESJA.

5.1.2 Running Times at M/2 = 1

Since ODJA and ESJA always yield the same (optimal) approximation quality, we examine the running times of the two techniques in the first special case discussed in Subsection 4.2.2, namely $M/2$=1.

In Figure 3, w is fixed at 400 time units, and N is increased from 800 to 5600. ODJA is 481% more efficient at N=800 and becomes 700% more efficient at N=5600. In Figure 4, where N

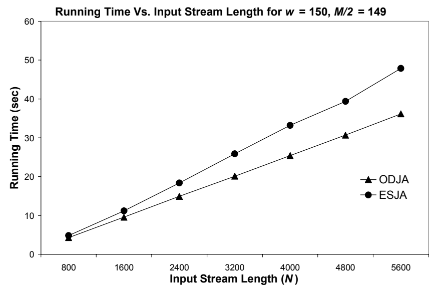


Figure 5. Running times for *M/2* = *w*–1

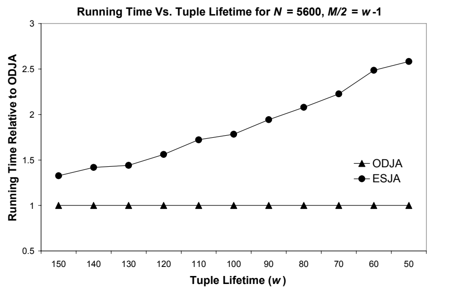


Figure 6. Running times for decreasing *w*

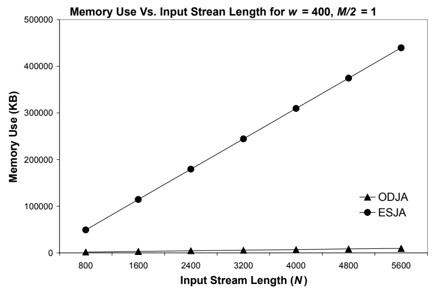


Figure 7. Memory use

is fixed at 5600 tuples and *w* varies from 100 to 800, ODJA's time efficiency advantage over ESJA progressively increases from 596% to 689%. ODJA's running time increases less rapidly with *N* and *w* because ODJA's running time approaches O(*Nw*) as *M*/2→1, while ESJA's is in O((*wN*)^3(log(*wN*))) (see Subsections 4.2.2 and 4.2.3).

5.1.3 Running Times at M/2 = w–1

We now consider ODJA and ESJA's relative running times in the second case described in Subsection 4.2.2, *M*/2=*w*–1. As Figure 5 shows, ODJA and ESJA require almost the same amount of time to compute the optimal approximation at *N*=800, but OJDA becomes increasingly more efficient as *N* alone is increased to 5600. This is because ODJA's running time, which is in O(*Nw*^{4}) when *M*/2=*w*–1, grows more slowly with increasing input stream length than ESJA's, whose running time is in O((*wN*)^3(log(*wN*))).

Figure 6 shows the effects of decreasing tuple lifetime on the relative running times of the two offline algorithms at *M*/2=*w*–1, given a fixed input stream length of *N*=5600. ODJA becomes more efficient relative to ESJA as *w* is reduced. Even though ODJA's running time increases more rapidly with increasing tuple lifetime than ESJA's (i.e. quadric vs. cubic), ODJA outperforms ESJA when *N* is sufficiently large because ESJA's running time is influenced more by the size of the input streams. Note that the when join memory is neither small nor large (i.e. *M*/2 is not close to 1 or *w*–1, respectively), ESJA is more efficient (results are omitted due to space constraints).

5.1.4 Memory Requirement

We investigate the effect of increasing input stream length on the memory requirement of ODJA and ESJA. Memory use, in kilobytes, is reported by *top*, the resource usage tool available in UNIX and LINUX operating systems. Figure 7, where *w*=400 and *M*/2=1, shows that ODJA's memory usage increases very slightly as *N* increases, whereas ESJA's memory use increases at a higher rate. ODJA should have a constant space requirement, regardless of the size of the input stream (see Subsection 4.2.2), yet our results show an increase from 1944 KB to 9704 KB as *N* is increased from 800 to 5600. This discrepancy arises because measurements of memory use include the input stream tuples *and* the algorithm state. Consequently, we conclude that ODJA's space requirement is constant, regardless input stream length. ESJA's memory requirement increases from 25.3 times ODJA's at *N*=800 to 45.3 times ODJA's at *N*=5600. This difference in memory utilization arises because ESJA must store all state (i.e. vertices and edges) in memory for *all* time instants, whereas ODJA only stores memory states for two time instants. A similar result (not shown) is obtained when *M*/2 is fixed at *w*–1 and the input stream length is increased from *N*=800 to *N*=5600. OSJA's memory requirement is invariably independent of *N*, so its space efficiency will be superior whenever *N* is sufficiently large.

5.2 Online Join Approximation

Before evaluating our online heuristics, we note that the online semantic approximation heuristic in [8], PROB, estimates match probabilities using the join attribute value distributions of the entire input streams. We maintain that no online heuristic can have knowledge of future tuples and, as a result, estimate match probabilities solely from the sliding windows.

5.2.1 Effect of Tuple Lifetime

We determine the effect of increasing tuple lifetime (*w*) on the relative approximation qualities of the online algorithms. Figure 8 shows the result set importance as a function of *w*, where *M*=200 (divided evenly between the windows) and *N*=5600. The two input streams have Zipf distributions of 1.0 and 0.0, respectively, and have a domain size of 100. The algorithms' relative efficacy is unaffected by increasing tuple lifetime. As expected, the maximum possible importance under the memory constraint, OPT (generated by either ESJA or ODJA), grows as tuple lifetime increases. This is because tuples that produce outputs can be retained longer, which increases their chances of finding matches in the opposite stream. For example, OPT's importance is 3.21 times greater at *w*=800 than at *w*=200. Among the online algorithms, DGL degrades most gracefully and yields the best approximation quality, followed by DIMPPROB and SIMPPROB. PROB's quality does not improve consistently because it pursues frequently occurring join attributes and ignores tuple importance. SIMP's quality improves little because it retains high importance tuples longer, without regard to whether they produce any output tuples. RAND's quality is, constant and poorest of all, since its eviction policy does not consider importance or match probability.

5.2.2 Effect of Memory Size

We consider the effect of increasing join memory size on the algorithms' approximation quality. In this experiment, *N*=5600, and the domain size is set to 100. *w* is fixed at 400, as it does not alter the algorithms' *relative* efficacy (see Subsection 5.2.1). The input streams have Zipf distributions of 1.0 and 0.0 as in Subsection 5.2.1. The results, shown in Figure 9, indicate that, all the heuristics approach EXACT as the join memory size increases. DGL produces the best approximations and degrades most gracefully. For instance, at *M*=100, the lowest memory setting in the experiment, DGL is 3.2% more efficacious than DIMPPROB, 20.6% more efficacious than SIMPPROB, 47.6% more efficacious than PROB, 52.9% more efficacious than

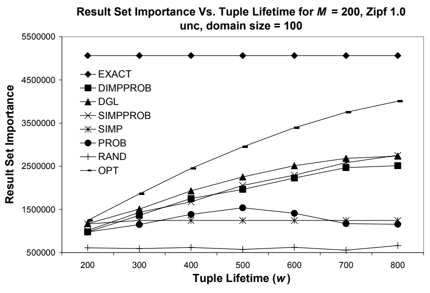


Figure 8. Effect of tuple lifetime

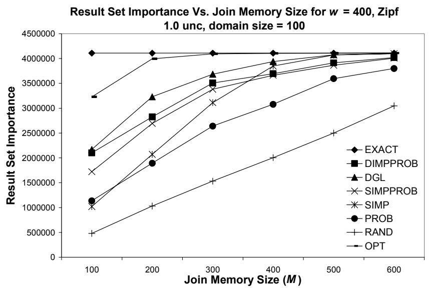


Figure 9. Effect of join memory size

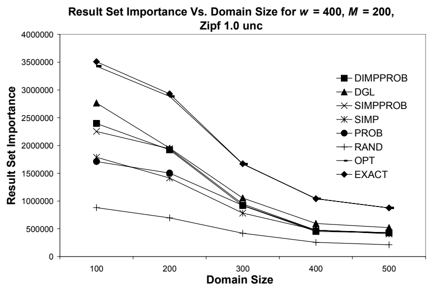


Figure 10. Effect of domain size

SIMP, and 77.8% more efficacious than RAND. Predictably, RAND performs most poorly. Its approximation quality scales linearly with *M* because its join memory can hold an increasing number of tuples. PROB's quality also increases linearly, though at a higher rate than does RAND's. PROB does not quickly converge upon OPT, since it ignores the importance attribute. In general, SIMP is a poor heuristic because it completely ignores match probabilities. It performs well only when the allotted join memory is close to the exact amount. SIMPPROB and DIMPPROB perform well, converging quickly upon EXACT. DIMPPROB's error remains somewhat less than SIMPPROB's because its match probability estimates are always updated. DGL outperforms DIMPPROB because skewness in the Zipf 1.0 input stream implies that some tuples occur with greater frequency than others. By inflating their priorities, DGL retains these tuples longer than DIMPPROB, giving them a better opportunity to find matches in the opposite stream.

5.2.3 Effect of Domain Size

Figure 10 shows the effect of increasing the join attributes' domain sizes on approximation quality. *M* is fixed at 200 tuples, *w*=400, *N*=5600, and the Zipf parameters are 1.0 and 0.0. Figure 10 shows that the result set importance varies inversely with the domain size. The online algorithms perform worse relative to EXACT as the domain size increases, while OPT very quickly converges upon exact. These results are explained as follows. Increasing the domain size in the Zipf 1.0 stream effectively reduces the number of join attribute values occurring with a high frequency and, simultaneously, increases the number of join attribute values occurring with a low frequency. Increasing the domain size in the stream with uniformly distributed join attribute values reduces the frequency of all join attributes values by an equal amount. One consequence of increasing "sparseness" in the distribution of the join attribute values of both streams is that tuples are increasingly unlikely to find matches. This lowers the importance of EXACT. Another result of increasing domain size is that, on average, tuples spend more time in the join memory before encountering matches, which explains the online algorithms' progressively worse performance relative to OPT, which relies on its foreknowledge to find a good retention strategy. The online algorithms' relative efficacy is the same as that obtained in Subsection 5.2.2: DGL yields the best approximation quality and degrades the least, followed by DIMPPROB, SIPPROB, PROB, SIMP, and RAND. Previous arguments account for their relative efficacies.

5.2.4 Effect of Skewness

In this experiment, the join attribute values of both streams are increased from Zipf 0.0 to Zipf 1.0 in a correlated fashion, while *w*=400, *M*=200, and the streams' domain size is fixed at 100. As Figure 11 shows, result set importance becomes progressively larger with increasing skewness, approaching OPT. Because the frequently and infrequently occurring join attribute values in both streams are correlated, input tuples, on average, have a greater expected number of matches and a reduced amount of time between matches as the skew parameter increases. Unlike previous scenarios, DIMPPROB and SIMPPROB are, on average, 2.1% more efficacious than DGL. This interesting result is due to the effects of correlation and skewness. The large number of matches causes priorities in DGL become inflated so that subsequently arriving tuples are less likely to be admitted into the join memory as compared to DIMPPROB. DIMPPROB more readily replaces old tuples that don't match with new tuples that do. DGL's favoritism of old high priority tuples over new tuples also causes it to perform worse than SIMPPROB, whose static match estimates prove accurate in this situation because of the large number of matches and the relatively small amount of time between matches.

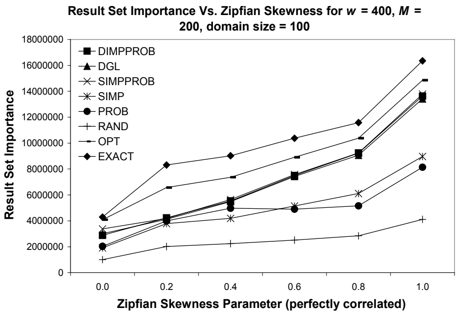


Figure 11. Effect of increasing skewness

5.2.5 Running Times of Online Algorithms

Table 1. Approximation quality and running time

	DIMPPROB	DGL	SIMPPROB	SIMP	PROB
Importance	5783148	6929937	5993464	4592168	5992694
Time	33.990	34.885	25.484	25.536	25.547

We compare the running times of the online algorithms using large, uncorrelated (Zipf 1.0 and 0.0) input streams of *N*=56,000 tuples, with *w*=2000, *M*=800, and a domain size of 2000. The results, shown in Table 1, confirm our previous approximation quality results. DGL provides the best approximation, followed DIMPPROB, SIMPPROB, and PROB, whose relative quality is not as low as in previous experiments because of the large domain size. Once again, the strategy of retaining high importance input tuples (i.e. SIMP) proves ineffective. The relative running times corroborate the cost modeling in Subsection 4.3, with DIMPPROB and DGL requiring more time than the static priority heuristics, including PROB. DGL's greater running time, offset by its superior efficacy, is attributed to its more complex tuple priority formulation.

5.3 Discussion of Experimental Results

We compared our sliding window join approximation algorithms to semantic and random load shedding techniques in memory-constrained situations. Input data consisted of synthetic data streams with tuple level importance semantics. Offline and online semantic and random load shedding produce poor join approximations when dealing with importance semantics, especially when very little join memory is available. However, our optimal offline algorithm, ODJA, and our extension to semantic load shedding, ESJA, compute an optimal join result and quickly converge upon the exact result as the amount of join memory is increased. ODJA is up to seven times more time efficient than ESJA at small and large join memory allotments. In addition, ODJA's memory requirement is independent of the input streams' size. It is 45.3 times more space efficient than ESJA for w=400 and M/2=1. Similar results hold for join memory allotments nearing EXACT. ESJA has superior space and time efficiency when M is neither very small nor large. On the other hand, our online algorithms are consistently superior to PROB, the online semantic load shedding heuristic, in situations where we vary tuple lifetime, memory size, domain size, and skewness. By properly detecting important tuples and those with high match probabilities, our online algorithms produce high quality approximations and quickly converge upon the exact result. DGL degrades most gracefully with decreasing join memory size in uncorrelated data streams and is most efficacious for joins over streams with large domain sizes and long tuple lifetimes, while DIMPPROB and SIMPPROB are most efficacious for correlated streams. The ineffectiveness of the SIMP heuristic and the consistency of our results on a large dataset give us confidence that the data sets do not inherently favor our techniques and, respectively, that that the efficacy of our methods is unaffected by specific choices of input stream size, domain size, tuple lifetime, and join memory size. Overall, we discovered that join approximation error diminishes with increasing memory size. The greatest optimization opportunity exists for small join memory sizes, and it is in this situation that our methods show the largest gains over the previous state of the art in terms of efficacy, space efficiency, and time efficiency.

6. CONCLUSIONS AND FUTURE WORK

This paper examined the problem of computing memory-constrained sliding window join approximations over data streams. We motivated the inclusion of importance semantics within input tuples and the objective function of maximizing the importance of the approximate join result. We introduced efficient optimal offline and effective, lightweight online algorithms and showed that previous load shedding techniques are insufficient for the objective function. Avenues of future work include exploring other useful QoS-based objective functions for window join approximation and incorporating our methods into complex continuous query processing frameworks.

7. REFERENCES

[1] Abadi, D. J., Carney D., Centintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB Journal*, 12(2), 2003, 120–139.

[2] Apers, P. and Wilschut, A. Dataflow query execution in a parallel main-memory environment. In *Proc. 1st Int. Conf. on Parallel and Distributed Information Syst.*, 1991, 68–77.

[3] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. *Models and issues in data stream systems*. In *Proc. ACM PODS*, 2002, 1–16.

[4] Bonnet, P., Gherke, J., and Seshadri, P. Towards Sensor Database Systems. In *Proc. 2nd Int. Conf. On Mobile Data Management*, 2001, 3–14.

[5] Burger, J., Naughton, J., and Viglas, S. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *Proc. VLDB Conf.*, 2003, 285–296.

[6] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD Conf.*, 2000, 379–390.

[7] Cortes, C., Fisher, K., Pregibon, D., Rogers, A., and Smith, F. Hancock: A Language for Extracting Signatures from Data Streams. In *Proc. ACM SIGKDD Conf.*, 2000, 9–17.

[8] Das, A., Gerkhe, J., and Riedewald, M. Semantic Approximation of Data Stream Joins. In *IEEE TKDE*, 17(1), 2005, 44–59.

[9] Fegaras, L., Maier, D., Sheard, T., and Tucker, P. Exploiting Punctuation Semantics in Continuous Data Streams. In *IEEE TKDE*, 15(3), 2003, 555–568.

[10] Franklin, M. J. and Urhan, T. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin*, 23(2), 2000, 27–33.

[11] Goldberg, A. V. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. In *J. Algorithms,* 22(1), 1997, 1–29.

[12] Guha, S., Indyk, P., Muthukrishnan, S., and Strauss, M. Histogramming Data Streams with Fast Per-Item Processing. In *Proc. 29th Int. Colloquium on Automata, Languages, and Programming*, 2002, 681–692.

[13] Golab, L. and Ozsu, M. T. Issues in Data Stream Management. In *ACM SIGMOID Record*, 32(2), 2003, 5–14.

[14] Golab, L. and Ozsu, M. T. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. VLDB Conf.*, 2003, 500–511.

[15] Hellerstein, J., Madden, S., Raman, V., and Shah, M. Continuously Adaptive Continuous Queries Over Streams. In *Proc. ACM SIGMOID Conf.*, 2002, 49–60.

[16] Kang, J., Naughton, J., and Viglas, S. D. Evaluating Window Joins over Unbounded Streams. In *Proc. ICDE Conf.*, 2003, 341–352.

[17] Naughton, J. and Viglas, S. Rate-Based Optimization for Streaming Information Sources. In *Proc. ACM SIGMOD Conf.*, 2002, 37–48.

[18] O'Callaghan L., Mishra N., Meyerson A., Guha S., and Motwani R. Streaming-data algorithms for high quality clustering. In *Proc. ICDE Conf.*, 2002, 685.

[19] Takaoka, T. O(1) Time Algorithms for Combinatorial Generation by Tree Traversal. In *Computer Journal,* 42(5), 1999, 400–408.