Proceedings of CASCON'2007, pp 188-202, Canada, Oct. 2007

Window Query Processing for Joining Data Streams with Relations *

Kristine Towne[†] Qiang Zhu[†] Calisto Zuzarte[‡] Wen-Chi Hou[♯]

[†]Department of Computer and Information Science
The University of Michigan, Dearborn, MI 48128, USA

[‡]IBM Toronto Laboratory, Markham, Ontario, Canada L6G 1C7

[♯]Department of Computer Science
Southern Illinois University, Carbondale, IL 62901, USA

Abstract

Query processing for data streams raises challenges that cannot be directly handled by existing database management systems (DBMS). Most related work in the literature mainly focuses on developing techniques for a dedicated data stream management system (DSMS). These systems typically either do not permit joining data streams with conventional relations or simply convert relations to streams before joining. In this paper, we present techniques to process queries that join data streams with relations, without treating relations as special streams. We focus on a typical type of such queries, called star-streaming joins. We process these queries based on the semantics of (sliding) window joins over data streams and apply a load shedding approximation when system resources are limited. A recently proposed window join approximation based on importance semantics for data streams is extended in this paper to maximize the total importance of the approximation result of a star-streaming join. Both online and offline approximation algorithms are discussed. Our experimental results demonstrate that the presented techniques are quite promising in processing star-streaming joins to achieve the maximum total importance of their approximation results.

Keywords: Database, streaming data, query processing, star-streaming join, sliding window query, load shedding.

1 Introduction

Data in the form of streams occur in many instances including data produced by sensors, Internet traffic, financial tickers, on-line auctions and transaction logs [6, 9, 10]. Applications utilizing data streams are becoming increasingly important.

Data streams are viewed as append-only sequences of data tuples where all tuples of the same stream have the same schema. These streams are often continuous and unpredictable as data can arrive sporadically and in bursts. Traditional approaches to query processing in a relational database management system (DBMS) are inadequate for streaming data. Data stream management systems (DSMS) have been developed to accommodate continuously arriving data stream tuples from multiple outside sources and process timely results to multiple continuous queries posed on the

*Research was partially supported by the IBM Toronto Laboratory, the US National Science Foundation (under grants # CNS-0521142) and The University of Michigan.

streams.

Continuous queries are long-running and persistent [9]. These queries differ greatly from traditional one-time queries in a DBMS. A traditional query is evaluated once at the time it is issued and its result is based on the current state of the database, while a continuous query is constantly re-evaluated and its result is typically updated when new data arrives from one or more streams in the query. In this paper, we focus on discussing continuous queries involving equi-joins.

Because of the infinite nature of data streams, many traditional query operators cannot be applied. For example, the join operator is blocking (when applied to two data streams) and, therefore, must be replaced with a non-blocking operator [14]. Although there are several approaches to addressing this issue, windowing is a common approach adopted in many papers, which is also applied in this paper. A window placed over a data stream represents a finite subset of the stream at some time instant. Count-based and time-based windows are common types of windows mentioned in the literature. In this paper, we consider a time-based window of size T holding tuples that arrived during the last T time units. Such a (sliding) window has freely moving endpoints that move forward a unit step at each time instant.

An important concept of a sliding window is timestamps. Timestamps provide an ordering on stream tuples that involve time or sequence numbers. This ordering is used to determine which tuples belong in a window. Time-based sliding windows apply timestamps with expiration times. The endpoints of a sliding window move forward in time causing older tuples to expire (i.e., are no longer part of the window). This approach works well in most realistic applications as older tuples are no longer relevant to queries [14]. We apply the eager re-evaluation of expired tuples as discussed in [9].

Situations occur when it is desirable to query data streams along with one or more conventional relations. This is the issue to be addressed in this paper. Most DSMSs do not permit joining data streams with relations. In some instances [5, 1], such a join is handled by converting the relations into streams before the join is performed. The Aurora [1] and Telegraph CQ [5, 15] systems support queries with static relations, while the STREAM system supports queries with dynamic relations [11]. We consider the situation where relations are not converted but treated as conventional relations in a DBMS in which relation access methods can be utilized.

The STREAM system converts data streams to relations by placing a window on the stream. It then performs extended SQL (CQL) queries between the relation and data stream window [2]. Relations in STREAM are dynamic and allow updates that are timestamped, necessitating a window placed on the relation [11]. In our approach, we apply an active-time interval to each tuple in a relation. The active-time interval is not a timestamp but is used solely to determine the validity of a tuple. We do not impose a window onto the relation. The STREAM system treats relations and data streams equally, while our approach considers a relation to be a fact or lookup table/relation for the data streams.

High-speed and time-varying data streams strain the system's memory and CPU resources [7]. Storing infinite and continuous data streams in their entirety is impossible. There are two types of resource limitations. In the fast CPU case, stream tuples are processed as quickly as they arrive but memory resources are limited, causing some stream tuples in a window to be evicted/replaced before their expiration times. When the CPU is slow, as in the slow CPU case, the system cannot process input stream tuples as quickly as they arrive. Tuples are placed into a queue and then processed in the order they arrive according to their timestamps. If the queue fills, some tuples may be dropped. In this paper, we adopt the fast CPU model where memory is our primary limitation.

When system resources (CPU or memory) cannot keep up with the data stream demand, approximate queries have to be considered. To process such queries, load shedding techniques are typically adopted to drop selected stream tuples prematurely before they reach the window or after they have entered into the window but before they expire. Since an exact window query cannot be performed in such a case, the

goal of approximation is usually to maximize the size of the query result (i.e., minimizing the result error).

Random load shedding techniques [8, 4, 12, 3] randomly drop tuples from a data stream without regard to the output the tuples may produce when joined with another data stream. The semantic load shedding technique [7] avoids the randomness by choosing to drop tuples with low probabilities for matching tuples in the opposite data stream in order to maximize the query result size. A more recent load shedding technique based on importance semantics [13] further extends the semantic load shedding technique by considering the importance of the output produced. In fact, the semantic load shedding technique is a special case of the new load shedding technique when all input tuples are of the same importance.

The above load shedding techniques were designed to handle window joins between data streams. In this paper, we extend the load shedding technique based on importance semantics in [13] to handle window queries joining data streams with conventional relations. Because of space limitation, we focus on discussing one typical type of such window queries, called the star-streaming join, which joins one relation with one or more data streams. We leverage the "lookup/fact" relation to pre-filter stream tuples that will never produce output so that they are prevented from being entered into the joining window. This strategy allows surviving stream tuples to stay in the limited-size window longer, resulting in a higher possibility of matches. The goal of our technique is to maximize the total importance of the output tuples in the query result. Compared with the previous techniques, our approach is more general in two aspects: (1) The data streams are joined via the fact/lookup relation rather than directly. Such a relation typically represents the most relevant matches desired by the corresponding applications. (2) The adoption of the extended load shedding technique based on importance semantics allows the approximate query result to be more relevant to the query objective, as opposed to the popular semantic load shedding and random load shedding techniques.

The remainder of this paper is organized as follows. The definition of the star-streaming join and its processing model are introduced in Section 2. Several offline and online approximation algorithms to process such queries are presented in Section 3. Experimental results are reported in Section 4. Section 5 summarizes the conclusions and future work.

2 Star-Streaming Join

In this paper, we focus on discussing a typical type of query, called the star-streaming join, for joining data streams with conventional relations. We characterize this type of query and present a processing model for such queries in this section. Algorithms to evaluate these queries will be discussed in Section 3.

2.1 Query Structure and Processing Model

A star-streaming join is a query to join one relation F with one or more data streams R_1, R_2, ..., R_n (see Figure 1), denoted as $\bowtie (F, R_1, R_2, ..., R_n)$ $(n \geq 1)$. It is similar to a star-schema query in data warehouses. Relation F plays the role as the fact relation here, while the data streams R_i's play the role as the (unbounded) dimension relations. Relation F usually contains information about the related tuples from data streams that the users are interested in. For simplicity, we only consider $n = 1, 2$ in this paper. The cases for $n > 2$ can be handled in a similar way as $n = 2$.

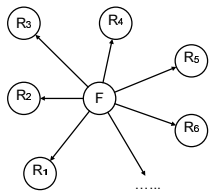


Figure 1: Structure of Star-Streaming Join

In all cases, we assume that one tuple arrives in a data stream at each time instant. The notation *r(j)* denotes a tuple that arrives in its data stream at time *j*. Each tuple arriving in a data stream has a format <*ts*, *sch*, *imp*> where

$ts \in \mathbf{N}$, the set of natural numbers, is the timestamp ordering; *sch* is the conventional schema of the stream; $imp \in \{ \text{ x } \mid \text{ x } \in \mathbf{R} \text{ and } 0 < \text{ x } \leq U \text{ }\}$ is the importance of the tuple predetermined by stream applications where $\mathbf{R}$ is the set of real numbers and U is the upper bound for importance. The importance, as described in [13], indicates how important this tuple is to the query source if output is produced. Without loss of generality, we assume *sch* contains only one (joining) attribute that also appears in (fact) relation F since other attributes of *sch*, if any, are irrelevant to our join computation. For the same reason, we assume that relation[1] F contains only the attributes for joining with the data streams. Hence the join condition for $\bowtie (F, R_1, R_2, ..., R_n)$ is the conjunction of equalities between the corresponding pair of joining attributes from relation F and data stream R_i for every $1 \leq i \leq n$. A tuple in the query result is the concatenation of the matching tuples from the respective data streams. The corresponding tuple in relation F is embedded/implied in such a result tuple. Note that a star-streaming join is a continuous query and its result changes over time as the data streams change.

For $n = 1$, star-streaming join $\bowtie (F, R)$ (i.e., $F \bowtie R$) is also called a semi-streaming join since one of the two operands is a data stream. This semi-streaming join at time t is defined as:

$$RS_1(t) = F \bowtie R = \bigcup_{j=0}^{t} \bigcup_{f \in F} \{f \bowtie r(j)\}, \quad (1)$$

where $r(j)$ is a tuple arriving in R at time j and $f \bowtie r(j)$ denotes an output tuple $o =< ts, sch, imp >$ if the values for the joining attributes in f and $r(j)$ match. $o.ts$ is the timestamp when o was produced, $o.sch$ is the matching value for the joining attributes, and $o.imp = r(j).imp$.

The (total) importance of the query result at time t in (1) is:

$$imp(R \bowtie F) = \sum_{o \in RS_1(t)} o.imp. \quad (2)$$

Since we adopt the fast CPU model, a semi-streaming join defined in (1) can always be computed exactly (i.e., no load shedding is needed) as follows. For each tuple $r(t)$ arriving in R at time t, we look up relation F, which is finite, for matches and produce output tuples if any. After this lookup, $r(t)$ can be discarded since it is no longer needed.

However, the same strategy cannot be applied to a star-streaming join for $n = 2$ (or higher), in which multiple unbounded (infinite) data streams are involved. A tuple r_1 arriving in one stream R_1 of the join cannot be discarded in the above way since it may match (via the fact relation F) a tuple arriving in the other stream R_2 of the join in the future. To resolve this problem, it is typically assumed that each tuple in a data stream R for such a join has a lifetime m. The join is calculated by imposing a sliding window R^w of size m over R. The window $R^w(j)$ of R at time j is defined as:

$$R^w(j) = \begin{cases} \{r(k) \mid j - m + 1 \leq k \leq j\}, \\ \qquad\qquad if\ j \geq m - 1, \\ \{r(k) \mid 0 \leq k \leq j\}, \\ \qquad\qquad if\ j < m - 1. \end{cases}$$

where $r(k)$ is the tuple arriving in R at time k. Note that window $R^w(j)$ is not full when $j < m - 1$. Tuple $r(i)$ in stream R is expired at time t if $r(i).ts + m \leq t$.

Based on this semantic, a star-streaming join $\bowtie (F, R, S)$ (i.e., $R \bowtie F \bowtie S$) at time t can be defined as follows:

$$RS_2(t) = R \bowtie F \bowtie S = \bigcup_{j=0}^{t} \bigcup_{k=j_0}^{j} \bigcup_{f \in F}$$
$$[\ \{r(j) \bowtie f \bowtie s(k)\} \cup \{s(j) \bowtie f \bowtie r(k)\}\ ], \quad (3)$$

where $j_0 = max\{0, j - m + 1\}$; $r(j) \bowtie f \bowtie s(k)$ denotes an output tuple $o =< ts, sch, imp >$ if the values for the joining attributes satisfy the join condition among $r(j)$, f and $s(k)$; $o.ts$ is the timestamp when o was produced; $o.sch$ consists of $r(j).sch$ and $s(k).sch$; and $o.imp = min\{r(j).imp, s(k).imp\}$; $s(j) \bowtie f \bowtie r(k)$ is defined similarly.

The (total) importance of the query result at time t in (3) is:

$$imp(R \bowtie F \bowtie S) = \sum_{o \in RS_2(t)} o.imp. \quad (4)$$

When there is enough memory to keep windows R^w and S^w with the size m, Formula (3)

[1] We assume that a relation contains no duplicate tuples.

calculates the exact result of the star-streaming join at time t. However, when the size of memory is not enough to keep all stream tuples before their expiration, an approximation result (i.e., a subset) has to be obtained via load shedding. A good window query approximation algorithm should maximize the total importance of the query result. We will present several such algorithms in Section 3.

2.2 Handling Dynamic Relations

The processing model discussed above is for static relations. A dynamic relation allows updates, insertions and deletions, although they occur infrequently compared with the changes in a data stream. The above model needs to be extended to handle a dynamic relation. For simplicity, we consider a dynamic relation with insertions and deletions only. An update can be simulated by a deletion followed by an insertion.

To handle such a dynamic relation F, we incorporate an active-time interval to each tuple in F to determine the state of F at a particular time instant. The active-time interval has the format of [*begin*, *end*) where *begin* is the time instant the tuple becomes active/valid (i.e., inserted) and *end* is the time instant at which the tuple becomes inactive/invalid (i.e., deleted). At time t, any tuple f in F with $f.begin \leq t < f.end$ is an active/valid tuple. The (active/valid) state of F at t consists of all active tuples at t. Tuples stored in relation F prior to the start of the data stream (i.e., prior to $t = 0$) are assumed to have *begin* = -1, and any tuple inserted afterwards will have the timestamp of the insertion as its *begin*. Tuples that have not been deleted from F have *end* = ∞ while a deleted tuple has the timestamp of the deletion as its *end*.

When performing a semi-streaming join between a data stream R and a dynamic relation F following Formula (1), $f \in F$ in the formula should be interpreted as f is an active tuple in F at time j. In other words,

$$f \bowtie r(j) \textit{ is an output tuple} \iff f.begin \leq j < f.end \ \textit{ and } \ f.a = r(j).a, \quad (5)$$

where a is the joining attribute between R and F.

Similarly, when performing a star-streaming join for a dynamic relation F and two data streams R and S following Formula (3), $f \in F$ in the formula should be interpreted as f is an active tuple in F at both times j and k. In other words,

$$\begin{aligned} r(j) \bowtie f \bowtie s(k) \textit{ is an output tuple} \iff \\ f.begin \leq j < f.end \ \textit{ and } \\ f.begin \leq k < f.end \ \textit{ and } \\ r(j).a = f.a \ \textit{ and } \ f.b = s(k).b, \end{aligned} \quad (6)$$

where a is the joining attribute between R and F, and b is the joining attribute between F and S. The condition for output tuple $s(j) \bowtie f \bowtie r(k)$ in Formula (3) can be interpreted similarly.

The concept of an active-time interval and the formulas to determine the validity of a tuple in the fact relation are very important to our model. Essentially, they suggest that insertions and deletions of tuples for relation F at time t should not affect the stream tuples that have already entered their streams at an earlier time i before t.

Clearly, an active-interval is different from the timestamp of a stream tuple. Timestamps are uniquely assigned to each tuple of a data stream and impose an ordering among the tuples. The active-time interval represents the valid duration of a relevant tuple in the fact relation, which may not be unique, nor does it impose an ordering or a window on the relation.

Based on the output tuple condition in (6), we pre-filter some useless tuples from a stream window. We determine that a stream tuple will not produce any output if the tuple does not match with an active tuple in relation F. When a stream window is filled, a newly arriving stream tuple will either be discarded or replace a stream tuple in the window. By not adding those tuples that do not match with F in the window, we allow the current tuples in the window to stay longer, possibly producing more output tuples.

2.3 Utilizing Relation Access Methods

Since the fact relation F is a conventional relation managed by the DBMS, the efficient rela-

tion access methods in the DBMS can be utilized to access F. To process semi-streaming joins, one can use any access method such as the sequential scan, the index scan, the hash table access or the binary search to access F. However, for star-streaming joins with two (or more) data streams, only the sequential scan and the index scan can be used by both data streams to look up F. The hash table access and the binary search can only be used by one of the data streams to look up F since two data streams may join on different attributes of F. Since F is relatively stable, it is beneficial to perform some preprocessing such as removing irrelevant attributes, deleting duplicates and creating relevant indexes before running a continuous star-streaming query.

Data streams are typically dynamical, read-once and unbounded. Hence, indexing, sorting or hashing is not suitable for accessing them unless archived data streams are considered. Data streams are usually accessed via the sequential scan. As a result, some traditional efficient join algorithms such as the sort-merging join are inapplicable to star-streaming queries. The pre-filtering strategy using the fact relation suggested in the last subsection is an effective way to optimize the access for the data streams. In fact, the fact relation can be used to provide other useful pre-filtering information, for example, by providing priorities to stream tuples for online processing.

If the number of tuples in relation F becomes too large, it is desirable to use a subset or reduced result of vital tuples from F so that the tuples arriving in the joining data streams can be processed in time. Reducing the effective size of the relation could be performed in several ways such as using statistics or other approaches. This is a topic for our future work.

3 Approximation Algorithms

As mentioned earlier, when the size of memory is not enough, we have to approximate a star-streaming join via load shedding. We present two types of approximation algorithms: one for offline and the other for online. The offline algorithm is useful for archived data streams, while the online ones are for real-time dynamic data streams. The goal for all approximation algorithms is to maximize the total importance of a query result. We focus on discussing star-streaming joins for one fact relation F and two data streams R and S. The discussion can be generalized to handle star-streaming joins involving $n > 2$ data streams.

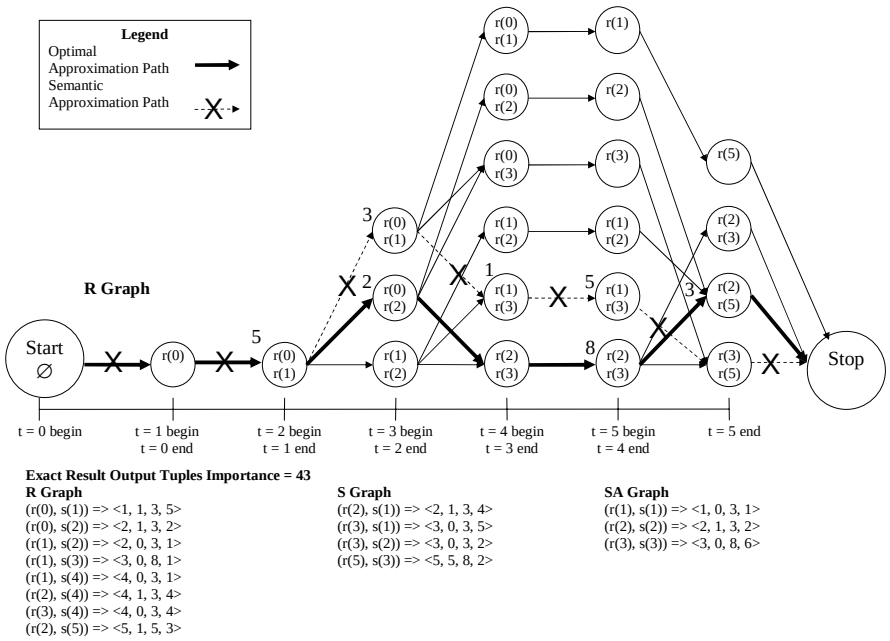
3.1 An Offline Optimal Approximation Algorithm

An effective offline optimal approximation algorithm for window joins was introduced in prior work [13]. However, this algorithm cannot be applied to the star-streaming joins defined in Section 2, since it does not allow a conventional relation in the query. In this section, we develop an offline approximation algorithm to process star-streaming joins by extending the algorithm in [13].

3.1.1 Join Memory State Graph

To obtain a query result with maximum total importance for a given memory size M (i.e., the window size for each data stream is $\lfloor M/2 \rfloor$), the algorithm applies the dynamic programming approach. It constructs an intermediate data structure, called the join memory state graph, and uses it to identify the optimal solution.

The algorithm needs to determine the (window) memory states that produce the largest approximation importance at each time instant. To determine which stream tuples should be contained in a window at each time instant, we construct a directed graph, the join memory state graph, for each data stream. We have three memory state graphs: R graph, S graph, and SA graph. Figure 2 shows an example of R graph, which illustrates the possible memory states for a given data stream R. Each memory state represents one possible combination of stream tuples in the relevant window at a particular time instant. Figure 3 shows an example of S graph, which illustrates the possible memory states for a given data stream S. If two input stream tuples arriving at the same time instant match with each other (via a tuple in fact relation F), they are inserted into



Exact Result Output Tuples Importance = 43

R Graph
(r(0), s(1)) => <1, 1, 3, 5>
(r(0), s(2)) => <2, 1, 3, 2>
(r(1), s(2)) => <2, 0, 3, 1>
(r(1), s(3)) => <3, 0, 8, 1>
(r(1), s(4)) => <4, 0, 3, 1>
(r(2), s(4)) => <4, 1, 3, 4>
(r(3), s(4)) => <4, 0, 3, 4>
(r(2), s(5)) => <5, 1, 5, 3>

S Graph
(r(2), s(1)) => <2, 1, 3, 4>
(r(3), s(1)) => <3, 0, 3, 5>
(r(3), s(2)) => <3, 0, 3, 2>
(r(5), s(3)) => <5, 5, 8, 2>

SA Graph
(r(1), s(1)) => <1, 0, 3, 1>
(r(2), s(2)) => <2, 1, 3, 2>
(r(3), s(3)) => <3, 0, 8, 6>

Figure 2: Example of R Graph

a memory state[2] in SA graph (see Figure 3). The vertices of R graph and S graph represent different possible window memory states at different time instants. The vertices of SA graph represent the cases when two input stream tuples arriving at the same time instant match with each other. Edges are created between two memory states of consecutive time instants and represent decisions to move from one memory state to the next. The weight for an (incoming) edge in R (S) graph is calculated as the total importance for all output tuples that can be produced by the tuples in the corresponding memory state (excluding the tuple arriving in this stream at the current time instant) that match the tuple arriving in the other stream at the current time instant. For presentation clarity, only non-zero edge weights along the optimal path and the semantic load shedding path (shown for comparison) are shown in the figures. The weight for an (incoming) edge in SA graph is the importance of the output tuple produced by the pair of the matching stream tuples (via F) in the corresponding memory state.

Different window memory states are possible at each time instant based on previous window states and the following possible decisions when a new stream tuple *r(t)* enters the system: (1) the window (memory) is not full and *r(t)* is admitted; (2) the window is full and *r(t)* replaces a non-expired tuple; (3) join memory is full or not full and *r(t)* is not admitted. Next states are constructed from previous states and these decisions. Recall that we apply the eager re-evaluation approach for expired tuples. Hence, expired tuples are always removed from the window at the beginning of each time instant before a new tuple is admitted.

In the following discussion, let M denote the (total) join memory size, m denote the tuple lifetime, and N denote the length of a (archived) data stream. In our example, we have stream length $N = 6$, total memory $M = 4$, and tuple lifetime $m = 4$. Since we have two data streams, each stream has a window (memory) size of $M/2 = 2$. Recall that the format of a tuple in a data stream is <*ts*, *sch*, *imp*> and the format of a tuple in the fact relation is <*R sch*, *S sch*, *active-time interval*>, assuming no irrelevant attributes in the fact relation or any data stream.

Assume stream R contains the following tuple sequence: <0, 1, 5>, <1, 0, 1>, <2, 1, 4>, <3, 0, 8>, <4, 2, 3>, <5, 5, 2>; stream S con-

[2]In fact, no memory/window is involved for SA graph. Using name "memory state" is just for consistency.

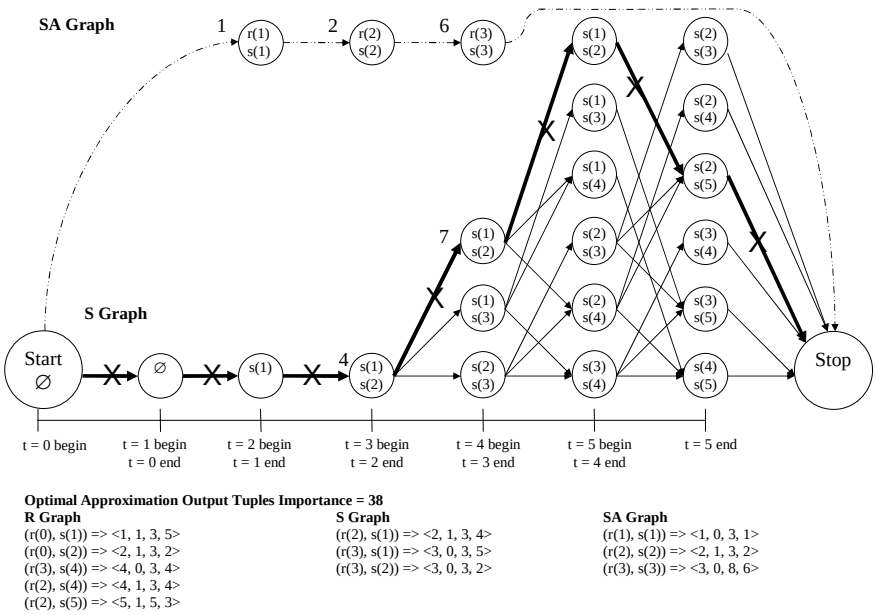


Figure 3: Example of S Graph and SA Graph

tains <0, 1, 1>, <1, 3, 5>, <2, 3, 2>, <3, 8, 6>, <4, 3, 4>, <5, 5, 3>; and relation F contains tuples <0, 3, [-1, ∞)>, <1, 5, [-1, ∞)>, <0, 8, [-1, ∞)>, <4, 5, [-1, ∞)>, <1, 3, [-1, 5)>, <5, 8, [3, ∞)>. Initially, relation F contains five tuples with active-time interval [-1, ∞), i.e., the tuples are active before the query starts and remain active until they are deleted. At time $t = 5$, tuple <1, 3, [-1, ∞)> is deleted, producing <1, 3, [-1, 5)>. At time $t = 3$, tuple <5, 8, [3, ∞)> is inserted.

The start vertex represents the empty memory state for the window before input stream tuples arrive. The stop vertex represents the state indicating all input stream tuples have been considered. The pre-filtering effect of fact relation F on the data streams is reflected in each of these join memory state graphs. The number of memory states at each time instant varies, and the number of stream tuples in each memory state also varies.

Input tuples from both data streams arrive at discrete time instants, beginning at $t = 0$. Each stream tuple u is assigned a timestamp ts as it arrives in the steam. We assign the timestamp as the discrete time instant at which the tuple u arrives in the stream and calculate the expiration time as $u.ts + m$.

The processing of a star-streaming join incrementally constructs the join memory state graphs at each time instant t when stream tuples *r(t)* and *s(t)* arrives in their data streams. Only active tuples in fact relation F at time t are considered in the join.

For our example, at time $t = 0$, *r(0)* matches with a tuple in F, but *s(0)* does not. In R graph, a memory state from *Start* is created, and *r(0)* is added to the memory state at $t = 0$ (end) based on Decision (1). A memory state in S graph is also created although *s(0)* is not added into it because of Decision (3).

At time $t = 3$ (end), memory states in S graph are created from the memory state at $t = 2$ (end). Since *s(3)* matches with a tuple in F, the window (memory) at state $t = 2$ (end) is full, and none of the tuples have yet been expired, the new memory states for $t = 3$ (end) are created based on Decision (2) or (3) (representing all feasible decisions in this case). Also, since $r(3)$ and $s(3)$ match with each other via a tuple in F, a new memory state containing $r(3)$ and $s(3)$ is created in SA graph at $t = 3$ (end).

Note that no duplicate memory states should be created at any time instant. If a required memory state has already been created at a time instant, only necessary new edges are added.

An edge weight in R (S) graph can be calculated by summing the importance values of the new output tuples created by all tuples (excluding the tuple arriving in this stream at that particular time instant) in the memory state that match (via F) the tuple arriving in the other stream at that particular time instant. For example, since $r(1)$ in memory state $\{r(1), r(3)\}$ at time $t = 3$ (end) matches with *s(3)* via a tuple in F, an output tuple $(r(1), s(3)) = <3, 0, 8, 1>$ with importance 1 will be produced. Hence the value 1 is added to the weight for each edge directed to memory state $\{r(1), r(3)\}$ in R graph. Note that, although $r(3)$ also matches with $s(3)$ via a tuple in F, the importance of the output tuple produced is assigned to the weight of the corresponding edge in SA graph.

An approximation result to the star-streaming join consists of the output tuples generated along a path from *Start* to *Stop* in each R graph and S graph and the output tuples represented by the path in SA graph. The total importance of the approximation result is the sum of the total edge weights of the paths selected for the approximation in the three graphs. The optimal approximation result of the star-streaming join is obtained by choosing the path with the largest total weight in each graph. The memory states along the chosen path record how the query result is calculated.

For our example in Figures 2 and 3, the exact result for the star-streaming join contains 15 output tuples with a total importance of 43. The optimal approximation result obtained by our algorithm contains 11 output tuples with a total importance of 38. The optimal approximation result obtained by the semantic load shedding approximation technique in [7] contains 12 output tuples but has a total importance of 34. Clearly, our algorithm produces a better approximation result to the query in terms of the achieved total importance.

3.1.2 Algorithm Description

In our following optimal star-streaming join approximation algorithm, the join memory state graphs are generated and the optimal approximation path is found in each graph.

In fact, the algorithm consists of three smaller algorithms: ComputeMaxImportance, ComputeSAImportance, and DisplayMemoryStates. Algorithm ComputeMaxImportance constructs R (S) graph and computes the total importance/weight for the optimal (with the largest total weight) path in the graph. It needs to be invoked twice: once for constructing R graph and another for constructing S graph. During its execution, it invokes three functions: FindQualifiedFactTuples, GetWeight, and InsertState. Function FindQualifiedFactTuples essentially finds the set of tuples in fact relation F that match with a given stream tuple. Function GetWeight calculates the edge weight for a given memory state. Function InsertState inserts a new state into the join memory state graph for a data stream. Algorithm ComputeSAImportance constructs SA graph and computes the total importance/weight on the path in the graph. If the memory states of the optimal paths in graphs are to be displayed, Algorithm DisplayMemoryStates is invoked for each graph after the graphs have been constructed.

In the discussion of the above algorithms, the following notation is used. Variable $MS[t]$ holds the memory states available at time instant t. Variable field *MI* of a memory state is the total importance/weight of the current optimal path from the start vertex to this state. Variable field *prev* of a memory state points to the previous memory state in the current optimal path. *StopVertex* and *StartVertex* are special memory states that do not contain stream tuples but signify the start and end of the graph. For simplicity, we assume that streams R and S are of the same length. We also assume that relation F is small enough so that the fast CPU model can be applied with a sequential scan.

The details of the algorithms and functions are given below.

ALGORITHM 3.1 : **ComputeMaxImportance**
Input: Data stream R, data stream S, and relation F.
Output: *StopVertex* for R graph and the total importance of the optimal path.
Method:
//when invoked for stream S, swap R with S
1. $t = 0$, $MS[t] = \emptyset$;
2. insert *Start* into $MS[t]$ as *StartVertex*;
3. **while** $t <$ length of stream R
4. $MS[t+1] = \emptyset$;
5. *QTuples* = FindQualifiedTuples(*s(t)*, F, t);
6. **if** $t = 0$ **then**
7. create empty memory state y from *StartVertex*

```
 8.     y.MI = GetWeight(y, QTuples);
 9.     if r(t) has a valid match in relation F then
10.       insert r(t) into y;
11.     end if
12.     y.prev = StartVertex;
13.     insert y into MS[t+1];
14.   else
15.     ∀ memory state x ∈ MS[t]
16.       let yy be a copy of x;
17.       remove expired stream tuple in yy if any;
18.       create copy of yy as new memory state y;
19.       y.MI = x.MI +
             GetWeight(y, QTuples);
20.       if window for yy is not full and r(t)
           has a valid match in relation F then
21.         insert r(t) into y;
22.       end if
23.       y.prev = x;
24.       InsertState(y, MS[t + 1])
25.       if window for yy is full and r(t) has
           a valid match in relation F then
            //generate memory states by replacing each
            //tuple in yy with r(t)
26.           ∀ stream tuple z in yy
27.             create copy of yy as new memory state y;
28.             remove z from y;
29.             y.MI = x.MI +
                   GetWeight(y, QTuples);
30.             insert r(t) into y;
31.             y.prev = x;
32.             InsertState(y, MS[t + 1])
33.       end if
34.     end if
35.     t = t + 1;
36.   end while
37.   MS[t] = {Stop} as {StopVertex};
38.   ∀ memory state x ∈ MS[t - 1]
39.     if x.MI ≥ StopVertex.MI then
40.       StopVertex.MI = x.MI;
41.       StopVertex.prev = x;
42.     end if
43.   return StopVertex, StopVertex.MI.
```

In Algorithm 3.1, lines 1 - 2 initialize relevant variables and create the start vertex. Lines 3 - 36 construct the memory states between the start and stop vertexes. Line 5 finds the tuples in fact relation F that match with the current stream tuple $s(t)$. Lines 6 - 13 construct a memory state directly from the start vertex. Lines 14 - 34 construct memory states at time $t+1$ from each memory state at time t. Line 17 removes expired stream tuple(s). Lines 20 - 22 admit stream tuple $r(t)$ into the window based on Decision (1). The new memory state containing $r(t)$ is created in lines 23 - 24. However, if $r(t)$ has no match in relation F or the window is full, a new memory state is created by copying non-expired tuples from previous memory state x in lines 18 - 24 based on Decision (3). Lines 25 - 33 construct new memory states based on Decision (2). The stop vertex is constructed in lines 37 - 42.

FUNCTION 3.1 : **InsertState**
Input: New memory state ms and available memory states $MS[t+1]$ at the time instant $t+1$.
Output: Revised memory states at time $t+1$.
Method:

```
     //determine the path of largest importance
 1.  if ∃ memory state z ∈ MS[t + 1] where set
      of tuples in z = set of tuples in ms then
 2.    if ms.MI > z.MI then
 3.      z.MI = ms.MI;
 4.      z.prev = ms.prev;
 5.    end if
 6.  else
 7.    insert ms into MS[t + 1];
 8.  end if
```

Function 3.1 determines if the new memory state is a duplicate. If the new state is a duplicate and has a larger MI than the current state, the edge directed to this state is updated to reflect a better path. If the new state is not a duplicate, the state is inserted into the graph in line 7.

FUNCTION 3.2 : **GetWeight**
Input: Memory state ms (excluding the tuple arriving in the corresponding stream at the current time instant) and a set $QTuples$ of matched tuples (with augmentation) in F.
Output: Edge weight for an edge directed to ms.
Method:

```
 1.  EdgeWeight = 0;
 2.  ∀ stream tuple x in ms
 3.    ∀ tuple y in QTuples
 4.      if x matches with y and y.begin ≤ x.ts
          and x.ts < y.end then
 5.        if x.imp > y.imp then
 6.          EdgeWeight = EdgeWeight + y.imp;
 7.        else
 8.          EdgeWeight = EdgeWeight + x.imp;
 9.        end if
10.     end if
11.   return EdgeWeight.
```

Function 3.2 calculates the total importance of output tuples generated by stream tuples in ms. If ms is empty, it returns 0. Line 4 checks if the current stream tuple matches with the current active/valid relation tuple. Lines 5 - 9 determine the importance of the output tuple generated by the current stream tuple and a matching stream tuple from the other stream via the current relation tuple and then add the importance to the edge weight.

FUNCTION 3.3 : **FindQualifiedFactTuples**
Input: A stream tuple st, relation F, and time t.
Output: Active tuples (augmented with st.imp) in F that matches with st.
Method:

```
 1.  QTuples = ∅
 2.  ∀ tuple x in relation F
 3.    if x.begin ≤ t and x.end > t then
 4.      if x matches with st then
 5.        insert <x, st.imp> into QTuples;
 6.      end if
 7.    end if
 8.  return QTuples.
```

Function 3.3 finds all tuples in F that match with st and augments them with st.imp. Only active/valid tuples in F are considered.

ALGORITHM 3.2 : **ComputeSAImportance**
Input: Data stream R, data stream S, and relation F.
Output: $StopVertex$ for SA graph and the total importance of its path.
Method:
1. t = 0;
2. set x as $StartVertex$ and x.MI = 0;
3. set y.$prev$ = x and y.MI =0;
4. **while** t < stream length
5. **if** r(t) matches with s(t) via F **then**
6. create memory state y from x for r(t) and s(t);
7. **if** r(t).imp > s(t).imp **then**
8. y.MI = x.MI + s(t).imp;
9. **else**
10. y.MI = x.MI + r(t).imp;
11. **end if**
12. y.$prev$ = x;
13. x = y;
14. **end if**
15. $t = t + 1$;
16. **end while**
17. $StopVertex.prev$ = y;
18. $StopVertex.MI$ = y.MI;
19. **return** $StopVertex$, $StopVertex.MI$.

Algorithm 3.2 checks if *r(t)* and *s(t)* match with each other via an active tuple in relation F for every time instant t. If so, a memory state in the SA graph is created.

ALGORITHM 3.3 : **DisplayMemoryStates**
Input: $StopVertex$ of a graph.
Output: The memory states on the optimal path traced from $StopVertex$ are displayed.
Method:
1. set x = $StopVertex.prev$
2. **while** x $\neq$ $StartVertex$
3. display x
4. x = x.prev
5. **end while**.

Algorithm 3.3 backtracks through the graph following the *prev* field of each vertex, displaying the optimal path in the graph.

3.2 Online Approximation Algorithms

An offline approximation algorithm can obtain the optimal approximation result. However, it requires prior knowledge about the input data streams. Hence, it is suitable for joining archived data streams. In many applications, online data stream processing is needed. In such a case, heuristics are typically employed to decide which stream tuples to retain or drop during query processing. We extend the heuristics suggested in [13] to handle star-streaming joins ($n = 2$) and study the effectiveness of the algorithms applying these heuristics.

An online approximation algorithm greedily retains tuples with higher priorities and evicts tuples (including the new stream tuple) with lower priorities when load shedding is required. For star-streaming join processing, a new stream tuple that does not match with an active tuple in fact relation F is always dropped whether the window is full or not. We assume that such a tuple has a priority value -1, indicating an immediate drop (pre-filtering). For a stream tuple $r(t)$ that has a valid match in F at time t, a priority $P(r(t))$ is assigned to it based on its importance ($r(t).imp$), its matching probability with the opposite stream ($mprob(r(t))$), and its expiration time (i.e., $expr(r(t)) = r(t).ts + m$). Hence,

$$P(r(t)) = f(r(t).imp,\ mprob(r(t)),\ expr(r(t))), \quad (7)$$

where $f()$ is some function and $mprob()$ is estimated.

When such a tuple arrives in the stream and the window is full, a heuristic-based algorithm will examine the priorities of tuples in the window and the new tuple. It drops the tuple with the lowest priority.

For our star-streaming joins, we also apply the following rule to perform dynamic prefiltering. For an existing stream tuple $t(i)$ in the window, its priority is reset to -1 at time t if the following condition holds:

$$P(r(i)) = -1 \quad if\ max\{f.end \mid f \in F \wedge f\ is\ a\ valid\ match\ for\ r(i)\} \leq t\ . \quad (8)$$

Under the fast CPU model, the match (via F) of $r(i)$ with tuples in the window for the opposite data stream has already been examined. Condition (8) indicates that all valid matches in F for $r(i)$ have been deleted at or before time t, implying that $r(i)$ will never be able to match any stream tuple arriving in the opposite stream via F from now on. Therefore, $r(i)$ is useless and should be dropped immediately (i.e., having -1 priority).

Different heuristics calculate $P(r(t))$ in (7) differently, i.e., applying different $f()$'s. We consider the following four heuristics. Note that the filtering rule in (8) is always applied together with each of these heuristics.

Static Importance Heuristic

In the Static Importance heuristic (SIMP), the priority is given by the stream tuple's importance:

$$P(r(t)) = r(t).imp.$$

This heuristic chooses a stream tuple with a higher importance to retain in hope that if the tuple produces an output tuple with a tuple from the opposite stream via fact relation F, the resulting output tuple will have a higher importance. Ties among the tuples with the same lowest priority are broken by dropping the oldest tuple with the lowest priority.

Static Importance Probability Heuristic

The Static Importance Probability heuristic (SIMPPROB) assigns the priority to a new stream tuple as the product of the tuple importance and the number $mn()$ of matches (via F) in the opposite window:

$$P(r(t)) = r(t).imp \ * \ mn(r(t)).$$

In other words, $mprob(r(t))$ in (7) is estimated by $mn()$.

Ties among tuples with the lowest priority are broken by dropping the tuple with the lowest importance. Ties among those tuples with the lowest priority and lowest importance are broken by dropping the tuple with the fewest matches. Ties between tuples with the lowest priority, lowest importance and fewest matches are broken by dropping the oldest tuple.

Dynamic Importance Probability Heuristic

The Dynamic Importance Probability heuristic (DIMPPROB) assigns the priority to a new stream tuple as calculated in heuristic SIMPPROB. However, this priority is then re-calculated at each time instant to reflect the changing number of matches in the opposite window, i.e., $mn()$ is dynamically re-calculated. Priority ties are broken in the same way as SIMPPROB.

Dynamic Gain Loss Heuristic

The Dynamic Gain Loss heuristic (DGL) assigns the priority to a new stream tuple in the same way as heuristic SIMPPROB. However, the priority is cumulatively re-calculated at each time instant to reflect the changing number of matches in the opposite window and the lifetime of the tuple. If the tuple produces an output tuple at a time instant t, the priority is increased by the product of the tuple importance, the number of matches in the opposite window and the lifetime of the tuple as:

$$P(r(i)) = P'(r(i)) \ + \ [\ r(i).imp \ * \ mn(r(i)) \\ * \ (expr(r(i)) \ - \ t) \ / \ \alpha) \ ],$$

where α is a constant and $P'(r(i))$ is the priority from the previous time instant. If the tuple does not produce any output at a time instant t, the priority is decreased by a constant amount β, that is not dependent on the tuple's age:

$$P(r(i)) = P'(r(i)) \ - \ \beta.$$

Note that $P(r(i))$ is set to 0 if a negative value is obtained on the right hand side. Priority ties are broken in the same way as heuristic SIMPPROB.

4 Experiments

Experiments were conducted to evaluate the performance of our offline and online algorithms. The typical experimental results are reported in this section.

4.1 Experiment Setup

We use the same notations as in Section 3.1.1 where the memory size is M, the window (memory) size for each stream is $M/2$, each stream tuple has a lifetime m, the stream length is N, and the fact relation size is $|F|$.

We use OSSJ to denote our offline star-streaming join optimal approximation algorithm. Since no existing load shedding techniques can be directly applied to star-streaming joins, for the comparison, we extended two popular approximation techniques, i.e., the semantic load shedding and the random load shedding, to handle the star-streaming joins. We use SJA-S and RAND-S to denote these two extended techniques, respectively, in the following discussion. As a reference base, the exact computation (denoted by EXACT), assuming the memory is large enough (i.e., $M = 2 * m$) to compute the exact result of a star-streaming join, is also included in the experiments. For the online algorithms using the relevant priority heuristics together with special pre-filtering

rules for star-streaming joins, we use the priority heuristic names followed by '-S' to denote the corresponding algorithms. For example, the online algorithm using priority heuristic SIMP is denoted by SIMP-S.

The data streams for the experiments were generated as follows. Data stream R was generated using the 1.0 zipf distribution, and data stream S was generated using the uniform distribution. Uncorrelated importance values for the data streams were generated in such a way that smaller importance values were assigned to tuples occurring frequently while larger importance values were assigned to tuples occurring infrequently. Tuples in fact relation F were generated by the uniform distribution.

The experimental programs were implemented in C++. The platform used in our experiments was a 1.80 GHz Intel® Celeron Toshiba Satellite machine with 512 MB RAM running Windows® XP.

4.2 Evaluation of Offline Algorithm

One set of experiments were conducted to evaluate the efficacy of our offline algorithm. Figure 4 shows the comparison of the total importance of typical query results obtained by our algorithm OSSJ, SJA-S and RAND-S. The exact results are also included as a reference base. In the experiments, M varies while $m = 10$, $N = 5000$, and $|F| = 250$.

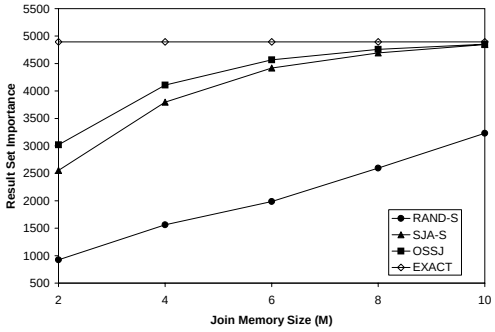


Figure 4: Efficacy Comparison for Offline Approximation

From the figure, we can see that the approximation error decreases as the join memory size increase for all techniques. We observed from our experiments that, although SJA-S produces larger query results than OSSJ, the latter produces query results with a greater importance than the former for all memory sizes. For example, for $M = 4$, the query result obtained by OSSJ has the importance of 4107, while the query result obtained by SJA-S has the importance of 3797. The query result size for OSSJ is 2184, and the query result size for SJA-S is 2221. This demonstrates that the objective to maximize the query result size in SJA-S may not lead to an optimal query result of the highest importance. As memory size increases SJA-S converges upon OSSJ, and both techniques converge upon EXACT quickly. Both OSSJ and SJA-S outperform RAND-S for all memory sizes since RAND-S does not approximate the query result based on its size or its importance.

Pre-filtering data streams utilizing the fact relation is a key strategy adopted in our algorithm OSSJ to improve its efficiency. To observe the pre-filtering effect on the performance of OSSJ, we conducted another set of experiments comparing the running times of the algorithm with and without pre-filtering. Figure 5 shows the comparison result. In the experiments, $N = 5000$, $M = 10$, $m = 10$, and $|F|$ varies. Each running time is the average of three executions.

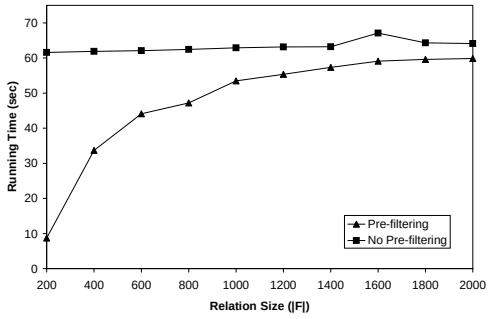


Figure 5: Performance Effect of Pre-filtering

From the figure, we can see that the algorithm with pre-filtering outperforms the one without pre-filtering. Its performance improvement increases as the fact relation size decreases since the pre-filtering power increases. Fortunately, this is typically the case in reality. The fact relation typically keeps information on

a small number of value combinations from two joining data streams.

4.3 Evaluation of Online Algorithms

We also conducted a set of experiments to evaluate the efficacy of the four online heuristic-based algorithms. We compare the importance of query results obtained from them with those obtained from RAND and EXACT. In the experiments, $N = 5000$, $m = 50$, $|F| = 250$, and M varies.

Figure 6 shows the comparison result. From the figure, we can see that DGL-S has the best performance since it takes the tuple importance, the number of matches in the opposite stream window and the tuple lifetime into consideration. DIMPPROB-S and SIMPPROB-S have a very similar performance. They generally outperform SIMP-S since both techniques take the tuple importance and the number of matches in the opposite stream window into consideration while SIMP-S does not consider the number of matches from the opposite stream window. All the above algorithms significantly outperform RAND-S (especially when memory size is small) as the latter does not take the tuple importance, the number of matches of the opposite stream or the tuple lifetime into account. As memory size increases all algorithms converge upon EXACT.

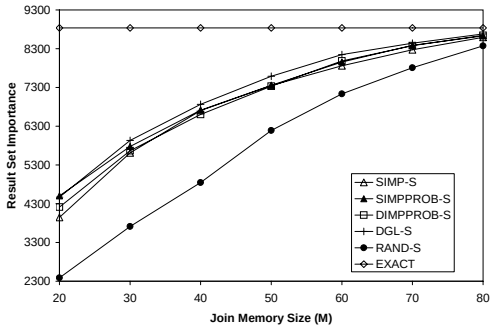


Figure 6: Efficacy Comparison for Online Approximation

5 Conclusions

This paper examines the issues for processing star-streaming joins, which join multiple data streams with one fact relation. These queries are processed based on the semantics of sliding window joins over data streams. When system resources are limited, a load shedding approximation is applied. The objective function of the approximation is to maximize the total importance of a query result. One offline optimal approximation algorithm and four online heuristic-based approximation algorithms that aim to achieve this objective are presented. A pre-filtering strategy is incorporated into the algorithms to improve efficiency. Mechanisms to handle dynamic fact relations are introduced. Experimental results demonstrate that the algorithms discussed in this paper outperform some popular techniques with extensions for star-streaming joins.

Future work in this area includes investigating techniques to reduce the size of a large fact relation without losing key information, developing special access methods for a fact relation supporting star-streaming joins, studying other types of queries joining multiple data streams with conventional relations, and examining system issues in existing DBMSs to support data stream processing.

Acknowledgments

The authors would like to thank Adegoke Ojewole for his useful discussions and help for the work reported here.

About the Authors

Kristine Towne is a graduate student in the Department of Computer and Information Science at The University of Michigan, Dearborn, USA. She is a graduate research assistant with an IBM CAS fellowship. She received a B.Sc. in Computer Science from The University of Michigan, Dearborn, USA, in 2003. Her research interests include database query processing and data stream processing.

Qiang Zhu is a Professor in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. He received his Ph.D. in Computer Science from the University of Waterloo in 1995. Dr. Zhu is a principal investigator for a number of database research projects funded

by highly competitive sources including NSF and IBM. He has numerous research publications in various refereed journals and conference proceedings including TODS, TOIS and VLDBJ. Some of his research results have been included in several well-known database research/text books. Dr. Zhu served as a program/organizing committee member for numerous international conferences. His current research interests include query optimization, data stream processing, multidimensional indexing, self-managing databases, Web information systems, and data mining.

Calisto Zuzarte is a senior technical manager at the IBM Toronto Laboratory. He has been involved in several projects leading and implementing many features related to the DB2 SQL compiler. His main expertise is in the area of query optimization including cost based optimizer technology and automatic query rewriting for performance. Calisto is also a research staff member at the Center for Advanced Studies (CAS).

Wen-Chi Hou is an Associate Professor in the Department of Computer Science at Southern Illinois University, Carbondale, USA. He received his Ph.D. in Computer Science from Case Western Reserve University, Cleveland, USA, in 1989. His research interests include statistical databases, mobile databases, XML databases and data streams.

Trademarks

References

[1] Daniel J. Abadi, Don Carney, Ugur Centintemel, Mitch Cherniak, et al.: Aurora: A new Model and Architecture for Data Stream Management. In *VLDB Journal*, 12(2): 120-139, 2003.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, et al.: STREAM: The Stanford Stream Data Manager. In *IEEE Data Eng. Bull.*, 26(1): 19-26, 2003.

[3] Ahmed M. Ayad and Jeffrey F. Naughton: Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proc. SIGMOD Conf.*, pp. 419-430, 2004.

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

[5] Sirish Chandrasekaran and Michael J. Franklin: Streaming Queries over Streaming Data. In *Proc. Int'l Conf. VLDB*, pp. 203-214, 2002.

[6] C. Cortes, C. Fisher, K. Pregibon, et al.: A Language for Extracting Signatures from Data Streams. In *Proc. ACM SIGKDD Conf.*, pp. 9-17, 2000.

[7] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald: Semantic Approximation of Data Stream Joins. In *IEEE TKDE*, 17(1): 44-59, 2005.

[8] M. J. Franklin and T. Urhan: XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin*, 23(2): 2733, 2000.

[9] Lukasz Golab and M. Tamer Ozsu: Issues in Data Stream Management. In *ACM SIGMOD Record*, 32(2): 5-14, 2003.

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

[11] Rajeev Motwani, Jennifer Widom, Avrind Arasu, et al.: Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. CIDR*, 2003.

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

[13] Adegoke Ojewole, Qiang Zhu, and Wen-Chi Hou: Window Join Approximation over Data Streams with Importance Semantics. In *Proc. ACM CIKM*, pp. 112-121, 2006.

[14] Kostas Patroumpas and Timos Sellis: Window Specification over Data Streams. In *EDBT 2006 Workshops*, pp. 445-464, 2006.

[15] Frederick Reiss and Joseph M. Hellerstein: Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *Proc. ICDE*, pp. 155-156, 2005.