Stream Window Join: Tracking Moving Objects in Sensor-Network Databases[1]

Moustafa A. Hammad Walid G. Aref
Purdue University
West Lafayette, IN 47907, USA
{mhammad,aref}@cs.purdue.edu

Ahmed K. Elmagarmid
Hewlett Packard
Palo Alto, CA, USA
ahmed_elmagarmid@hp.com

Abstract

The widespread use of sensor networks presents revolutionary opportunities for life and environmental science applications. Many of these applications involve continuous queries that require the tracking, monitoring, and correlation of multi-sensor data that represent moving objects. We propose to answer these queries using a multi-way stream window join operator. This form of join over multi-sensor data must cope with the infinite nature of sensor data streams and the delays in network transmission. This paper introduces a class of join algorithms, termed W-join, for joining multiple infinite data streams. W-join addresses the infinite nature of the data streams by joining stream data items that lie within a sliding window and that match a certain join condition. W-join can be used to track the motion of a moving object or detect the propagation of clouds of hazardous material or pollution spills over time in a sensor network environment. We describe two new algorithms for W-join, and address variations and local/global optimizations related to specifying the nature of the window constraints to fulfill the posed queries. The performance of the proposed algorithms are studied experimentally in a prototype stream database system, using synthetic data streams and real time-series data. Tradeoffs of the proposed algorithms and their advantages and disadvantages are highlighted, given variations in the aggregate arrival rates of the input data streams and the desired response times per query.

1. Introduction

The widespread use of sensor networks introduce revolutionary opportunities for different disciplines such as life and environmental sciences and civil engineering. The readings collected from these sensors form a continuous supply of streaming data. Many of these applications such as monitoring, tracking and surveillance applications involve continuous queries over an infinite supply of streaming data from multiple sensors.

An important aspect for stream query processing is the introduction of operators that are non-blocking and that can process infinite amounts of data. Most recently, researchers have expressed interest in window aggregate operations for data streams [9], where the window defines a prefix of the stream. Window join processing is reported in [6, 12, 13] as a practical way to join multiple infinite data streams. In [13], the Telegraph project suggests implementing a multi-way join operator (referred to as SteM), to handle joining multiple data streams. In this approach, the window is defined in terms of tuple count and new tuples in each stream force old tuples from the same stream to expire after a specific count of tuples. Psoup [6] uses the notion of time window over data streams and processes the window constraint similar to the SteM approach. The recent work in [12] addresses the window join over two streams where the two arriving streams have different arrival rates. The authors suggest using asymmetric join (e.g., using a nested loop for one stream and building a hash table on the other stream), to reduce the execution cost. The approach in [12] for window join evaluation is quite similar to that of SteM and Psoup.

The approaches in [6, 12, 13] work efficiently for synchronized data streams (no delays among tuples from different data streams) or for windows defined in terms of tuple counts. For windows defined in terms of time intervals, delays may occur among tuples from different data streams. In this case, the approaches in [6, 12, 13] tend to provide approximate answers for the window join operation. This is the result of old tuples expiring before completely joining with delayed tuples.

Delays are likely to occur when sensors are connected over the network. This motivates the need for algorithms that accommodate both synchronous and asynchronous arrivals of tuples from data streams. This class of sliding time window join, which we refer to as *W-join*, is the focus of

[1]This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, IIS-0209120, and EIA-9983249.

Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM'03)
1099-3371/03 $17.00 © 2003 IEEE

this paper. Two examples of *W-join* are:

Example 1: Spotting the spread area of a pollution cloud using a sensor network. A user issues a query to monitor the propagation and expansion of a pollution cloud in a sensor network. The user specifies a time window for the propagation to occur based on wind speed and the maximum time for the cloud to propagate between two sensors and hence represents a window of interest over the readings between any two sensors.

Example 2: Tracking objects that appear in video data streams from multiple cameras. The objects are identified in each data stream and the maximum time for the object to travel through the monitoring devices defines an implicit time window for the join operation.

1.1 The Forms of Window Join

W-join can be of several forms depending on the existence of window constraints among each pair of the joined streams. We start by describing the different forms of W-join. In the following sections, we develop window join algorithms to handle all the different forms described here. We introduce the SQL construct WINDOW(A,B) to define the time window between tuples in the streams A and B.

Form 1: *A binary window join* joins two input streams using a single window constraint. For example, the SQL query corresponding to Example 1 is (w is the maximum time needed for objects to move from sensor A to sensor B):

```
SELECT A.Gas FROM Sensor1 A, Sensor2 B
WHERE A.GasId = B.GasId
WINDOW(A,B) = w
```

Form 2: *A path window join* joins multiple streams where the window constraints connect the streams along one path. In Example 2 where an object Obj, needs different times $(w_1, w_2, ..)$ to travel from one camera to the other, the user may issue the query:

```
SELECT A.Obj
FROM Camera1 A, Camera2 B, Camera3 C
WHERE similar(A.Obj,B.Obj) AND similar(B.Obj,C.Obj)
WINDOW (A,B)=w1 AND WINDOW (B,C)=w2
```

The *similar()* user-defined function determines when two objects that are captured by different cameras are similar, and is beyond the scope of this paper. For the purposes of this paper, the reader may think of this function as equivalent to an equality predicate on object identifiers, i.e., A.Obj = B.Obj.

Form 3: *A graph window join* joins multiple streams where the window constraints among the streams form a graph structure. For example, the following query tracks the spread of a hazardous gas that is detected by multiple sensors, where the maximum time for the gas to travel through the sensors defines the time windows, (w_1, w_2, w_3, w_4):

```
SELECT A.Gas
FROM Sensor1 A, Sensor2 B, Sensor3 C, Sensor4 D
WHERE A.Gas=B.Gas AND B.Gas= C.Gas AND C.Gas=
D.Gas
WINDOW(A,B)=w1 AND WINDOW(B,C)=w2 AND
WINDOW(B,D)=w3 AND WINDOW(C,D) =w4
```

Notice that the window constraints may not exist among all possible pairs of the streams (e.g., streams A and C may not be constrained by a window due to the existence of a barrier that prevents the gas from spreading directly).

Form 4: *A clique window join*, is a generalization of Form 3, where the window constraints exist between every pair of the joined streams. In the previous example, if there are no barriers between any pair of sensors, then there exists a window constraint between every pair of sensors, which represents the maximum time for the gas to travel between them. As a special case of **Form 4**, we consider the *uniform clique window join*, where all the streams are joined together using a single window constraint (same as Form 4 except that the windows between each pair of the streams are equal). For example, to monitor the sales from multiple department stores using a sliding time window, say w, an administrator may issue the query:

```
SELECT A.ItemName
FROM Store1 A, Store2 B, Store3 C, Store4 D
WHERE A.ItemNum=B.ItemNum AND
B.ItemNum=C.ItemNum AND C.ItemNum=D.ItemNum
WINDOW = w
```

Notice that "window = w" applies to all stream pairs.

1.2 Contributions of the Paper

The paper introduces two new algorithms; the backward and forward evaluations of W-join, termed BEW-join and FEW-join, respectively, for evaluating time window constraints for all the alternative forms of W-join, described in the previous section. BEW-join and FEW-join are different in the way they evaluate the window constraints and update their underlying streams. We identify ranges of arrival rates where each algorithm provides improved performance in terms of either response time or throughput.

We present the two join algorithms in the context of the *uniform clique window join*, presented in the previous section. In addition, we adapt algorithms FEW-join and BEW-join to handle the other forms of W-join, where some window constraints may not be present in the query. For that, we propose two approaches; the *global and local conservative* approaches (GCA and LCA, respectively) that utilize a filter-refine paradigm. GCA chooses the *maximum* window constraint to reduce the join into a uniform clique W-join and applies any of the W-join algorithms. In contrast, LCA optimizes FEW-join to consider the maximum window constraint per sensor.

We present an extensive performance study of all algorithms implemented in a prototype stream database system while using synthetic data streams and real time-series data.

2 Preliminaries

2.1 Stream Model

We model each stream data item as two components $< \mathrm{v}, \mathrm{t} >$, where *v* is a value (or set of values) of the data item, and *t* is the timestamp that defines the order of the stream sequence. The value of the data item can be a single value or a vector of values. *Time* is our default ordering domain. The *timestamp* is considered the sequence number, which is implicitly attached to each new data item. This notion of time may refer to either the *valid time* or the *transaction time* [16], where valid time is the time assigned to the item at its source stream (when it is first created), and transaction time is the time assigned to the data item at the query processing system. Synchronizing the arrival of tuples that belong to different data sources is not easy to enforce especially if the sources are independent and each uses its own clock to generate the timestamps (valid time). Moreover, if the query processor adds timestamps to the arriving tuples (transaction time), because of tuple scheduling in stream data systems, delays among tuples from data streams may occur during the query execution. The algorithms that we propose in this paper handle stream data with either type of timestamps.

2.2 The W-join Operation

To illustrate the operation of a W-join, Figure 1(a) shows a W-join among five data streams (A – E). The position of the tuples on the x-axis represents the arrival order over time. The indices of each tuple represent the order of arrival of this tuple relative to its data stream. The black dots correspond to tuples from each stream that satisfy the WHERE clause. The window constraint implies that only tuples within a window of each other can join. Thus, the tuple $< a_2, b_2, c_1, d_2, e_2 >$ is a candidate for the W-join, however the tuple $< a_2, b_2, c_1, d_2, e_1 >$ is not, since e_1 and d_2 are more than window, w, away from each other.

The W-join in Figure 1(a) represents *a uniform clique W-join*. A graph model of W-join can be obtained by representing tuples from the streams as nodes in a graph, where edges correspond to the window constraint (e.g., tuples from stream A and B must be within window of time from each other). With this model, the uniform clique W-join represents a complete graph, Figure 1(a). The non-uniform clique, graph, and path W-joins are shown in Figure 1(b), respectively.

It is evident from the W-join operation that we need an efficient approach for continuously verifying window constraints among the input streams and for updating the *join buffers* (intermediate structures that hold tuples from each stream during the join) to contain only eligible tuples. A

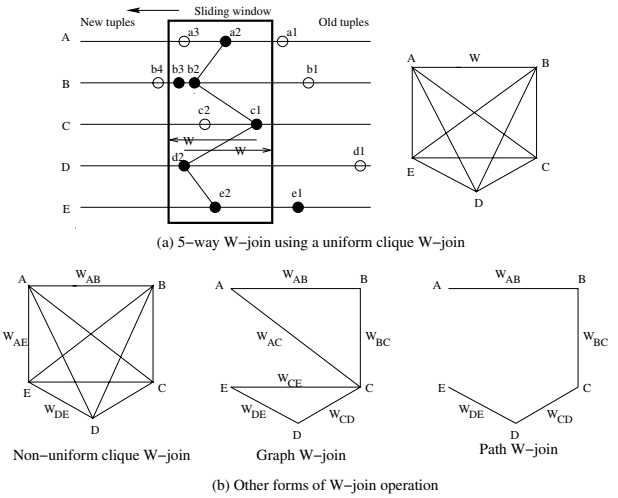


Figure 1. Variations of W-join.

brute-force approach to verify window constraints among streams requires verifying the constraint between each pair of n streams, adding $\binom{n}{2}$ additional comparisons for each input tuple. A more efficient approach is suggested in [2] to verify that n objects, each from a different class, are within a fixed-radius from each other. A similar approach is also introduced for band join in [8].

The BEW-join algorithm adopts a similar approach to [2, 8] to verify the window constraints among the individual tuples. However, the algorithms in [2, 8] cannot deal with infinite streams and may block if data is delayed from one stream. In contrast, the BEW-join algorithm is non-blocking and provides online update of join buffers to reflect tuples that reside within the window from each other.

3 The Uniform Clique W-join

In this section, we introduce the W-join algorithms in the context of the uniform clique W-join. In the following sections, we demonstrate how we address the other W-join query forms.

3.1 The Backward Evaluation W-join Algorithm

To identify tuples that are within window w from each other, the BEW-join algorithm maintains a period, say P, that includes the timestamp of the new arriving tuple t_{new} and timestamps of the other $n-1$ tuples (each from a different stream). P starts with length equals twice the size of the window constraint, w, and has the timestamp of t_{new} as its center. The period is completely constructed when it includes a tuple from every stream (including t_{new}) such that the difference between the maximum timestamp and minimum timestamp of the tuples in P is less than or equal to w. All the tuples that currently belong to P must satisfy the

window constraint and can be checked for satisfaction of the join predicate. While constructing P, a tuple, say t_{old}, from a stream, say S, can be safely removed from the join buffer of S if and only if t_{old} has a timestamp that is older than the newest tuples in all streams, other than S, by more than w. Tuple t_{old} is guaranteed not to W-join with any new tuples from the other streams and need not be stored. The process of updating P and updating the join buffer is repeated for t_{new} and with all combination of tuples in the join buffers of the other streams. This step is terminated as all combinations are exhausted and in this case the algorithm can process a different new tuple, not necessarily from the same stream, and construct a new period in a similar fashion. We use the term *iteration* to refer to the process of W-joining a new tuple and until the algorithm is ready to process another new tuple. The algorithm has no notion of *fixed outer* stream, where it always starts the join process, and therefore does not block waiting for tuples from a single stream.

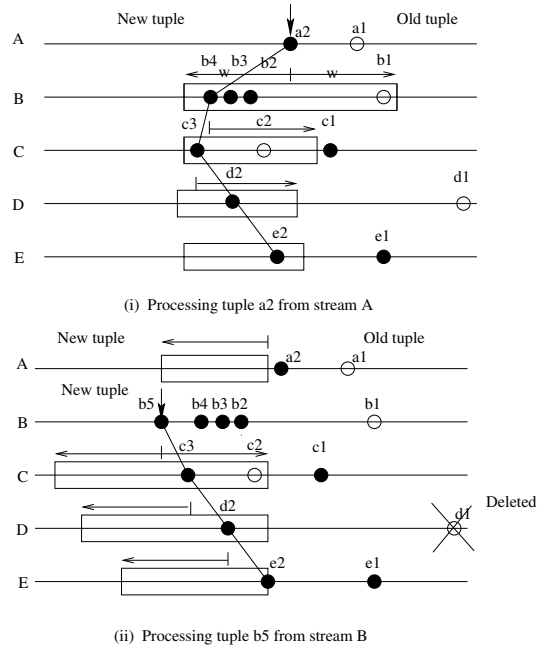


Figure 2. The BEW-join algorithm.

Figure 2(i) depicts the status of the algorithm when processing tuple a_2 from Stream A and forming a window of length $2w$ centered at a_2. The algorithm iterates over all tuples of Stream B which are within the window of tuple a_2. These tuples are shown inside the rectangle over B. b_4 satisfies the join predicate (for illustration of the algorithm, we assume that only the *black* dots (tuples) from each stream satisfy the join predicate[2]) and is located within the window of a_2 (i.e., it is included in the rectangle). The period is modified (shrunk) to include a_2, b_4 and all tuples within w of both of them. This new period is used to test tuples in Stream C, and is shown as a rectangle over Stream C in Figure 2(i). The same process as with Stream B is repeated for Streams C, D and E, respectively, and finally the 5_tuple $< a_2, b_4, c_3, d_2, e_2 >$ is reported as output. The algorithm recursively backtracks to consider other tuples in Streams D, then C and finally B. The final output 5_tuples in the iteration that starts with tuple a_2 are: $< a_2, b_4, c_3, d_2, e_2 >$, $< a_2, b_3, c_3, d_2, e_2 >$, $< a_2, b_3, c_1, d_2, e_2 >$, $< a_2, b_2, c_3, d_2, e_2 >$ and $< a_2, b_2, c_1, d_2, e_2 >$, respectively. While iterating over stream D, tuple d_1 is located at distance more than w from all the last tuples in streams A, B, C, E, (i.e., tuples a_2, b_4, c_3, e_2). Tuple d_1 can be safely dropped from the join buffer of stream D. After finishing with tuple a_2, the algorithm starts a new iteration using a different new tuple(if any). In the example of Figure 2, we advance the pointer of Stream B to process tuple b_5. This iteration is shown in Figure 2(ii) where periods over Streams C, D, E and A are constructed, respectively.

The algorithm never produces spurious duplicate tuples, since in each iteration a new tuple is considered for the join (the newest tuple from a stream). The output tuples of this iteration must include the new tuple, thus duplicate tuples cannot be produced.

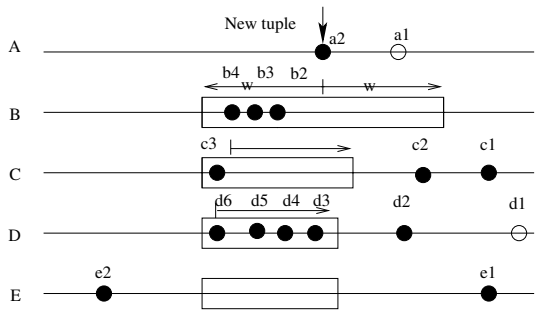
The nested loop implementation of the BEW-join algorithm is straightforward from the previous description. The join buffers are FIFO (First In First Out) queues over each data stream and the algorithm keeps an *ordered* vector, V, of timestamps for the newest tuples in each data stream. Each new tuple, say t_{new}, is added to the queue corresponding to its data stream and updates V. Then t_{new} starts to iterate over and update the other steam queues as described above. The hash implementation of the algorithm is quite similar to the nested loop implementation except that the FIFO queues are hash tables and the scanning is replaced with probing the corresponding bucket in the other hash tables. Bucket updating is carried similarly to nested loop using vector V.

3.1.1 Discussion

One clear advantage of the previous algorithm is that it does not block waiting for tuples from a single data stream. The BEW-join algorithm processes any stream that has tuples waiting for the join (no fixed outer data stream). In addition, the update of the data streams (dropping of old tuples) is performed during the join. Therefore, the join buffers do not increase indefinitely during the join.

However, the BEW-join may iterate over tuples that will never W-join as we illustrate by the following example. Figure 3 illustrates this case where data streams are not of equal arrival rates and one of the data streams is very slow compared to the others. In this example, the BEW-join

[2] Although the join predicate is equi-join, this algorithm can work with any types of join predicate such as non-equi-join or user defi ned functions.



(i) Processing tuple a2 from stream A

Figure 3. BEW-join extra processing.

algorithm will examine 12 tuples (1x3x1x4 from streams A, B, C and D, respectively), none of which will contribute to the final W-join results. This example illustrates that the BEW-join may fall into cases that include lengthy and nested iteration over tuples that may not satisfy the window constraint. In addition, during the iterative execution of BEW-join, the algorithm verifies for each tuple the period inclusion and update the period accordingly. This check and update of the period adds to the time complexity of processing each tuple in the nested iteration (many of them are examined in each iteration).

3.2 Forward Evaluation of the W-join (FEW-join) Algorithm

As presented in the previous section, the repeated verification of the period inclusion and update of the period increases the execution time per input tuple. To eliminate the repeated window test we need to constrain the join buffers (that store the current tuples from different data streams) to hold only tuples that are within window of each other. In this case, a blind (without window verification) iteration can be performed to verify the join predicate. The FEW-join algorithm achieves this by maintaining the join buffers in a way that always reflects tuples, across the different data streams, that are within window w of each other and restricts the window constraint test to occur only once when a new tuple arrives.

3.2.1 Algorithm Description

The basic idea of the FEW-join algorithm is to use the current tuples in the join buffers to determine a combination of tuples, *starting n_tuples* that bounds the past (the tuples that have already arrived in the rest of the data streams and needs to be checked by a new tuple). The algorithm also determines a forward point in time, F_t, before which all arriving tuples can join without a need to verify the window constraint. The join buffers maintain tuples from each data stream that are time-bounded at one end by a tuple from the starting n_tuples and at the other end by F_t. We describe the details of the FEW-join algorithm as follows.

The FEW-join algorithm iterates between two modes, *Mode 1* and *Mode 2*. In *Mode 1*, a plane sweep is performed on the time axis for all incoming tuples from all the streams. The target is to locate the first n_tuples at window distance w from each other, where n refers to the number of joined streams. As this tuple is found, Mode 1 constructs a period, P, that includes all the n_tuples and determines $F_t = max\_time\_stamp(P)$. In Mode 1, all the tuples older than the starting n_tuples are dropped from the streams' buffers. Then Mode 2 begins admitting tuples that follow the starting n_tuples and that have timestamps within the period P. Those tuples are guaranteed to be within w time units from the tuples already included in the period. As a result, no repeated verification for the window inclusion is necessary while joining the new tuple to the ones already in the period. Mode 2 continues processing new tuples until they fall outside the period. In this case, Mode 1 is restarted to determine new starting n_tuples and new period.

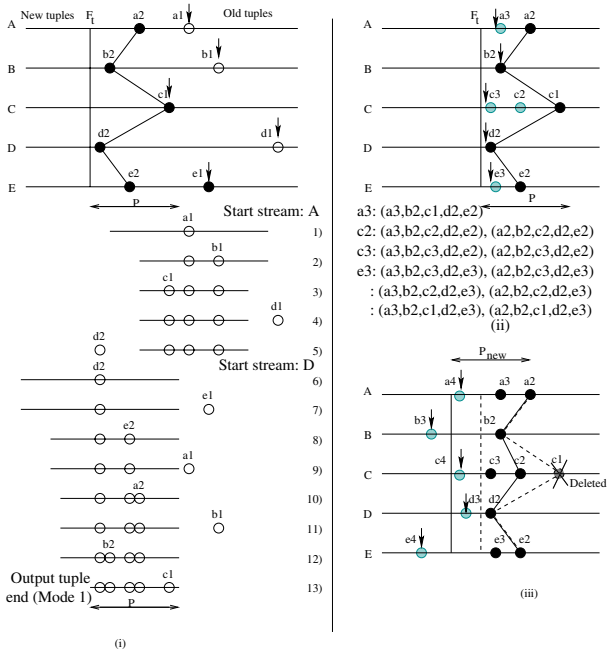


Figure 4. The FEW-join Algorithm.

Figure 4 shows an example of joining five Streams, A, B, C, D, and E. Assume that a_1, b_1, c_1, d_1, e_1 are the first tuples in each stream. Mode 1 starts with Stream A and determines the period that includes a_1, shown in Step 1 of Figure 4(i). The algorithm proceeds to Stream B, and b_1 is tested to determine if it falls within the period determined by a_1. Since b_1 is included inside the period, the period is modified (shrunk) to include a_1, b_1 and all tuples within window of each of them, the period adjustment is

shown in Step 2. Stream C is processed similar to Stream B (Step 3). In Stream D, since d_1 is older than the current period, d_1 is dropped (deleted from the join buffer) and the next tuple, d_2, is considered for Mode 1. Since d_2 is older than the current period, the current period is discarded and a new period is created in Step 6 which is centered at d_2. Mode 1 then considers Stream D as the starting stream, and continues testing the rest of the streams before looping back to Stream A. Stream E is now processed. e_1 is older than the period, so it is dropped and the next tuple from Stream E is tested. e_2 is included within the period, which is updated in Step 8 to reflect the inclusion. Streams A and B are processed similar to Stream E (Steps 9-12). At Step 13, tuple c_1 in Stream C is added to the period and all the streams participate by a single tuple in the final period, P. This indicates the end of Mode 1. The final tuple becomes the starting n_tuples for Mode 2. Note that the window join predicate is not considered during Mode 1, and only the period inclusion is verified.

Mode 2 begins iterating over all tuples in the neighborhood of the n_tuples and included within P. In Mode 2 we start by verifying the join predicate for the starting n_tuples only if it includes new tuples from the streams. Afterwards, Mode 2 processes the new tuple from each stream repeatedly as long as it falls within P.

Figure 4(ii) illustrates the layout of the data streams as the new tuples a_3, c_2, c_3 and e_3 arrive, respectively. The output tuples are illustrated below Figure 4(ii) at the time each new tuple arrives. Note than for each of these tuples (and generally during Mode 2) there is no need to verify the window constraint as it is guaranteed that tuples already in the join buffer are included within the period (i.e., within window from each other). The algorithm stops Mode 2 when the next tuple from each stream is newer than the current period P. In Figure 4(iii) as tuples a_4, b_3, c_4, d_3 and e_4 arrive, Mode 2 drops the tuple c_1 (the oldest tuple in the period P) and restarts Mode 1 again to search for next starting n_tuples and so on.

3.2.2 Discussion

Mode 1 of the FEW-join algorithm updates the buffers (i.e., by adding new tuples and dropping tuples from the join buffers) of each stream such that the join buffers only include tuples that lie within window of each other. Therefore, the algorithm *avoids nested-iteration over any tuples outside the window*. In addition *Mode 2 needs to test the window inclusion only when admitting a new tuple* and not during the nested iteration over tuples from the different streams.

The advantages of the FEW-join algorithm comes at the expense of increasing the waiting time for some input tuples. For example, in Mode 2, switching to Mode 1 (to determine a new period) is delayed until all new tuples from the different data streams are also newer than the current period. The waiting times for tuples increase as they need to wait for tuples from other data streams before processing. This is not the case for BEW-join where a new tuple is processed immediately as long as BEW-join can handle it.

3.2.3 Hash Implementation of FEW-join

The FEW-join is described as a nested loop join. In this section we describe the implementation of the algorithm when the tuples from each stream are stored in a hash table. The only requirement when using the FEW-join is that the tuples in the hash table should be organized in the order of their arrival. In this case, Mode 1 can perform a linear scan over tuples in each stream and update the hash table as a whole. For this requirement we use a structure of the hash table that maintains the order of tuples at the level of a bucket as well as the level of the whole hash table by two doubly linked lists (DLL). Mode 1 uses the DLL that links tuples of the whole hash table to update the hash table by dropping old tuples. New tuples are inserted in the bucket that has a similar hashing value and the links of the two DLLs (the DLL of the bucket and the one of the hash table) are updated accordingly. Mode 2 only iterates over the bucket probed by the new tuple.

4 Performance Study

4.1 Experiments Setup

The implementation is performed on a prototype stream database system, STEAM, based on PREDATOR [14], which is modified to accommodate stream processing. The modifications include introducing an abstract data type *stream-type* that can represent source data types with streaming capability. Any stream-type must provide the following interfaces, *InitStream, ReadStream, and CloseStream*. In order to collect data from the streams and supply them to the query execution engine, we developed a *stream manager* as a new component of the stream database system. The main functionality of the stream manager is to register new stream-access requests, retrieve data from the registered streams into its local buffers, and supply data to be processed by the query execution engine. The stream manager runs as a separate thread and schedules the retrieval of tuples in a round robin fashion (other scheduling options are still being studied). To interface the query execution plan to the stream manager, we introduce a *StreamScan operator* to communicate with the stream manager and receive new tuples as they are collected by the stream manager.

We performed our experiments on both real time-series data and synthetic data streams. For real data, we use the logs of the transactions from Wal*Mart stores[3]. A sold item in a store appears as a transaction of customer purchases. A single transaction for each store includes the item number in addition to other information, such as the item description, the item unit price, the item quantity and the item purchase date and time. The timestamp of each tuple is the time of the transaction, *valid time*. Data is extracted from an NCR Teradata machine that holds 70GBytes of Wal*Mart data. We consider also synthetic data streams, where each stream consists of a sequence of integers, and the inter-arrival time between two numbers follows the exponential distribution with mean λ.

All the experiments are run on a Sun Enterprise 450 machine, running the Solaris 2.6 operating system with 4 GBytes main memory.

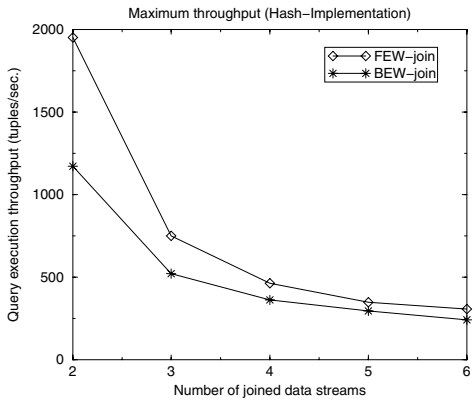
4.2 Experimental Results

4.2.1 Query Execution Throughput

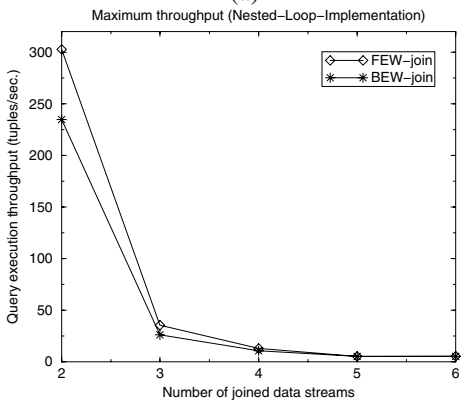
In the following set of experiments we use the *throughput* as the performance measure for the query execution, i.e., the number of tuples processed by each algorithm in a unit time. The throughput indicates how fast each algorithm executes. All measures are collected at the steady state. To compute the throughput, we measure the total execution time every 1000 tuples and calculate the average. We repeat this calculation for an increasing number of input tuples until this value is almost unchanged. We repeat this experiment while varying the arrival rate of input data streams and measure the throughput in each case. We monitor the throughput until the point that the W-join algorithm is no more capable of handling the total arrival rate over all its input data streams. After this point the W-join algorithms start to drop tuples from the input data streams. The experiment is repeated for $2-6$-way W-joins and using the hash and nested loop implementations for the BEW-join and the FEW-join algorithms. For brevity, we present only the results as the input rates of all data streams are comparable (no significant differences among the input rates from one data stream and the other). We have experimented the case when the arrival rate between data streams is different and the obtained results are similar in spirit to that when the arrival rates are comparable.

We use synthetic data streams with integer values that range between 1 and 1000. For nested loop implementations we use queries with window of size 10 seconds. For hash implementations we use queries with window of size of 1 minute.

Figure 5 shows the maximum throughput that can be supported by each algorithm as we increase the input arrival rates and for $\{2-6\}$-way W-joins. The FEW-join algorithm achieves higher maximum throughput than BEW-join, which falls earlier behind the input arrival rates. This is the case for all the experimented $\{2\text{-}6\}$-way W-joins. The improvement is approximately 70% for 2-way W-join using hash implementations (Figure 5(a)), and 40% for 2-way using nested loop implementations (Figure 5(b)). In both implementations, the difference between the maximum throughput in BEW-join and FEW-join is reduced as we increase the number of joined data streams. This is the case as both algorithms spend most of the execution time iterating over large number of tuples (joining more data streams increases exponentially the number of tuples to test). Besides, for FEW-join algorithm as we increase the arrival rates the algorithm switches between Mode 1 and Mode 2 almost for every new tuple. The switch back and forth between Mode 1 and Mode 2 increases the overhead of Mode 1 in total execution time and reduces the save in comparisons introduced by Mode 2.



(a)



(b)

Figure 5. Maximum execution throughput.

4.2.2 Response Time

In this section we study the average response time of a new tuple to complete execution. This time includes the waiting

[3]Donated to Purdue University by Wal*Mart and NCR Corporation

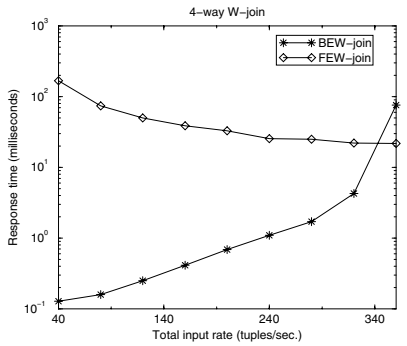


Figure 6. Response time using the hash implementations

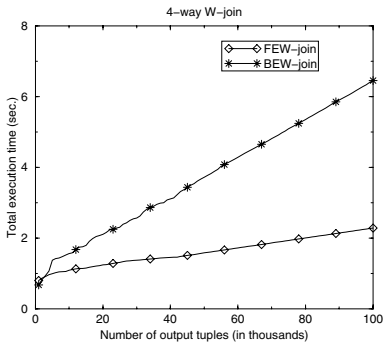


Figure 7. Total execution time for output tuples.

time plus the execution time for a tuple. We use hash implementations of the BEW-join and FEW-join algorithms. In the first experiment we use 4-way W-join with window of size one minute and use synthetic data streams. We change the average arrival rate per each stream from 10 to 90 tuples/sec. We measure the time elapsed from the arrival of each new tuple until its completion, averaged over a range of 10,000 input tuples.

As illustrated in Figure 6, for low arrival rates the BEW-join algorithm has low response time when compared to the FEW-join algorithm. This is mainly due to the behavior of Mode 1 and Mode 2 of the FEW-join algorithm. Both modes need to hold new input tuples before finding the starting n_tuples for Mode 1 and before switching to Mode 1 from Mode 2. This also increases the average waiting time of new tuples in the FEW-join algorithm. For the BEW-join algorithm, the processing of each new tuple is independent of arriving tuples in other data streams. As a result, the waiting time is minimal and is determined by the current waiting tuples before the W-join operation. As the data arrival rate increases, the BEW-join algorithm spends more time processing a tuple and therefore the response time is increased. For FEW-join, the execution time is smaller than that of the BEW-join algorithm and as we increase the arrival rate, Mode 1 and Mode 2 need not wait for tuples for long periods of time. As a result the response time for the FEW-join algorithm decreases as we increase the arrival rate. The maximum throughput of BEW-join is approximately 340 tuples/sec (aggregate input arrival rate). The response time of BEW-join crosses that of FEW-join algorithm and gets higher afterwards.

4.2.3 Output Execution Time for Real Time-series Data

In this experiment we study the performance of the proposed algorithms while using real time-series data from Wal*Mart stores. We measure the total output execution time of each algorithm for various (increasing) sizes of stream databases. We use hash based implementation of both algorithms and run W-join queries similar to Example 1 in Section 1. We use window of size one hour and conduct the experiment of $4, 5$ and 6-way W-join. We illustrate only the results for 4-way W-join. Figure 7 shows that the FEW-join algorithm can produce more output in less amount of time than the BEW-join algorithm. For example, FEW-join can produce 100,000 output tuples in 2.3 seconds for 4-way W-join, whereas the BEW-join algorithm produces the same amount of output tuples in 6.5 seconds.

5 Variations of W-join

In this section, we study the forms of W-join where the window is not unique between all the streams and/or some pairs of the streams are not window-constrained (Form 4 and Form 3 described in Section 1.1). The path W-join, Form 2, is presented as a special case of Form 3, the graph W-join.

5.1 The Non-uniform Clique W-join

The BEW-join and FEW-join algorithms as described in Section 3 require a single window constraint to be applied over all streams. Having different window constraints between every pair of the streams requires adaptation of these algorithms. One *conservative* approach for solving W-join with different window constraints is to consider the *largest* window constraint as the single window constraint among all streams, and apply the BEW-join or the FEW-join algorithms to find all candidate tuples. This step is referred to as a *filtering* step. Since the filtering step will result in *false positive* answers (tuples that should not be reported in the actual W-join), we use a *refinement* step where all the windows' constraints are verified between the tuples included in the output n_tuples. This final test will only be applied to the output of the filtering step. We refer to this approach as a *global conservative approach, GCA for short*.

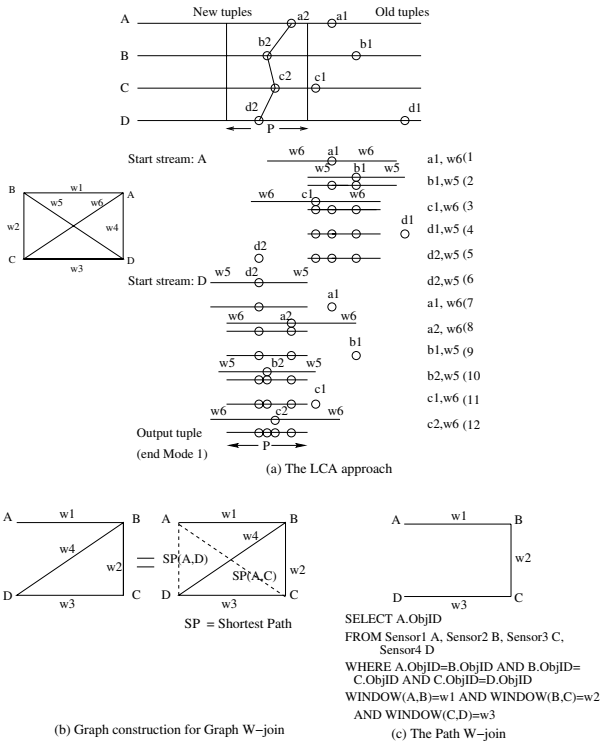


Figure 8. Variations of W-join.

GCA may assign an unnecessarily large window size during the join, resulting in an excessive number of additional comparisons. We propose a variant of the FEW-join algorithm that uses the maximum local windows for each stream. We call this approach *the local conservative approach, LCA for short*.

LCA has the same filter and refine steps as those of GCA. The filter step also alternates between two modes, Mode 1 and Mode 2, similar to the of FEW-join. Mode 1 produces the starting n_tuples and the period, and Mode 2 iterates on all tuples in this period to produce the results. However, during Mode 1, the construction of the period considers the maximum *local* window for each stream. We describe the filtering step using the example shown in Figure 8(a). The example illustrates a W-join among four streams. We assume the window sizes are ordered by their indices (i.e., w_6 is the maximum window size). The execution steps of LCA are similar to those of FEW-join.

5.2 The Graph W-join and the Path W-join

In both graph and path W-joins, one or more pairs of the streams are not window-constrained[4]. For these types of W-join, we propose two approaches. The first approach is similar to the conservative approaches developed in Section 5.1. The second approach uses the BEW-join approach to update tuples (add and drop tuples) in data streams. In the following sections we describe the two approaches.

[4]We assume a connected graph with regard to window constraints among the streams.

5.2.1 Using Conservative Approaches

With the absence of some window constraints, the implicit assumption would be that the two streams join in *infinite* window sizes. However, due to the nature of the window constraints, we can deduce more practical values for the missing constraints. Figure 8(b) shows a 4-way W-join, where the window constraints between Streams (A, C) and (A, D) are not specified. The upper bound to the missing constraint is *the shortest path* between these two streams. For example, in Figure 8(b), the window constraint between Streams (A, D) is MIN($w_1 + w_2 + w_3$, $w_1 + w_4$). The calculation of the shortest path is only performed at query compilation time and no query execution time is affected.

We repeat the construction of the bound evaluation for all missing window constraints until arriving at a complete graph. At this point, either one of the conservative approaches can be used, as described in Section 5.1.

5.2.2 Using a Variation of BEW-join

The BEW-join algorithm drops a tuple, say t, from one stream if and only if t is more than a time-window older than all new tuples that arrive at the other joined data streams. This mechanism can be extended to handle graph and path W-joins in the following way. For each data stream, say S, we select a subset of the window constraints that S participates in. We refer to this subset as C_S. Tuples that satisfy all the window constraints in C_S are reported as output after being evaluated and filtered by the refinement step. Tuples that violate all the window constraints in C_S are dropped from S during W-join. On the other hand, tuples that only violate some of the window constraints are kept (as these tuples may satisfy the window constraints with a different combination of tuples from the other streams). For example, in Figure 8(b), the window constraints for stream B are as follows (assume that $TS(t)$ represents the timestamp for tuple t):

$$\begin{aligned} C_B \quad = \{ \quad & (TS(t_A) - TS(t_B)) \leq w_1, \\ & (TS(t_C) - TS(t_B)) \leq w_2, \\ & (TS(t_D) - TS(t_B)) \leq w_4 \quad \} \end{aligned}$$

A clear advantage of the modified BEW-join approach over the conservative approaches is that the window size per stream could be smaller. The advantage is clear when one of the time windows is much larger than all the other time windows. In this case, for path and graph W-joins, BEW-join iterates over less tuples and processes a new tuple faster than any of the conservative approaches.

6 Related Work

Seshadri et al. [15] provide the SEQ model and implementation for sequence databases. A sequence is defined as a set with a mapping function to a defined ordered domain. The work in [11] provides a data model for chronicles (sequences) of data items and discusses the complexity of executing a view described by the relational algebra operators. In [10], the work includes a study of algorithm complexity on computation over data stream. Adaptive query processing [3] and execution of continuous query [7] address reordering of operators during execution and how long running queries are managed. The COUGAR [5] system and the work in [17] focus on executing queries over sensor data and stored data. Sensors are represented as new data types with special functions to extract the sensor data when requested. The STREAM [4] project discusses the new demands imposed by data streams on data management and processing techniques. The work on STREAM also addresses the processing of query operators using bounded memory space [1], suggesting that some queries over data streams (e.g., projection with duplicate elimination operators) may be answered using limited memory by considering the relationship between the terms in the WHERE clause.

7 Conclusion

In this paper we identify different variations of sliding time window joins, referred to as *W-join*. We describe two W-join algorithms, BEW-join and the FEW-join. BEW-join achieves low response time for lower aggregate rates of data streams. The FEW-join algorithm outperforms the BEW-join algorithm as the arrival rate of data streams increases. The FEW-join algorithm processes tuples faster than the BEW-join algorithm and can support high arrival rates from the input data streams before starting to thrash. We compared both algorithms using nested loop and hash implementations using real implementation on a prototype stream database system. The performance study validates our previous conclusion. We proposed alternative ways to solve the other variations of W-join using adaptations of our algorithms and utilizing the filter-refine paradigm and compared their performance. We plan to study similar techniques for w-join when the data arrival rates are higher than what the system can handle, and hence will have no choice but drop some tuples, hopefully the insignificant ones.

Acknowledgment

We would like to thank Michael Franklin for his valuable comments.

References

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. Of PODS 2002, June.*, 2002.

[2] W. G. Aref, D. Barbará, S. Johnson, and S. Mehrotra. Effi - cient processing of proximity queries for large databases. In *Proc. of the 11th ICDE, March*, 1995.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the SIGMOD Conference*, 2000.

[4] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record Vol 30 No 3 Sept.*, 2001.

[5] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. of the 2nd Int. Conference on Mobile Data Management, Jan.*, 2001.

[6] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. of the VLDB Conference, Aug.*, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagracq: A scalable continuous query system for internet databases. In *Proc. of the SIGMOD Conference*, 2000.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *17th VLDB Conference, Sept.*, 1991.

[9] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of SIGMOD Conference*, 2001.

[10] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *Technical Note 1998-011, Digital Systems Research*.

[11] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proc. of PODS, May*, 1995.

[12] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE, Feb.*, 2003.

[13] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the SIGMOD Conference, June.*, 2002.

[14] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

[15] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. of 22th VLDB Conference, Sept.*, 1996.

[16] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann, 2000.

[17] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proc. of the 2003 Conference on Innovative Data Systems Research (CIDR), to appear Jan.*, 2003.