Department of Computer Science Technical Reports

Department of Computer Science

2002

Joining Multiple Data Streams with Window Constraints

Moustafa A. Hammad

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
02-010

JOINING MULTIPLE DATA STREAMS WITH WINDOW CONSTRAINTS

Moustafa A. Hammad
Walid G. Aref
Ahmed K. Elmagarmid

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #02-010
May 2002

Joining Multiple Data Streams with Window Constraints

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

This paper introduces a class of join algorithms, termed stream window join (W-join for short), 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. Many practical queries can be answered efficiently with W-join, especially in multi-sensor networks. We describe new algorithms for W-join, and address variations and local/global optimizations related to specifying the nature of the window constraints. In contrast to existing stream join algorithms that store the entire stream prefixes in intermediate structures, we avoid repeated iterations over non-window-related tuples. We present a variety of non-blocking and pipelined algorithms for performing W-join and its variations. These algorithms include nested-loop, hash, and merge-join implementations of W-join (NLW-join, HW-join, and MW-join, respectively). The algorithms utilize a filter-refine paradigm to handle the variations in window constraints and filter out false-positive answers. For comparison purposes, we adapt existing stream join algorithms in the literature to handle the W-join operation. We experiment with the new and existing algorithms in a prototype stream database system, developed at Purdue, using both real and synthetic data streams. We demonstrate that the newly proposed W-join algorithms outperform the existing algorithms by an order of magnitude under a variety of stream data rates and stream delays.

1 Introduction

The widespread use of devices for capturing digital data streams and the importance of the information that can be extracted from them have led to increased research addressing query processing techniques for stream data. The data streams may represent continuously incoming data for a variety of applications, e.g., multi-sensor networks, telecommunication management, network traffic analysis, video security monitoring, surveillance applications and financial data analysis. 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 [8], where the window defines a prefix of the stream. Other studies such as [14, 19] consider a join operation between a data stream and a typical database relation or a self-join over a single stream. However, none of these studies address the join operation among multiple data streams. The recent stream join techniques such as ripple-join [10], the non-blocking hash-joins [7, 20, 21] emphasize on producing early results during join execution. However, they all require processing the entire contents of the stream seen thus far (the whole stream prefix), as well as all incoming tuples. In this paper, we address joining *multiple* data streams over a sliding *window* of time. We refer to the window join operation by W-join. The W-join operation is needed to answer many practical queries that are important for financially and societally relevant applications such as the following:

Example 1: To *monitor* sales from different department stores, a marketing administrator wants to find *common* items sold by all stores over sliding one hour intervals. Current transactions from each store represent streams of data containing information about items as they are sold.

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

Example 3: To detect packets that pass through multiple networks, a network administrator joins traffic information from these networks where the maximum time for a packet to travel between any two networks defines the time window for the join between them.

Such queries can execute separately or are combined with aggregate functions to produce summary information. While these queries carry some similarity with temporal queries, efficient algorithms for temporal-joins [13, 22] depend on the underlying access structure, which is not available for online stream data sources. Also, the temporal join algorithms do not consider answering long running queries over infinite data streams.

In this paper we address the issue of joining multiple data streams over a sliding window of time and that match a certain join condition. We refer to this join operation as *W-join*. W-join can be of several forms depending on the variation and/or the existence of the window constraint between each pair of the joined streams. We start by describing the different forms of W-join along with an example.

Form 1: *A single window constraint* is used to join all input streams. In Example 1, the monitoring of the sales from multiple department stores using a sliding time window, w, the administrator may issue the query:

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

Form 2: *Different window constraints* are considered between all pairs of the input streams. In Example 2, the tracking of objects by multiple video cameras where an object needs different times $(w_1, w_2, ..)$ to travel from one camera to the other, the user may issue the query:

```
SELECT A.ObjID
FROM Camera1 A, Camera2 B, Camera3 C
WINDOW (A,B)=w1 AND WINDOW (B,C)=w2 AND WINDOW (A,C)=w3
WHERE A.ObjID=B.ObjID AND B.ObjID=C.ObjID
```

Form 3: *Some window constraints do not exist* between pairs of input streams. In Example 3, the network packet detection, there may be a direct network link between the networks (A, B) and (B, C), however no direct link between the networks (A, C). An example query is:

```
SELECT S1.PacktID
FROM Network1 A, Network2 B, Network3 C
WINDOW (A,B)=w1 AND WINDOW (B,C)=w2
WHERE A.PacktID=B.PacktID AND B.PacktID=C.PacktID
```

In all these queries, WINDOW(A,B) defines the time window between tuples in A, B.

We introduce a class of W-join algorithms that can be used to answer the different W-join forms. For Form 1, a single window constraint, we present nested-loop, hash and merge W-join algorithms (NLW-join, HW-join and MW-join, respectively). All these algorithms are non-blocking to adapt for the variation in stream data rates. Both algorithms can be easily integrated in a query pipeline execution. The MW-join algorithm uses a different approach to seek the prefixes of all streams that always W-join with each other. The MW-join never iterates over tuples that are not window-related and its intermediate structures (join buffers) always reflect the prefixes of the streams that actually satisfy the window constraint.

In addressing Form 2 of W-join, different window constraints exist between all pairs of the streams. We propose *global and local conservative* approaches that utilize a filter-refine paradigm. The global conservative approach (GCA) chooses the *maximum* window constraint to reduce the join into a single window constraint and applies any of the NLW-join, HW-join or MW-join algorithms. In the local conservative approach (LCA), we adapt the MW-join algorithm to consider the maximum window constraint per each stream. For Form 3 of W-join, when some window constraints are absent, we show how to pre-process the query in order to utilize the conservative approaches. Finally, we address a special case of the W-join, termed the *Path W-join* (e.g., tracking of objects along a certain path), and present the solution based on the conservative approaches, NLW-join and HW-join algorithms.

We present an extensive performance study of all algorithms using a prototype stream database system, developed at Purdue, and using both real and synthetic data streams. In the experiments, we compare with new stream join techniques such as the ripple-join [10] and XJoin [20], which are adapted to answer the W-join operation. The results illustrate significant improvement when using the proposed window join algorithms in answering W-join queries for multiple streams. We also test the algorithms when data streams are of variable rates and illustrate that the MW-join has a comparable performance to the HW-join, which is limited to equi-join cases.

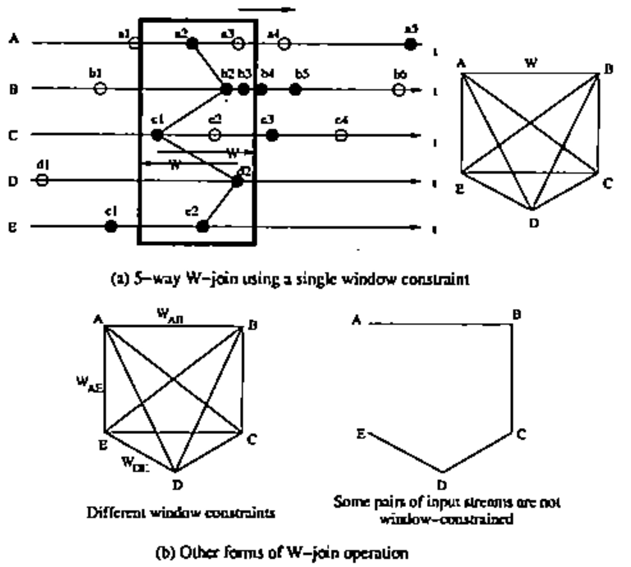
The rest of the paper is organized as follows. Section 2 introduces background information. Section 3 describes the NLW-join, HW-join and MW-join algorithms. In Section 4 we introduce the performance study. Section 5 describes the conservative approaches for the other forms of W-join. Section 6 discusses related work and we conclude in Section 7 with a summary.

2 Preliminaries

In this section we introduce the stream model and our abstract presentation of the W-join that we use in developing the algorithms.

2.1 Stream Model

We consider streams with an infinite sequence of data items, where the items are added to the sequence over time and the sequence is ordered by the time-stamp at which each item is added to the stream. Accordingly, we model each stream data item as a binary



(a) 5-way W-join using a single window constraint

(b) Other forms of W-join operation

Figure 1: Variations of W-join.

tuple < v, t >, where *v* is a value (or set of values) of the data item, and *t* is the time-stamp that defines the order of the stream sequence. The value of the data item can be a single value or a vector of values, and each value can be a simple or composite data type. The *time* is our default ordering domain. The *time-stamp* is considered as 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* [18], where valid time is the time assigned to the item at its source stream, and transaction time is the time assigned to the data item at the query processing system. We refer to the source of any stream as a *sensor*. A sensor is any data source that is capable of providing infinite streams of data, either continuously or asynchronously.

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 black dots correspond to tuples from each stream that join together. The window restriction 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 a window away from each other.

It is evident from the W-join operation that we need an efficient approach for verifying window constraints between the input streams and for updating the join buffers to contain only eligible tuples. A brute-force approach to verify window constraints between streams requires verifying the constraint between each pair of N streams, adding C_2^N additional comparisons for each input tuple. A more efficient approach is suggested by Aref et al. [1] to verify that N objects, each from a different class, are within a fixed-radius from each other. We adopt a similar approach to verify window constraints among the individual tuples. The algorithms in [1] cannot deal with infinite streams and may block if data is delayed. In contrast, our proposed approach is non-blocking and does not require complete scans over the streams. For updating the join buffers we provide an online approach in contrast to a straightforward approach that updates the join buffers either periodically or as storage overflows. We will present the details of the algorithms in the next section.

The W-join in Figure 1(a) represents *the single* window constraint form of W-join. If we represent tuples from the streams as nodes in a graph, the W-join with a single window constraint can be represented as a complete 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). Figure 1(a) depicts the graph representation of W-join. The other variations of the W-join that consider different window constraints between the streams and/or partial window constraints are shown in Figure 1(b).

3 W-join with a single window constraint

The single window constraint for W-join is defined as follows:

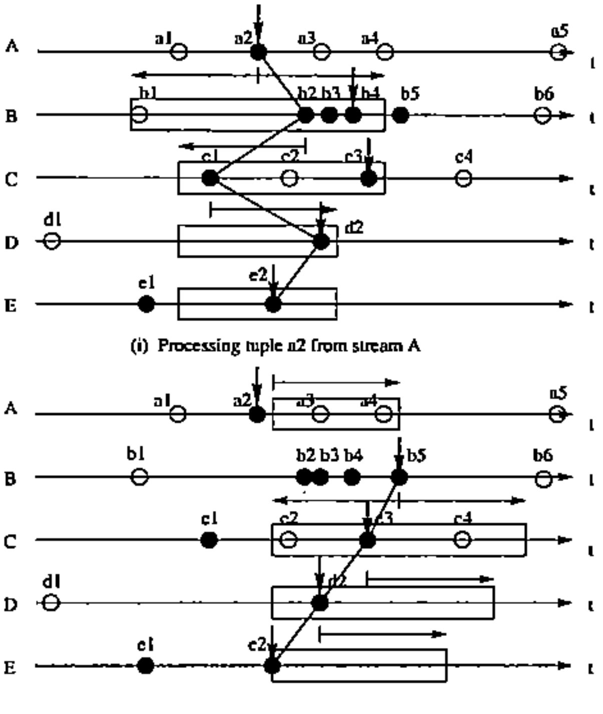
> *Given N data streams and a join condition (a boolean expression on the tuples' values), find the tuples that satisfy the join condition and that are within a sliding time window of length w units from each other.*

We use the term *distance* to refer to the time difference between two time-stamps, and the term *period* [18] to refer to an anchored interval in time, where the start and end time-stamps bound the period.

In the following sections, we present the NLW-join, HW-join algorithms and the MW-join algorithm.

3.1 The W-join Algorithms

We describe the approach followed by both the NLW-join and HW-join using an example W-join among five streams, A, B, C, D, E, as shown in Figure 2. In this example, the five streams are joined together using a single window constraint of length w that applies between every two streams. For illustration of the algorithm, we assume that only the *black* dots (tuples) from each stream satisfy the join predicate (e.g., equality over objectID). The W-join maintains a buffer for each stream and we assume that the *vertical* bold arrows are currently pointing to the *last* tuple processed from each stream.



(i) Processing tuple a2 from stream A

(ii) Processing tuple b5 from stream B

Figure 2: Two iterations of the W-join.

The algorithm processes each stream in order (i.e., Stream A is processed first then Streams B, C, D, E, A ...). At each iteration, a new tuple from a different stream is processed. If the current stream has no tuples, the next stream is processed. In other words, the algorithm *does not block* waiting for tuples from a single stream.

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_1 is white, i.e., it does not qualify the rest of the query predicate and hence does not join with a_2. b_2 satisfies the join predicate 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_2 and all tuples within w of either 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 process of checking the join condition is repeated for tuples in C. Since tuple c_1 satisfies the join predicate and also lies inside the rectangle, a new period is calculated that includes tuples a_2, b_2, c_1 and all tuples that are within w of any of them. This period is shown as a rectangle over Stream D. In Stream D, d_2 satisfies the join predicate and is located within the rectangle formed by a_2, b_2, c_1. A new period is formed which includes the previous tuples and any further tuples within w of any of them. This period is shown as a rectangle over Stream E. The step is repeated for Stream E, and the 5_tuples, $< a_2, b_2, c_1, d_2, e_2 >$ 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: $< a_2, b_2, c_1, d_2, e_2 >$, $< a_2, b_2, c_3, d_2, e_2 >$, $< a_2, b_3, c_1, d_2, e_2 >$, $< a_2, b_3, c_3, d_2, e_2 >$, $< a_2, b_4, c_1, d_2, e_2 >$, $< a_2, b_4, c_3, d_2, e_2 >$.

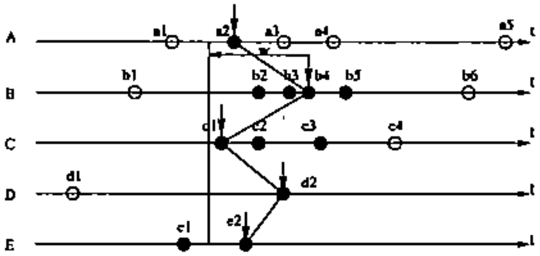


Figure 3: Tuple removal in W-join.

After finishing with tuple a_2, the algorithm starts a new iteration using a different stream. 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. This iteration produces no output, since no tuples join together in the constructed rectangles. Note that tuples in the rectangle over A are not yet inside A's buffer (after arrow location) and as such are not considered for the join. Those tuples will be processed in a later iteration when processing Stream A. The same applies to tuple c_4 in Stream C.

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

The algorithm must address the removal of *old tuples* from the buffer associated with each stream, where old tuples are those which will never W-join with any incoming tuples from the streams. We remove a tuple from a stream if it is located at a distance more than w from the last tuple in all other streams. In Figure 2 (i), we remove d_1 as we process Stream D, since d_1 is located at distance more than w from the last tuples of *all* streams (namely tuples a_2, b_4, c_3, e_2). Note that we cannot remove a tuple if it is at a distance more than w from only a single stream. This is shown by noting in Figure 3 that although e_1 is located at distance more than w from b_4 when tuple b_4 from Stream B is processed, when we later process Stream C, the tuple c_2 can join with e_1 as well as with a_2, b_2, d_2.

W-join is non-blocking in the sense that it does not stop if one of the streams has no tuples. Rather, it continues processing the tuples from the other streams. In addition, W-join can be easily implemented in the pipeline query plan. This is more evident if we note that the new tuple from each stream is actually compared with the tuples in the period constructed thus far, and those tuples can be produced from lower levels in the pipeline tree.

In the following sections, we show two pipelined implementations of W-join using nested-loop and hash approaches. We call these two implementations *NLW-join and HW-join*, respectively.

3.1.1 The Nested-Loop W-join Algorithm

We consider a left deep pipelined execution plan of the joins, where at each level in the pipeline an additional stream is introduced to the join and the source streams correspond to the leaves in the execution tree (see Figure 4.) The left stream may be a *source stream* or an *intermediate stream* from a lower level in the tree. The right stream is always a source stream. Except at the lowest node in the tree, tuples in the left stream do not have a single time-stamp per period. In other words, as the left m_tuples climbs the tree, an additional time-stamp is added, where m is the number of streams joined so far. We store the time-stamps of all tuples joined so far with the joined m_tuples (to handle the tuple removal and cover the anomaly case, which is described in Figure 3). The period of each m_tuples can be easily calculated as; $[TS_{max} - w, TS_{min} + w]$, where TS_{max} is the MAX(time-stamps of the m_tuples), and TS_{min} is the MIN(time-stamps of the m_tuples).

All the tuples received from one stream and not yet dropped are stored in a *stream buffer*. The buffer is either small enough to fit in memory, or part of it (the tail) is swapped to disk.

The pipelined algorithm alternates between retrieving tuples from the left and right streams. The algorithm does not block if one of the streams has no tuples; in this case, it continues retrieving tuples from the other stream. If the other stream blocks as well, the join produces an empty tuple to higher levels. An empty tuple signals higher levels to process other non-blocking streams or call the lower level again. As an m_tuples arrives from the left stream, it is used to iterate over all qualifying tuples in the right stream. Only tuples in the right stream that reside inside the period of the m_tuples are tested for the join predicate. If a tuple satisfies the join predicate from the right stream, a new (m+1)_tuples is reported to higher levels, and the time-stamp of the right tuple is added to the set of time-stamps already in the m_tuples. The same process is performed as a tuple arrives from the right stream.

To remove tuples from the buffer of a source and an intermediate stream, we need to verify that these tuples are not included within the window of any arriving tuples from *all* the other streams. For this purpose, the algorithm keeps a vector of time-stamps per buffer. This vector is updated to reflect the maximum time-stamps that appear in each source stream. For example, for right streams, the time-stamps vector $\mathcal{V}_{right}$ has a single value that represents the maximum time-stamp of the tuples from that stream. For left streams, $\mathcal{V}_{left}$ is a vector of maximum time-stamps for each source stream in its subtree. The time-stamps vector is shown in Figure 4. During the iteration of one stream, we remove the tuple that is located at a distance more than w from the tuples in the time-stamps vector of the other stream. This tuple is guaranteed not to W-join with later tuples appearing in the other stream.

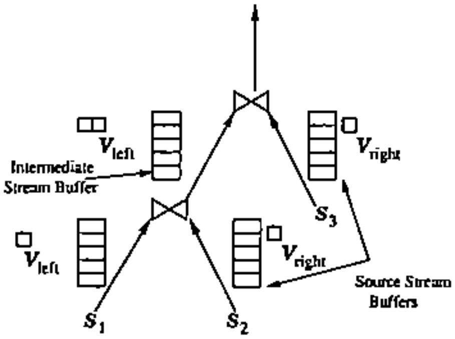


Figure 4: Binary Pipelined NLW-join.

In Figure 5, we present the GetNext algorithm of each join in the pipeline. The subscripts i and j represent left(0) and right(1) streams, respectively. The stream can be either a source or an intermediate stream. The variable TurnOnStream indicates the stream to process when starting the algorithm, the list CurrentJoinOutput stores the W-joined tuples. The Function GetPeriod(t_i) returns the period of tuple t_i. Function IsOut(t_k, $\mathcal{V}_i$, w) returns true if t_k is located at a distance more than w from every tuple in $\mathcal{V}_i$. Function IsIn(t_k, CurrentPeriod) returns true if t_k is included within CurrentPeriod. The iteration returns after scanning both streams, *ProcessedBothStreams*, to allow higher join nodes to process their streams.

3.1.2 The Hash W-join Algorithm

We now present the hash-based implementation for the W-join. We assume that the join predicate is an equality predicate over the join attribute. This algorithm uses an approach for updating the window that is similar to the one used by NLW-join. However, the algorithm builds hash tables based on the join equality attribute for both streams. The iteration for the HW-join is similar to the one in Figure 5, except that the new tuple is used to probe the hash table of the other stream instead of scanning it. As with the nested loop algorithm, we need to update the buffer (hash table) of one stream as tuples arrive from the other streams. However, for hashing, the probing tuple only visits part of the hash table, the bucket that has the same hash key. This may leave some buckets, which are probed infrequently, occupied with old tuples. To account for this situation, we call an update routine to remove old tuples from the other buckets as well. This routine is called for the rarely-probed buckets only in

```
GetNext(){
    if(CurrentJoinOutput!=NULL)
        return_next_output_tuple;
    While(1){
        i = TurnOnStream; j = (i+1) mode 2;
        Retrieve tuple, $t_i$;
        if ($t_i$ != NULL) {
            Update $\mathcal{V}_i$;
            Insert $t_i$ in stream$_i$'s buffer;
            CurrentPeriod= GetPeriod($t_i$);
            For all tuples $t_k$ in stream$_j$'s buffer
                if(IsOut($t_k$, $\mathcal{V}_i$, w))
                    delete $t_k$;
                else
                    if( IsIn($t_k$, CurrentPeriod))
                        If(CheckJoinPredicate($t_i$, $t_k$, $t_{result}$)
                            AddToOutput($t_{result}$, CurrentJoinOutput);
        }/* end if */
        TurnOnStream = (i+1) mode 2;
        if(CurrentJoinOutput!=NULL)
                return_first_output_tuple;
        else if(ProcessedBothStreams)
            /* both streams are empty, do not block and return */
                return NULL;
    }/* end while */
}
```

Figure 5: A single iteration of NLW-join

contrast to a naive periodic approach that scans all the buckets. It is worth noting that with a large number of arriving tuples and with non-skewed values of the hashing attribute, the buckets get updated without the need to call a separate update routine for them.

3.2 The Merge W-join Algorithm

Let "m_tuples" denotes the W-joined tuples between m streams at an intermediate stage during W-join execution. In the NLW-join and HW-join algorithms, when considering a new stream to join the m streams, we need to verify that the resulting m+1_tuples satisfies the window constraint. This test requires a boundary checking (two comparisons) and is repeated multiple times, for each tuple in the new stream. In this section, we present an algorithm where this test occurs only once, when a new tuple arrives, and the algorithm controls its buffers to keep only tuples that are within a window of each other. Our experimental study for this algorithm shows an order of magnitude speedup compared to the nested loop W-join algorithm. We refer to this algorithm as MW-join.

The MW-join iterates between two modes, *Mode 1* and *Mode 2*. In *Mode 1*, a plan sweep is performed on the time axis for all incoming tuples. The target is to locate the first N_tuples at window distance w from each other. As this tuple is found, a period, v, is constructed that includes all the N_tuples. In Mode 1, all the tuples beyond 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 time-stamps within the period v. 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.

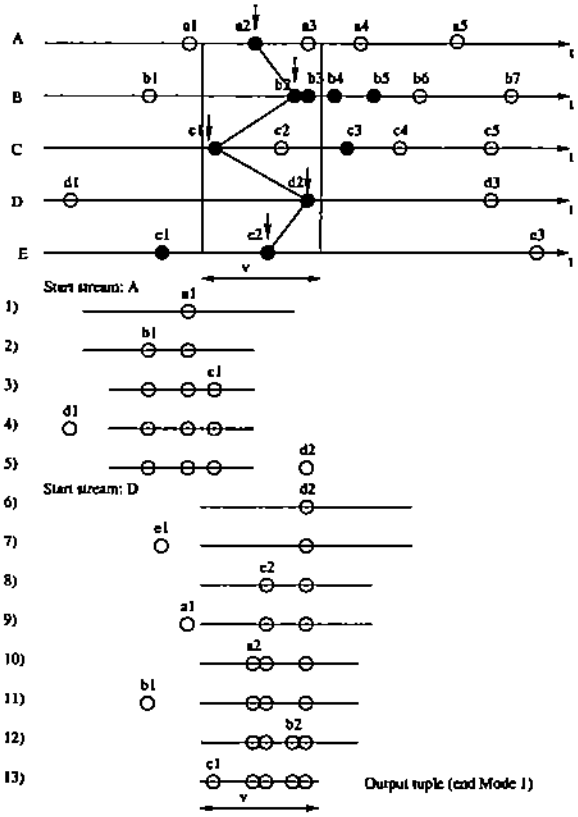


Figure 6: The Merge W-join Algorithm.

Figure 6 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 6. 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 modification is shown in Step 2. Stream C is processed next, and c_1 is tested for period inclusion. As c_1 is also included within the period, the period is modified to reflect the addition of c_1 and Mode 1 proceeds to Stream D. In Stream D, as d_1 is to the left of the current period, as a result, d_1 is dropped and the next tuple, d_2 is considered for Mode 1. Since d_2 is to the right of 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 to the left of 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. Stream A is processed again, and since tuple a_1 is to the left of the period, it is dropped and next tuple is tested. a_2 is in the period, the period is updated, and Mode 1 advances to Stream B. At Stream B, tuple b_1 is dropped (to the left of the period) and tuple b_2 is added to the period. Finally tuple c_1 in Stream C is added to the period and all the streams participate by a single tuple in the final period, v. This indicates the end of Mode 1. The final tuple becomes the starting N_tuples. Note that the join predicate is not considered during Mode 1, and only the period inclusion is verified.

Mode 2 then begins, iterating over all tuples in the neighborhood of the N_tuples and included within v. 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. Mode 2 admits the new tuple only if it is included within the period v, otherwise Mode 2 considers a new tuple from a different stream (assuming the order A, B, C, D, E, A, ...). Mode 2 does not *block* waiting for tuples to arrive at a specific stream, rather, it considers another stream meanwhile.

For each new and already admitted tuple, Mode 2 performs a nested loop join that includes the new tuple and all the tuples currently in the other streams' buffers. For example, as we consider tuple a_2 we perform a nested loop to test the join predicate for the tuple combination $< a_2, x_b, x_c, x_d, x_e >$, where x_b represents all tuple currently in the buffer of Stream B. After considering a_2, we check the tuples $< x_a, b_3, x_c, x_d, x_e >$, and so on. We stop Mode 2 when the next tuple from each stream is to the right of the current period v. The output of Mode 2 in our example is the set of N_tuples: $\{< a_2, b_2, c_1, d_2, e_2 >, < a_2, b_3, c_1, d_2, e_2 >\}$.

At this point, Mode 1 is restarted by dropping the tuple c_1 and searching for next starting N_tuples. Note that, all incoming tuples from the streams will not W-join with c_1. After Mode 1 succeeds in locating the next starting N_tuples, Mode 2 is restarted.

The interesting feature of this algorithm is the simplicity in determining and updating the period in Mode 1. In addition, Mode 2 only iterates over tuples that are within a window from each other (that is, those that belong to the period produced by Mode 1). This means that the algorithm *is only performing the necessary work without checking any tuples outside the window*. In addition, Mode 1 updates the buffers of each stream such that *Mode 2 needs to test the window inclusion only when admitting a new tuple*. Mode 2, blindly tests the join predicate with all tuples currently in the join buffer, with the guarantee that they are located at the correct window distance from each other.

Note that Mode 2 never produces spurious duplicate tuples. Each time Mode 2 starts, at least one pair of tuples is enumerated for the first time, and since we move forward in each iteration (no backtracking), duplicates cannot appear in the output.

4 Performance Study

We evaluate the performance of the NLW-join, HW-join and MW-join algorithms in a prototype stream database system using both real and synthetic data streams. We compare their performance with adapted versions of the ripple join and the XJoin to consider the window constraint and periodic updates of their join buffers. In the following section, we provide an overview of the implementation of the prototype stream database system. We then describe the workload data and introduce the experimental results.

4.1 Implementation

The three W-join algorithms (NLW-join, HW-join and MW-join) are implemented on a real database system, PREDATOR [16], which is modified to accommodate stream processing. We introduce an abstract data type *stream-type* that can represent source data types of streaming capability. Any stream-type must provide the following interfaces, *InitStream, ReadStream, and CloseStream*. This approach in the design is similar to *table functions* [15] in introducing external data sources to the database. The stream table has a single attribute of stream-type. 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 in 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 using the stream-type interfaces and supply data to be processed by the query execution engine. To interface the query execution plan to the stream manager, we introduce a *StreamScan operator* to communicate with the stream manager and retrieve new tuples. If no tuples are ready because the stream is blocked, the stream manager replies with a STREAM_NO_VALUE message and the StreamScan reports a NULL record to higher nodes in the query tree.

The queries over data streams are long running queries which are usually terminated by a stop request from the user or by a STREAM_END message received by the StreamScan. The window specification is added as a special construct for the query. For example, to W-join 3 streams (A, B, C), the window predicate is $WINDOW = w$, where w represents the length of the window in time units.

4.2 Workload Data

We performed our experiments on both real and synthetic data. For real data, we use the logs of the transactions from Wal*Mart stores. We consider joining multiple stores, where the join predicate is equal-

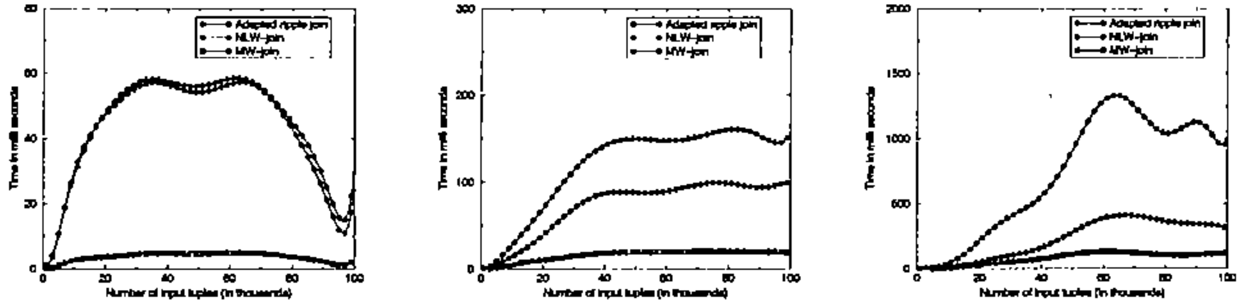


Figure 7: Comparing the performance between the adapted ripple join, NLW-join and MW-join for 2,3 and 4 streams using real data streams.

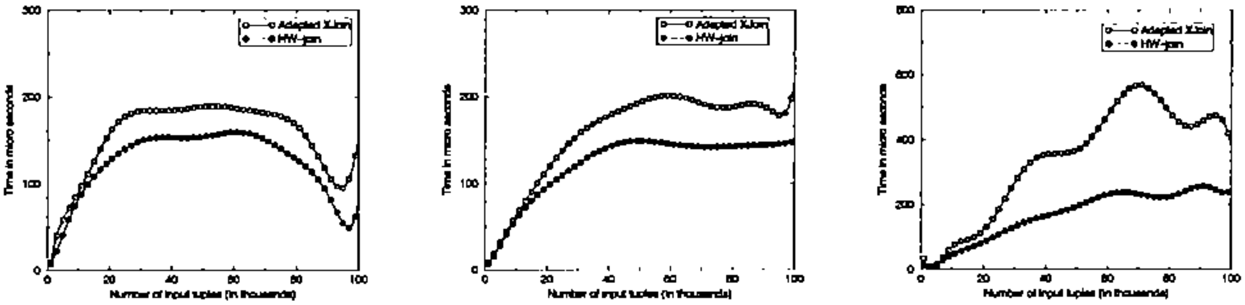


Figure 8: Comparing the performance between the adapted XJoin and HW-join for 2,3 and 4 streams using real data streams.

ity over the item number which is sold by different stores. A sold item in a store appears as a transaction (visit_scan) 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. Each transaction is a tuple with an approximate size of 200 bytes. The single store is a stream of items that have been sold. The time-stamp of each tuple is the time of the transaction, *valid time*. Our data is extracted from the NCR Teradata machine that holds 70GBytes of Wal*Mart data. Each store is represented in our system as a stream data type.

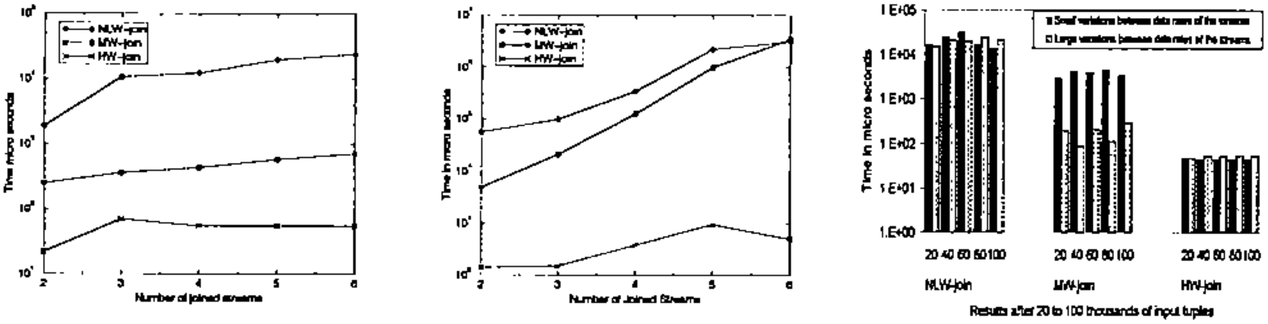
We also consider synthetic data streams, where each stream consists of a sequence of integers, and the inter-arrival time between two numbers follows the exponential distribution with mean λ. We consider a small set of integer numbers as values for the data streams in order to increase the selectivity of the join and test the algorithms under a considerably heavy workload. Note that increasing the selectivity will increase the number of comparisons in the iteration of any of the algorithms due to an increased number of joined tuples. The experiments are run on a Sun Enterprise 450 machine, running the Solaris 2.6 operating system with 4GBytes main memory.

4.3 Experimental Results

Our performance measure is the *service time* for each arriving tuple, which represents the time needed to handle an arriving item in the NLW-join, HW-join and MW-join algorithms. The service time depends on the number of comparisons in each iteration and reflects the efficiency of each algorithm. We collect the service time averaged by 1000 input tuples during the time of the experiment. We repeated the experiment multiple times to consider the trends in all the curves.

Figures 7 and 8 show the results for {2,3,4}-way join using real data streams with the sliding window set to 1 hour. In Figure 7 we compare the NLW-join and MW-join with a version of ripple join that is periodically updating the join buffers every 10,000 tuples to remove non-window related tuples. In Figure 8 the same experiment is performed for comparing the HW-join and a version of XJoin where the hash tables are updated periodically as in the ripple join. The steep increase and decrease in the curves for 2-way join are due to the nature of the real data as it represents transactions at the start and end operational hours of the stores (7:30 am to 8:00 pm).

Notice that, the plots do not build up as more tuples are processed from the streams (due to the effect of updating the join buffers). This is more clear when comparing the plots of our algorithms (NLW-join, MW-join and HW-join) with the adapted ripple



(a)

(b)

(c)

Figure 9: The left two graphs represent W-join for 2-6 data streams with variable and comparable data rates, respectively. The right graph illustrates the sensitivity of MW-join to variation in stream data rates.

join and XJoin that are updated periodically (the two rightmost graphs in Figures 7 and 8). It is clear that the NLW-join and HW-join outperform the ripple join and XJoin in all cases. This was the case in all experiments, so we drop the ripple join and the XJoin plots in the remaining figures.

The MW-join algorithm achieves the best performance when compared to NLW-join and the ripple join (almost an order of magnitude faster than the NLW-join). This is expected since MW-join avoids repeated iterations over non-window related tuples and uses only two comparisons to verify the window relationship with other tuples.

In Figures 9(a) and 9(b), we test the performance of the NLW-join, MW-join and HW-join algorithms as we increase the number of joined streams. We repeat this experiment for both synthetic and real data streams. Figures 9(a) gives the results when using synthetic data streams of variable data rates (between 1000 and 100 tuples/sec). Figures 9(b) gives the results of the experiment when using real data streams (the off-line analysis of the real data streams shows an average data rate of 5000 transactions/hour per all streams). For variable rate data streams, the MW-join is more than an order of magnitude better than NLW-join and this relative performance is maintained even as we increase the number of streams. When we consider data streams with comparable data rates (Figures 9(b)), the performance of the NLW-join and the MW-join converges to each other. This is due to the fact that as more streams of comparable data rates are joined together, the number of tuples compared during Mode 2 of the MW-join is increased, and the saving in comparisons per single tuple is dominated by the increase in the number of tuples. In other words, Mode 2 of MW-join (the nested loop) becomes similar to NLW-join. As a final note from this experiment, HW-join is more scalable and achieves the best performance. However, hash joins is best used with equijoin, which is a restricted case compared with NLW-join and MW-join.

Our next experiment is to test the *sensitivity* of our algorithms in situations where one of the streams has a slow arrival rate. We use the synthetic data for four streams (A, B, C, D) and measure the service time per input tuple when all joined streams have comparable rates, 1000 tuple/sec or $\lambda = 0.001$. We run the experiment again but with Stream D changing its rate between 10 and 1000 tuples/sec ($\lambda = 0.1$ and $\lambda = 0.001$) every 100 input tuples. As illustrated in Figure 9(c), MW-join reacts rapidly to changes in the stream rates and its performance is greatly improved. In fact MW-join runs most of the time in Mode 1 trying to locate the starting N_tuples. The performance for the algorithms NLW-join and HW-join are not changed significantly.

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, these refer to Form 2 and Form 3 described in Section 1. The path W-join is presented as a special case of Form 3, the partially constrained form of the W-join. Finally, we present the experimental study.

5.1 W-join with Different Windows Constraints

The NLW-join, HW-join and MW-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 between all streams, and apply the NLW-join, HW-join or the MW-join algorithm 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 needs C_2^N comparisons, but will only be applied to the output from the filtering step. We refer to this approach as a *global conservative approach, GCA*.

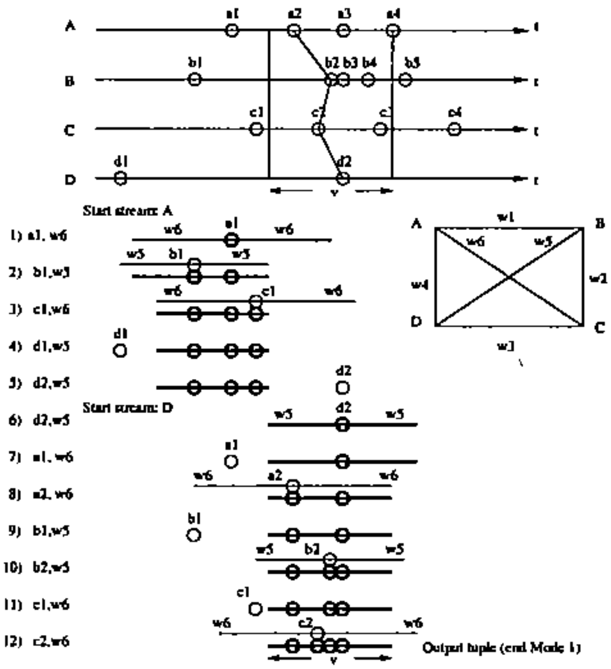


Figure 10: The local conservative approach.

The GCA may assign an unnecessarily large window during the join, resulting in an excessive number of additional comparisons. We propose a variant of the MW-join algorithm that avoids working with the maximum window size. The modified algorithm considers the maximum local windows for each stream. We call this approach *the local conservative approach, LCA*.

The LCA has the same filtering and refinement steps as the GCA . The filtering step also alternates between two modes, Mode 1 and Mode 2, similar to the MW-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 10. 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).

As shown in the figure, Mode 1 considers window of sizes w_6 and w_5 as it processes Stream A and B respectively. Due to space restriction we omit the detailed description for Mode 1. The final output of Mode 1 is the starting N-tuples $< a_2, b_2, c_2, d_2 >$ and the period v. At this point, Mode 2 of the LCA starts to enumerate all tuples in the vicinity of the N_tuples $< a_2, b_2, c_2, d_2 >$. The join predicate is applied and finally the refinement step is performed to the output. In the refinement step, the window restrictions is applied for each pair of tuples.

5.2 Partially Constrained W-join

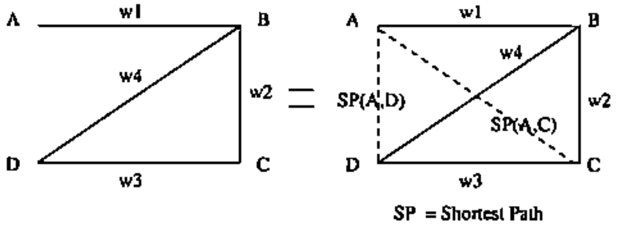
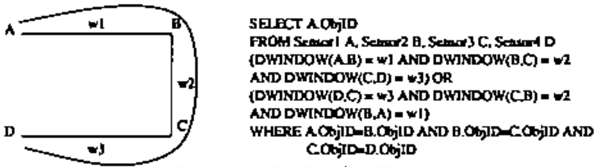


Figure 11: Graph construction for the partially constrained W-join.

If one or more pairs of the streams are not window-constrained[1], 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 constraint. Figure 11 shows a 4-way W-join, where the window constraints between Streams A, C and A, D are missing. The upper bound to the missing constraint is *the shortest path* between these two streams. For example, in Figure 11, the window constraint between Streams A, D is MIN($w_1 + w_2 + w_3$, $w_1 + w_4$). Note that, the calculation of the shortest path is only performed at query compilation time.

We repeat the construction of the bound evaluation for all missing window constraints until arriving at a complete graph. At this point, the problem can be solved by either one of the conservative approaches described in the previous section.

5.2.1 The Path W-join



```
SELECT A.ObjID
FROM Sensor1 A, Sensor2 B, Sensor3 C, Sensor4 D
{DWINDOW(A,B) = w1 AND DWINDOW(B,C) = w2
AND DWINDOW(C,D) = w3} OR
{DWINDOW(D,C) = w3 AND DWINDOW(C,B) = w2
AND DWINDOW(B,A) = w1}
WHERE A.ObjID=B.ObjID AND B.ObjID=C.ObjID AND
          C.ObjID=D.ObjID
```

Figure 12: Path W-join.

In this section, we consider a special query that targets the partially constrained W-join. The query is to retrieve tuples from the streams that satisfy the window constraints along a specific path. Consider the tracking of an object by multiple monitoring devices installed along a certain path. The query is to find the object that follows this path. For example, in Figure 12, a candidate output needs to appear at any of the paths: *ABCD, DCBA*. Tuples from Streams A, B, C, D need to satisfy the set of the window constraints $\{w_1, w_2, w_3\}$, in addition to the join predicate (e.g.,

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

equality on the object ids). We consider a simple bidirectional path (no cycles[2]). The SQL-like representation is also shown in Figure 12, where DWINDOW indicates a directed window constraint between tuples that appear in Stream A first then Stream B (not the opposite). The query is evaluated, as in Section 5.2, by reconstructing the complete graph and applying the conservative approaches.

It is important to note that this query can be answered using the NLW-join or HW-join discussed in Section 3.1. However, the detection of the paths ABCD and the reverse path DCBA requires the union of two queries. The first query is evaluated to detect the path ABCD and the second query is evaluated to detect the path DCBA. In this case, both algorithms are modified to iterate in a certain order and propagate a period controlled by only one of the joined streams. For example, in detecting the path ABCD, joining A and B propagates a period in the form [TS_b, TS_b+ w_2], where TS_b indicates time-stamp of a tuple b at the Stream B and w_2 is the window between the Streams B and C.

5.3 Performance Study for the Variations of W-join

We implemented the global and local conservative approaches for the W-join with different window constraints in our prototype system. We add the final refinement step before producing tuples as output to higher query operators. In our experiment, we use synthetic data as in Section 4.2 with data rates equals 1000 tuples/sec ($\lambda = 0.001$) and a 4-way W-join. The window among all streams is set to 0.8 second, except between one pair of the streams where it is set to 1 sec. The GCA considers the 1 sec window as the fixed-size window whereas the LCA considers the local windows per each stream (in our example the window equals 0.8 second per two streams and 1 second for the other streams). We repeated the experiment where we choose 0.1 seconds instead of 0.8 seconds. The results are illustrated in Figure 13. ¿From the figure, the LCA outperforms the GCA even for small variations between window values.

For testing the Path W-join, we use both the LCA (where we reconstruct the complete window constraints) and the modified NLW-join where we join the streams pairwise in a pipeline fashion. In Figure 14, we introduce the results for joining four streams and the window constraints is described along the path ABCD or DCBA as in Figure 12. We run the experiment the first time for streams with small variations between their data rates and another time with large variation in data rates. As illustrated in Figure 14, for small variations, the NLW-join outperforms the LCA, however for large variations LCA performs the best (LCA is based on MW-join which is significantly fast in seeking the correct prefixes of the stream to iterate within).

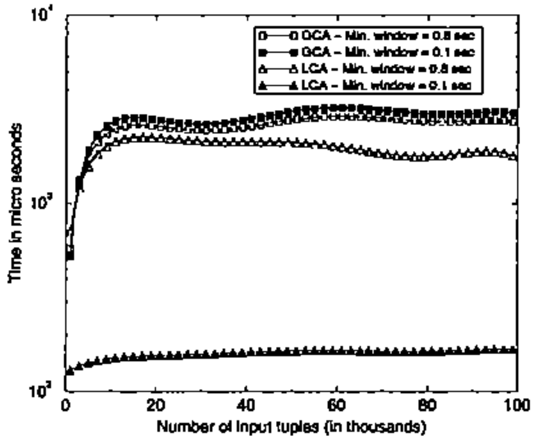


Figure 13: GCA vs LCA for W-join with different window constraints.

6 Related Work

Praveen et al. [17] provide the SEQ model and implementation for sequence database. The sequence is defined as a set with a mapping function to a defined ordered domain. The work in [12] 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 [11], the work includes a study of algorithm complexity on computation over data stream. Recent work on computing correlated and approximate average over data streams is described in [8, 9]. The work on [8] proposes a one pass algorithm to compute the correlated aggregate using either a fixed or sliding window over data streams. In [9] the authors propose the use of wavelet transformation methods to provide small space representations of the stream for answering aggregate queries. Adaptive query processing [2] and execution of continuous query [5] address reordering of operator during execution and the execution of long running queries. The COUGAR [4] system focuses 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 [3] project discusses the new demands imposed by data streams on data management and processing techniques. The band join [6] technique addresses the problem of joining two relations (of fixed sizes) for values within a "band" of each other. Tribeca [19] is a specialized query processing system designed to support network traffic analysis. The system mainly focuses on query processing over streams of network traffic either online or off-line. Index-based and partition-based algorithms are presented in [22, 13] for temporal-join over finite relations. The SQL presentation of the sequence and temporal join is presented in [18].

[2] Complex paths such as the path ABCBCD introduces a cycle, self-join for Streams B and C.

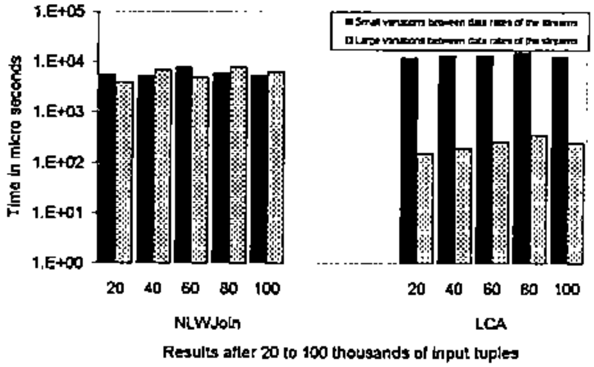


Figure 14: NLW-join vs LCA for path W-join using streams with comparable and variable data rates.

7 Conclusion

In this paper we introduce a class of join algorithms, referred to as *W-join*, for joining multiple infinite data streams. We provide a general graph representation of the W-join, and its different variations. We describe three W-join algorithms, the NLW-join, the HW-join and the MW-join. The NLW-join and the HW-join algorithms can be easily implemented in the query pipeline plan and use an efficient approach to verify the window constraint between the streams. The MW-join algorithm outperforms the NLW-join algorithm in terms of execution time per input tuple. The MW-join algorithm provides a major performance speedup and sensitivity in cases where the streams are of variable rates. We compared our algorithms with adapted versions of the ripple join and the XJoin to handle W-join queries and using a real implementation on a prototype stream database system. The performance study indicates major performance speedup when using our algorithms. We studied the different ways to solve the other variations of W-join using adaptation of our algorithms and utilizing the filter-refine paradigm. We tested the adapted algorithms and we presented the performance study for each.

References

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

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

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

[4] 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.

[5] 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.

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

[7] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of SIGMOD Conference*, 1999.

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

[9] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proc. of 27th VLDB Conference, September*, 2001.

[10] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of SIGMOD Conference*, 1999.

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

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

[13] H. Lu, B. C. Ooi, and K. L. Tan. On spatially partitioned temporal join. In *20th VLDB Conference, Sept.*, 1994.

[14] S. Madden and M.J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[15] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. T. Tran, and S. Vora. Heterogeneous query processing through sql table functions. In *Proc. of the 15th ICDE, March*, 1999.

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

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

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

[19] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX, New Orleans, Louisiana, June*, 1998.

[20] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin 23(2)*, 2000.

[21] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the 1st PDIS Conference, Dec.*, 1991.

[22] D. Zhang, V. J. Tsotras, and B. Seeger. Efficient temporal join processing using indices. In *ICDE*, 2002.