2003

Scheduling for Shared Window Joins Over Data Streams

Moustafa A. Hammad

Michael J. Franklin

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

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

Report Number:
03-001

SCHEDULING FOR SHARED WINDOW JOINS OVER DATA STREAMS

Moustafa A. hammad
Michael J. Franklin
Walid G. Aref
Ahmed K. Elmagarmid

CSD TR #03-001
January 2003

Scheduling for shared window joins over data streams

Moustafa A. Hammad
Purdue University
mhammad@cs.purdue.edu

Michael J. Franklin
University of California at Berkeley
franklin@cs.berkeley.edu

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

Ahmed K. Elmagarmid
Hewlett Packard
ahmed_elmagarmid@hp.com

ABSTRACT

This paper addresses the problem of scheduling the shared execution of multiple window-join queries over data streams. Each join has its own sliding window, which can be expressed in terms of time units or tuple counts. Joining tuples from the underlying data streams may serve the purpose of multiple window join queries. Sharing the execution of these queries will maximize the utilization of system resources. One way to share the execution of multiple window joins is to use the largest window among all queries. This approach, the Largest Window Only(LWO), would penalize the response time of queries with small windows to serve the query with the largest window size. Two new algorithms for scheduling the execution of the shared window-join are presented; the Smallest Window First(SWF) and the Greedy algorithms. An analytical study of the tradeoffs between the LWO and SWF algorithms leads to the development of the Greedy algorithm. The performance study of the three algorithms show that the Greedy algorithm provides the best performance in terms of response time for all queries.

1. INTRODUCTION

Data stream processing has recently become a topic of intensive interest in the database community. The reasons for this interest are several. First, in many emerging applications, particularly in pervasive computing and sensor-based environments, data streams play a central role, because devices continuously report up-to-the minute readings of sensor values, locations, status updates etc. Data streams also feature prominently in other networked applications such as "real-time" business processing and enterprise application integration. Secondly, data streams break many of the assumptions upon which traditional query processing technology is built, providing the opportunity to rethink many fundamental database management techniques.

One major difference that arises in data stream processing systems (compared to more traditional stored database management) is the notion of long-running continuous queries over those streams. The emerging data stream processing architecture involves potentially large numbers of such queries that are effectively constantly running, and that continuously react to new data as it arrives at the system. The availability of a significant set of queries raises the potential to aggressively exploit shared processing of common work needed by multiple queries. Furthermore, the high data rates and requirements for split-second responsiveness in many streaming applications dictate that such opportunities for efficiency be exploited.

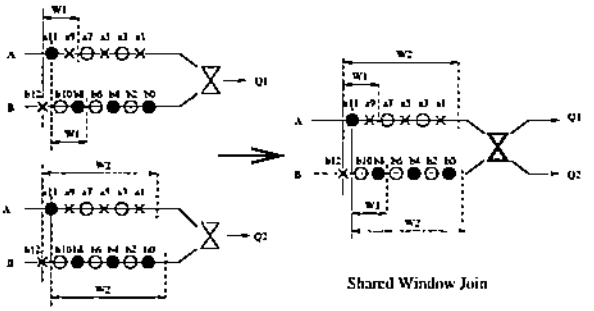


Figure 1: The shared execution of two window joins.

In this paper, we focus on a fundamental problem that arises in systems for processing continuous queries over data streams. Namely, we investigate the problem of scheduling processing for multiple windowed joins over a common set of data streams. We show that the approach followed in earlier systems has a previously unreported performance problem that discriminates against queries that have small windows. Ironically, it is exactly such queries that are likely to have strict responsiveness constraints. We propose two new scheduling algorithms that address this problem and analyze their performance in detail using a prototype database system extended to support these algorithms.

2. BACKGROUND

2.1 Context and Environment

We consider a centralized architecture for stream query processing in which data streams continuously arrive to be processed against a set of standing continuous queries (CQs). A key issue in the design of such a system is the efficient processing of the CQs as data arrives, particularly if data rates are high or if there are a large number of active CQs in the system.

In this paper, we consider streams to be unbounded sequences of data items. A stream may represent readings from a sensor device or a set of sensors (e.g., temperature sensors, financial and news tickers, network monitors). Each data item in a stream is associated with a time stamp that identifies the time at which the data item enters the system. The data items of a single stream may arrive in a burst fashion (a group of data items arriving within a short period of time) or they may arrive in equally-spaced intervals. Examples of the first type are network traffic streams, phone call records, and sensors that push data to the system triggered by an independent event (monitoring sensors). Examples of the second type are pull-based sensors, where the system contacts the sensor device to retrieve data periodically (e.g., once every second.) Our discussion focuses on streams with a burst arrival pattern.

Queries over streams often exploit the temporal nature of stream data. Furthermore, due to the unbounded nature of streams, queries over streams are often defined in terms of sliding windows. For example, consider a data center containing thousands of rack-mounted servers, cooled by a sophisticated cooling system.In modern data centers, sensors are used to monitor the temperature and humidity at locations throughout the room. For a large data center, thousands of such sensors could be required. A control system monitors these sensors to detect possible cooling problems.

We can model this example scenario as a system with two streams, one for temperature sensors and one for humidity sensors. The schema of the streams can be of the form (LocationId, Value, TimeStamp), where LocationId indicates a unique location in the data center, Value is the sensor reading, and TimeStamp is as described above. A window query that continuously monitors the sensors to determine if both the humidity and temperature values exceed specific thresholds within a one minute interval could be specified as follows:

SELECT T.LocationId
FROM Temperature T, Humidity H
WHERE T.Value > $Threshold_t$ and H.Value > $Threshold_h$
and T.LocationId = H.LocationId
WINDOW 1 min;

A second example query continuously searches for sensors with similar heating behavior within a 30 minute interval and applies a user-defined function, *correlated*, to determine similarity between the temperature and humidity values as follows:

SELECT T.LocationId, H.LocationId
FROM Temperature T, Humidity H
WHERE *correlated*(T.Value, H.Value) and T.LocationId
$\neq$ H.LocationId
WINDOW 30 min;

The WINDOW clause in the query syntax indicates that the user is interested in executing the queries over the sensor readings that arrive during the time period beginning at a specified time in the past and ending at the current time. When such a query is run in a continuous fashion, the result is a sliding window query.

Window queries may have forms other than the time sliding window described in the preceding examples. One variation of the window join is to identify the window in terms of the number of tuples instead of the time units. Another variation is to define the beginning of the window to be a fixed rather than a sliding time. Other variations associate different windows with each stream [11] or with each pair of streams in a multi-way join. In this paper, we address sliding windows that are applied across all streams and where the windows can be defined either in terms of time units or tuple counts. We present our algorithms using time windows, and in a separate section describe how our algorithm can be applied to windows defined in terms of tuple counts.

As with any query processing system, resources such as CPU and memory limit the number of queries that can be supported concurrently. In a streaming system, resource limitations can also restrict the data arrival rates that can be supported. Some recently stream query processing systems (e.g., Aurora [4], Telegraph [12] and STREAM [3]) propose mechanisms to respond to resource overload by reducing quality of service (e.g., dropping tuples from the input or answers). In contrast, in our work, we focus on the case where no loss occurs. That is, we ensure that the system is run at a rate where it is possible to execute all queries correctly. While such a restriction may be unsupportable in some applications, our main argument is that the workload volume that can be sustained by a shared CQ system can be dramatically increased by exploiting, wherever possible, shared work among the concurrent queries.

There are a number of recently published algorithms that support shared execution between multiple queries over data streams [7, 6, 12, 5]. One result from this previous work is that for shared CQ systems, the sharing of join processing among queries can be greatly enhanced through the use of "selection pull up" [6]. In a shared CQ system, the traditional heuristic of pushing selection predicates below joins would significantly reduce the potential for sharing the work of multiple joins, because such joins would effectively have different signatures (i.e., they would be over different streams). Thus, in a CQ system it is usually beneficial to pull the selections up above the joins, thereby allowing more queries to share the output of a single join operator. Our work is very much in this spirit, and in fact, extends the sharing approach to efficiently allow sharing of join processing across all queries that share a join predicate, regardless of their window specification.

2.2 Problem Definition

Consider the case of two or more queries, where each query is interested in the execution of a sliding window join over multiple data streams. We focus on the set of concurrent queries that have the same join predicate over the same data streams[1], and where each query has a sliding window that represents its interest in the data. The goal is to share the

[1]Note, the restriction to a single join predicate allows us to use hash-based implementations of the algorithm. The nested loop based implementations could be extended to deal with different join predicates

execution of the different window joins to optimize the utilization of system resources.

We illustrate this definition using an example of two queries in Figure 1. In the figure tuples arrive from the left, and are tagged with their stream identifier and timestamp. We indicate tuples that satisfy the join predicate (but not necessarily the window clause) by marking them with the same symbol (e.g., cross, black circle, etc.). In the figure, Q_1 performs a join between the two streams A and B, using predicate p with window size w_1 = one second. Q_2 performs a join between the same two streams A and B, using predicate p with window size w_2 = one minute. We assume that new tuples are appended to the left of each stream. There is an obvious overlap between the interests of both queries, namely, the answer for Q_1 (the smaller window) is included in the answer for Q_2 (the larger window). We refer to this as the *containment property*, that is, the answer of any query is also contained in the answer of the queries with larger windows.

Executing both queries separately wastes system resources. The common join execution between the two queries will be repeated twice, increasing the amount of memory and CPU power required to process the queries. Implementing both queries in a single execution plan (Figure 1, right) is a significant improvement in resource usage. The new shared join operator has the same common input data streams and produces multiple output data streams for each separate query. The output data streams are identified by the distinguishing query windows sizes, and at least one query must be attached to each output data stream. The shared join operator is divided into two main steps: the join step and the routing step. The join step produces a single output stream for all queries and the routing step produces the appropriate output data streams for the various queries.

While shared execution has significant potential benefits in terms of scalability and performance, we need to ensure that such sharing does not negatively impact the behavior of individual queries. That is, the shared execution of multiple queries should be transparent to the queries. We define two objectives for such transparency:

1. A query executed in a shared fashion should produce the identical resulting output data stream as if it were run individually. In other words, the shared execution should not alter the content or the order of the output data stream.

2. The response time penalty imposed on any query when a new query is included in a shared plan should be kept to a minimum.

This paper investigates methods for sharing the execution of multiple window join queries which satisfy these two objectives.

3. THE SCHEDULING ALGORITHMS

In this section, we present three scheduling algorithms for performing a shared window join among multiple queries. These are: Largest Window Only (LWO), Shortest Window First (SWF), and Greedy. LWO was implicitly used, but not elaborated upon in [12, 5]. LWO is a natural way to address the problem of shared join processing, but as we will see, has some significant performance liabilities. The SWF and Greedy algorithms are contributions of this paper. We present all three algorithms in the same framework and study the performance tradeoffs of each.

One important consideration for all three scheduling algorithms is the order in which the output tuples are produced. We adopt a "stream-in stream-out" philosophy. Since the input stream is composed of tuples ordered by some timestamp, the output tuples should also appear as a stream ordered by a timestamp. In our algorithms, the output tuples are emitted as a stream ordered by the maximum time stamp of the two tuples that form the join tuple.

As described in Section 2.2, the shared execution of window joins should abide by the isolated execution property, i.e., each window join, say j_w, that is participating in the shared execution, produces an output stream that is identical to the output stream that j_w produces when executing in isolation. All three scheduling algorithms presented in this section abide by this property. Note that in this section we describe the algorithms assuming a nested loops-based implementation. As will be described in Section 3.4, all of the algorithms can be implemented using either nested loops or hashing.

3.1 Largest Window Only (LWO)

The simplest approach for sharing the execution of multiple window joins is to execute a single window-join with a window size equal to the maximum window size over all queries. Due to the containment property, the processing of the maximum window query will produce output that satisfies the smaller window queries as well. The join operation then needs to route its output to the interested queries. We call this approach *Largest Window Only*, or LWO for short.

The join is performed as follows. When a new tuple arrives on a stream, it is matched with all the tuples on the other stream that fall within the time window. This matching can be done in a *nested loops* fashion, working backwards along the the other stream, from most to least recent arrival, or can be done using *hashing* as described in Section3.4. Tuples can be aged out of the system once they have joined with all subsequently arriving tuples that fall within the largest window.

To perform the routing step for the resulting tuples, the join operator maintains a sorted list of the windows that are interested in the results of the join. The windows are ordered by window size from smallest to largest. Each output tuple maintains the maximum and minimum timestamps of the input tuples that constitute the output tuple. The routing step uses the difference between these two timestamps to select the windows, and hence the output data streams, that will receive this tuple. The output tuple is sent to all output streams that have windows greater than or equal to the time difference of the tuple.

We illustrate the operation of the shared window join with the example shown in Figure 2. The figure shows a shared

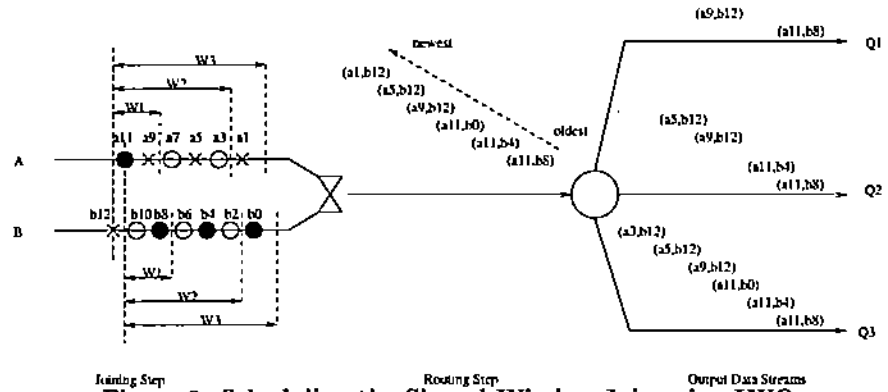


Figure 2: Scheduling the Shared Window Join using LWO.

window join over two data streams A and B. The join is shared by three queries, Q_1, Q_2, and Q_3 with window sizes (ordered from smallest to largest), w_1, w_2, and w_3, respectively. In the figure, tuples with similar symbols join together (i.e., they satisfy the common join predicate). The join step uses window w_3 for the single window join since w_3 is the largest window. As tuple a_{11} arrives, it joins with tuples b_8, b_4, b_0 in Stream B and the output tuples are streamed to the the routing step. The routing step determines that the output tuple (a_{11}, b_8)must be routed to all three queries, tuple (a_{11}, b_4) be routed to queries Q_2 and Q_3 and tuple (a_{11}, b_0) be routed only to query Q_3. After completing the join of tuple a_{11} with stream B, the join step begins to join tuple b_{12} with stream A. The resulting output tuples are $(a_9, b_{12}), (a_5, b_{12}), (a_1, b_{12})$ and they are routed in the same way to the queries.

One advantage of LWO, besides its simplicity, is that arriving tuples are completely processed (i.e., joined with the other streams) before considering the next incoming tuple. In this way, the output can be streamed out as the input tuples are processed, with no extra overhead. This property satisfies our objective of isolated execution. However, LWO delays the processing of small window queries until the largest window query is completely processed. In the preceding example, query Q_1 cannot process tuple b_{12} until tuple a_{11} completely joins a window of size w_3 from stream B. This means that tuple b_{12} waits unnecessarily (from Q_1's perspective) and increases the output response time of query Q_1. The effect is more severe as we consider large differences between the smallest and largest windows. Thus, LWO may not satisfy our other objective, as a large window query could severely degrade the performance of smaller window queries. In the following section we examine the average response time of each window involved in the shared window join when using the LWO algorithm.

3.1.1 *Analysis of response time*

In this section, we analyze the average response time of N queries sharing the execution of a window join operator. We assume that the shared window join operates on only two streams and that each query Q_i has a unique window, w_i. The mean time between tuple arrivals at each stream follows an exponential distribution with rate λ tuples/sec. The size of the join buffer (the amount of memory needed to hold the tuples for the join operation) for each stream differs for every query and is determined by the window size associated with the query. The buffer size S_i per stream for an individual query Q_i is approximately equal to $S_i = \lambda w_i$. Let w_{max} be the maximum window size among all the N query windows and S_{max} be the maximum buffer size per stream. Then, $S_{max} = \lambda w_{max}$. As a new tuple arrives, the expected number of tuples that join with this tuple inside a query window w_i can be estimated by αS_i tuples, where α is the selectivity per tuple.

Consider the case when m tuples arrive simultaneously in one of the streams, say stream A. LWO needs to schedule the execution of the window-join of each of the m tuples with the tuples in the other stream, say stream B. Each of the m tuples in A is checked against S_i tuples in B. Let $ATime(a)$ and $CTime(a)$ be the arrival and completion times of tuple a, respectively. For query Q_i, let AvgRT(Q_i) be the average response time of joining each of the m tuples for query Q_i. Then,

$$AvgRT(Q_i) = \frac{\sum_{k=1}^{m\alpha S_i}(CTime(joinTuple_k) - ATime(joinTuple_k))}{m\alpha S_i}$$

where the sum is taken over all output join tuples. Let $joinTuple_k$ corresponds to the tuple (a_i, b_j). Since $joinTuple_k$ is an output tuple of window w_i, then, $|ATime(a_i) - ATime(b_j)| < w_i$ and $ATime(joinTuple_k) = max(ATime(a_i), ATime(b_j))$. $CTime(joinTuple_k)$ represents the time at which the output tuple is received by Q_i . For simplicity of the analysis, let $\alpha = 1$.

Let t_p be the time needed to check that a tuple pair, say (a_i, b_j), satisfies the join predicate and the window constraint $|ATime(a_i) - ATime(b_j)| < w_i$. Then, for window w_i, the first tuple of the m tuples will produce S_i output tuples with a total delay of $t_p + 2t_p + \cdots + S_i t_p$ or $\frac{t_p}{2} S_i(S_i + 1)$. The second tuple of the m arriving tuples will have an additional delay of $t_p S_{max}$ as the second tuple has to wait until the first tuple scans the maximum window. Similarly, the third tuple will have additional delay of $2t_p S_{max}$ and so on. By averaging the response time of all m input tuples, therefore,

$$AvgRT(Q_i) = \frac{t_p}{2}((S_i + 1) + (m - 1)S_{max}) \qquad (1)$$

To clarify this equation we plot the AvgRT for multiple queries while using the following values: $t_p = 0.01$ usec, $\lambda = 100$ Tuples/Sec, $m = 50$ tuples. The windows are cho-

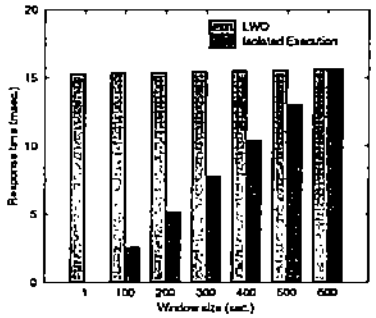


Figure 3: LWO versus Isolated Execution.

sen to span a wide range (from 1 second to 10 minutes) as follows ($w_1 = 1second, w_2 = 100sec., w_3 = 200sec., w_4 = 300sec., w_5 = 400sec., w_6 = 500sec., w_7 = 600sec.$). Figure 3 compares the average response time for each query when executed in isolation from the other queries, with the average response time of the query when executed using LWO. When executed in isolation, Q_i's average response time is $\text{AvgRT}(Q_i) = \frac{1}{2}(mS_i + 1)$[2]. It is clear from the graph that the query with smallest window, i.e., Q_1 (with $w_1 = 1sec.$) is severely penalized when using LWO. This penalty is expected because newly arriving tuples have to wait until the old tuples scan the largest window. While a simple analysis clearly predicts these results, it is important to recall that LWO is the only previously published scheduling approach for shared join processing in CQ systems.

These analytical results are validated by experiments on an actual implementation of the algorithm, which is reported in Section 5.1.1.

3.2 Smallest Window First (SWF)

To address the performance issues that arise with small windows in LWO, we developed an alternative approach called Smallest Window First (SWF). As the name suggests, in this algorithm, the smallest window queries are processed first by all new tuples, then the next (larger) window queries and so on until the largest window is served. A new tuple does not proceed to join with a larger window as long as another tuple is waiting to join with a smaller window. SWF is not quite as straightforward as LWO. As will be seen in the following, it requires significant bookkeeping.

We illustrate SWF with the example in Figure 4, which has the same configuration as that of Figure 2. When tuple a_{11} arrives, it scans a window of size w_1 in stream B. The result is the output tuple (a_{11}, b_8). After this scan, tuple b_{12} arrives and is waiting to join. Since tuple b_{12} will join window w_1 (the smallest window), b_{12} is scheduled immediately. Tuple a_{11} has not finished its join with stream B so it is stored along with a pointer to tuple b_6. Now b_{12} scans a window of size w_1 in stream A, resulting in the output tuple (a_9, b_{12}). The scheduler is invoked again to switch to tuple a_{11}. Tuple a_{11} proceeds to join with the remaining part of window w_2, namely, the partial window $w_2 - w_1$ in stream B. The resulting output is (a_{11}, b_4). The scheduler then switches back to tuple b_{12} to join with the remaining part of window w_2, the partial window $w_2 - w_1$, in stream A. The process continues until tuple b_{12} joins with the partial window $w_3 - w_2$, of stream B. Figure 4 shows the output upto this point.

SWF needs to store bookkeeping information with the arriving tuples. When the scheduler switches from serving one tuple to serving another, the current status status of the first tuple must be maintained. This status describes where to resume scanning in the other stream and the new window size (the next window size) to be applied. When a tuple gets rescheduled, it starts to join beginning at this pointer until completing the new window.

Note that the output of the joining step is *shuffled* when compared with the LWO scheduling. This shuffling occurs as we switch back and forth to serve the different arriving tuples. To produce the desired output stream for each query we need to modify the routing step from that of LWO. The routing step must hold the output tuples and release them only when the outer tuples (a_{11} and b_{12} in our example) completely scan the corresponding windows of the queries.

Figure 4 illustrates how the output tuples are released to the queries. In the figure, when the output tuple (a_{11}, b_8) is produced (Step 1), the routing step decides that tuple a_{11} completely scanned window w_1 and hence (a_{11}, b_8) can be released to query Q_1. We can also release (a_{11}, b_8) to queries Q_2 and Q_3 (Step 2). When the output tuple (a_9, b_{12}) is produced (Step 3), the routing step releases it to Q_1 since tuple b_{12} completely scanned window w_1 (Step 4). Note that (a_9, b_{12}) cannot be released to queries Q_2 and Q_3 as these two queries are waiting to receive the remaining output tuples that may result from joining a_{11} with their partial windows $w_2 - w_1$ and $w_3 - w_1$, respectively. When tuple (a_{11}, b_4) is produced (Step 5), it is released to both query Q_2 and Q_3 (Step 6). When tuple (a_5, b_{12}) is produced (Step 7), the tuples (a_9, b_{12}) and (a_5, b_{12}) are both released to query Q_2 (Step 8). In the same way, tuple (a_{11}, b_0) will be released to query Q_3 and tuple (a_1, b_{12}) (Step 11) will release the tuples (a_9, b_{12}), (a_5, b_{12}), (a_1, b_{12}) to query Q_3 (Step 12).

The SWF scheduling algorithm uses the following data structures:

- joinBuffers: joinBuffers represent main memory buffers used to store the tuples arriving from the input data streams. The size of a single JoinBuffer is limited by the maximum window size in the query mix.
- A list of queues for storing the tuples that need to be scheduled (or rescheduled). Each queue, $SchedulingQueue(w)$, represents one window (w), and contains the tuples waiting to be scheduled to join with w. The list of queues is ordered according to the size of the windows associated with each queue.
- An output buffer to hold the output tuples until they are ready to be released to the queries.

Given these structures, SWF can be described as follows:

[2]This equation can be obtained from Equation (1) by substituting S_{max} with S_i.

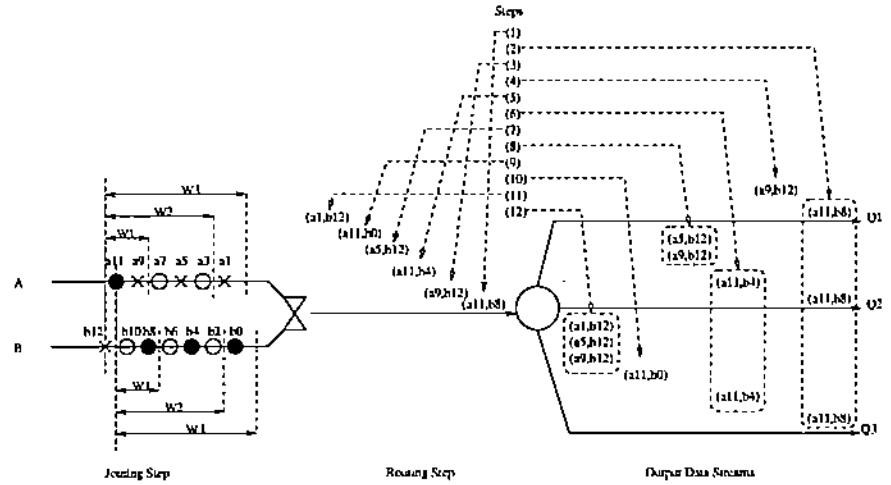


Figure 4: Scheduling the Shared Window Join using SWF.

1. Get a new tuple t (if exists) from any of the input data streams, say stream A. Store t in JoinBuffer(A).

2. If Step (1) results in a new tuple t, schedule the join of t with stream B using a window of smallest size and starting at the most recent tuple of B. Goto Step (4).

3. If Step (1) results in no tuples, get a tuple t from the list of *SchedulingQueues*. Assume that t belongs to stream A and is stored in SchedulingQueue(w_i). If no such tuple t exists, i.e., all the *SchedulingQueues* are empty, return to Step (1). Otherwise, schedule t for a join with stream B using window w_i (that corresponds to the queue SchedulingQueue(w_i)) and starting at the pointer location previously stored with t.

4. If the scheduled join of t results in output tuples, notify the router by sending the output tuples along with t to the routing step. Add t to the next queue, i.e., SchedulingQueue(w_{i+1}) in the list along with a pointer to stream B indicating where to restart next. Go to Step (1).

In Step (3), in order to maintain small joinBuffer sizes, the join step drops the old tuples in one stream that are outside the largest window. This process is performed dynamically while the join step is in progress. To retrieve a tuple from the list of SchedulingQueues, SWF finds the *first* nonempty queue (scanning smaller window queues to larger window queues) and retrieves the tuple at the head of the queue.

The routing step maintains a data structure, outputBuffer, to hold result tuples until they can be released. The join step sends the outer tuple along with the corresponding output tuples to the routing step (Step 4). Let the outer tuple be t, where t may either be a new tuple or an rescheduled tuple. In the first case, t is added to outputBuffer, and the output tuples are stored with t in outputBuffer but are also sent to all output data streams. In the second case, t is a rescheduled tuple from a scheduling queue, say SchedulingQueue(w_i). In this case, all the output tuples currently held for t along with the new output tuples are released to the queries with windows $\geq w_i$. If w_i is not the maximum window, the output tuples are added to the current outputBuffer of t. Otherwise, the entry for t is deleted from outputBuffer since t has been completely processed.

3.2.1 *Analysis of response time*

To estimate the average response time per query when using SWF, we use the same assumptions we outlined in Section 3.1.1. For a new arriving tuple, say outer tuple t in stream A, the resulting output tuples for a certain window w_i are only produced when t completely scans a window of size S_i in stream B. The average response time for window w_i can be estimated as the average waiting time of t until t joins completely with the window of size S_i.

For a query with window w_1, the first arriving tuple waits for time $S_1 t_p$, the second tuple waits for time $2S_1 t_p$ and the third tuple waits for time $3S_1 t_p$,... etc. The average waiting time for m tuples to scan window w_1 is: $\frac{m+1}{2} S_1 t_p$
For the second window, the waiting time for the first tuple is $mS_1 t_p + (S_2 - S_1)t_p$, for the second tuple is $mS_1 t_p + 2(S_2 - S_1)t_p$ and for the m^{th} tuple is $mS_1 t_p + m(S_2 - S_1)t_p$. Therefore, the average waiting time for the second window is: $mS_1 t_p + \frac{m+1}{2}(S_2 - S_1)t_p$
Generally, for window w_i, the average waiting time (also the average response time) can be computed as follows:

$$AvgRT(Q_i) = mS_{i-1}t_p + \frac{m+1}{2}(S_i - S_{i-1})t_p \qquad (2)$$

Figure 5 shows the response times for seven queries using the same setup as described in Section 3.1.1. Also, we plot the response time for the isolated execution of each query. The figure shows that the average response time for small queries is greatly reduced at the expense of the average response time for the larger queries. The performance of SWF is explored further in Section 5.

3.3 The Greedy Algorithm

In comparing the performance of SWF and LWO, it can be seen that SWF favors small window queries at the expense of larger window queries, whereas LWO favors larger window queries over smaller ones. This clear tradeoff between SWF and LWO motivates the development of our third scheduling

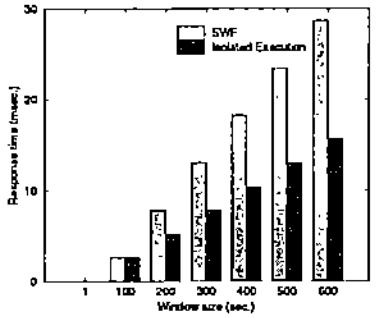


Figure 5: SWF versus Isolated Execution.

algorithm, which we call "Greedy". Intuitively, Greedy is more flexible than either LWO or SFW, choosing at any instant to process the tuples that are likely to result in the release of the most output tuples.

Recall that SWF suspends the processing of the join of a tuple with its next window whenever a newly arrived tuple needs to join with a smaller window. The suspended tuple, however, was supposed to scan a partial window (the difference between the window it had already scanned and the next larger window). This partial window could actually be smaller than the smallest full window. Allowing the old tuple to complete its current window before switching to the newly arrived tuple would result in an earlier release of a set of output tuples. This leads us to use a greedy approach to selecting the next tuple to schedule. With Greedy, the priority for scheduling is based on the amount of work that is expected to be accomplished by scheduling a tuple for a join.

We illustrate the Greedy algorithm by the example in Figure 6. In the figure, we have three queries with three different windows w_1, w_2, and w_3. We have omitted the details of the routing step from the figure. We choose w_1, w_2, and w_3 such that w_1 is larger in size than the difference $(w_2 - w_1)$ and the difference $(w_3 - w_2)$ is the largest. For illustration, we assume that the arriving tuple will join with all tuples in the other stream ($\alpha = 1$).

As illustrated by the output tuples from the join, tuple a_{11} joins for the small window w_1 and continues the join for the next window w_2 since the new tuple b_{12} is expected to scan more tuples in w_1 than a_{11} with $w_2 - w_1$. The Greedy scheduler will switch back to b_{12} when finishing a_{11} with w_2 since b_{12} is expected to scan smaller window in this case. Finally the Greedy scheduler will serve a_{11} with the partial window $w_3 - w_2$, followed by b_{12} with the partial window $w_3 - w_2$.

Join scheduling in the Greedy algorithm is determined by the sizes of the windows and their differences, which are static for a given set of window queries. As a result, a fixed priority can be assigned to the smallest window and to all the partial windows when a query is introduced. The priority is *inversely* proportional to the size of the smallest window or the partial window.

The steps for the Greedy algorithm are the same as those for the SWF algorithm, except in the priority assignment for the SchedulingQueues and in the routing step. In the Greedy algorithm, each queue in the list of the SchedulingQueues is assigned a priority based on the difference between the size of its window and the preceding (smaller) window. At each scheduling step, the tuple is selected from the SchedulingQueue with the maximum priority. Therefore, Step (2) in the SWF algorithm is modified to add the arriving tuple to the SchedulingQueue with the smallest window.

The selection of the tuple does not require a scan of the entire list of SchedulingQueues, since we can maintain a list to the SchedulingQueues ordered by priority, and traverse this list starting at the top priority SchedulingQueue to find first non-empty one. In the Greedy routing step, we can release the output tuples before the outer tuple completely scans the corresponding window, due to the static scheduling order. This means that in Figure 6, the output tuples (a_{11}, b_{12}) and (a_0, b_{12}) can be released to the query of window w_2, even before b_{12} completely scans w_2.

3.4 Hash-based Implementation

The algorithms in the previous sections assume a nested-loop implementation of the join step. A new arriving tuple t in one stream, say A, scans all the tuples in the other stream, say B, that lie within a certain window of t. The scanning starts from the most recent tuple of B and proceeds backwards. In the case of large window sizes, a linear scan of the tuples inside the window may be expensive. We describe a hash-based implementation of shared window-join using a symmetric hash join [16].

In a symmetric hash join, each stream has its own hash table. When a new tuple arrives in stream A, it is inserted into A's hash table and is used to probe the hash table for stream B.

We use a linked list to maintain a sliding window of the arriving tuples. The size of the sliding window is equal to the maximum window size in the shared queries. Tuples in the sliding window are ordered based on their arrival time. Whenever a new tuple arrives, it is added at the head of the linked list representing the sliding window. Tuples at the tail of the sliding window are dropped out of the hash table when probed by arriving tuples from the other stream.

Although the size of each bucket is relatively small, it is costly to scan the whole bucket (e.g., during LWO) to serve multiple window queries. However, based on a real implementation inside PREDATOR [13], we found that the cost of producing output tuples constitutes a major part in the cost of join processing in contrast to the cost of scans in either the nested-loop or the main-memory hash implementations of a single window join. The experiments demonstrate that the production of output tuples can be as high as 40% of execution time. Thus, scanning at the level of the bucket is still advantageous.

Two pointers, head and tail, are used to facilitate the insertion and deletion operations. In order to avoid scanning the sliding window to perform a join, we build a memory-based hash index on top of the sliding window linked list. Each

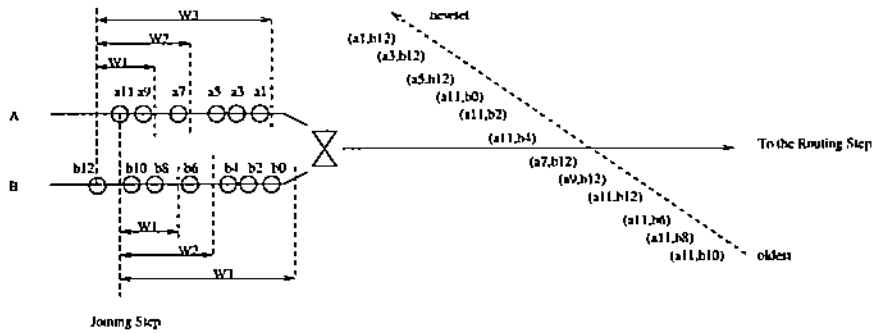


Figure 6: Scheduling the Shared Window Join using Greedy.

bucket in the hash table is implemented by a doubly linked list of tuples belonging to the bucket. This implies that we add two additional pointers for each tuple in the sliding window. Whenever a tuple drops at the tail of the sliding window, the doubly linked list allows us to delete this tuple from its corresponding bucket in constant time.

4. PROTOTYPE IMPLEMENTATION

In order to compare our three scheduling algorithms, we implemented them in a prototype database management system, PREDATOR [13], which we modified to accommodate stream processing. We implemented both hash-based and nested loop versions of the shared window join. Streaming is introduced using an abstract data type *stream-type* that can represent source data types with streaming capability. Stream-type provides the interfaces *InitStream, ReadStream, and CloseStream*. 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 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 to 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.

As the focus of this paper is on the operation of the shared join, we used the simple optimization already implemented in PREDATOR to generate the query plan for a new query. The execution plan consists of a single multi-way join operation at the bottom of the plan followed by selection and projection and (if present) the aggregate operator. Using this simple plan one can determine if the new query actually shared its join with other running queries or not. When adding a new query to the shared plan, the shared join operator creates a new output data stream (if the query uses a new window) or uses the output of an already existing data stream, which has the same window, as the input to the next query operators. For the case of SWF and Greedy, the shared window join operator creates a new SchedulingQueue if the query introduces a new window and updates the priority accordingly. The window specification is added as a special construct for the query syntax as was shown in the examples of Section 2.1.

5. EXPERIMENTS

All the experiments were run on a Sun Enterprise 450 , running Solaris 2.6 with 4GBytes main memory. The data used in the experiments are 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. The selectivity of a single tuple, S_i, is approximated as 0.002. The windows are defined in terms of time units (seconds).

In all experiments, we measure the average and maximum response time per output tuple as received by each query. In some cases we also report on the maximum amount of the main memory required during the lifetime of the experiment.

All the measurements represent steady state values (i.e., the window queries had been running for some time). As the maximum window could be large (e.g., 10 minutes), the experiments are "fast forwarded" by initially loading streams of data that extend back in time to the maximum window length. We collect performance metrics starting after this initial loading has been completed, and run the experiments until 100,000 new tuples are completely processed by the shared window join operator. The response times we report include both the cost of producing output tuples and the cost of the routing step.

5.1 Varying Window Distributions

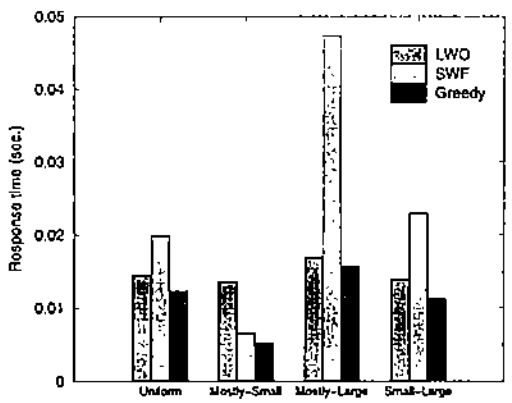
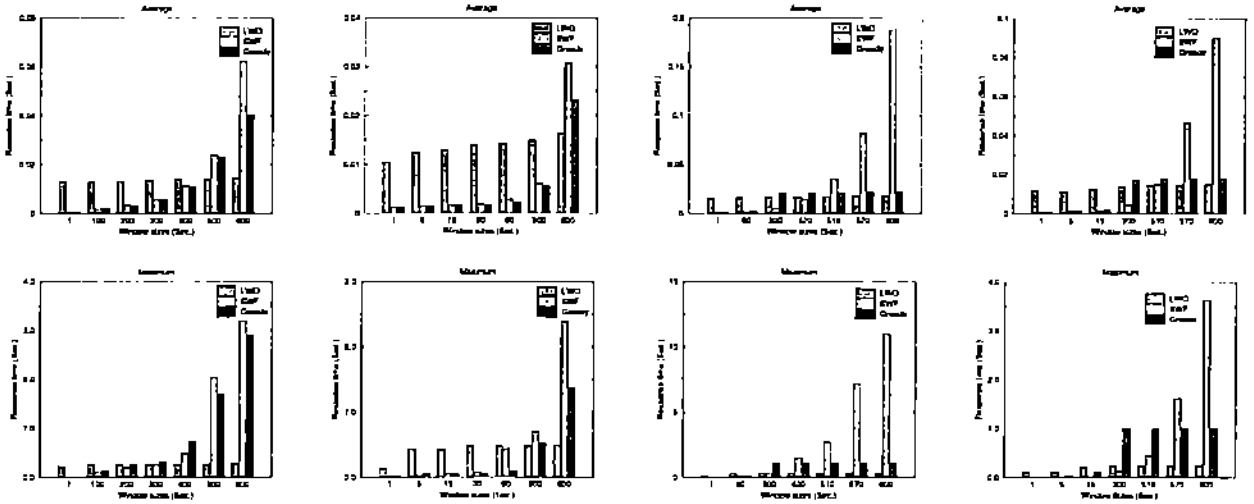


Figure 7: Average response time for all windows using different window distributions (hash-based).

In the first set of experiments we study the performance of our implementations of the LWO, SWF and Greedy algo-



(a) Uniform (b) Mostly-Small (c) Mostly-Large (d) Small-Large

Figure 8: Average and Maximum response time per window (hash-based).

Table 1: Window Sizes (in seconds)

Dist.	w_1	w_2	w_3	w_4	w_5	w_6	w_7
Uniform	1	100	200	300	400	500	600
Mostly − Small	1	5	15	30	60	300	600
Mostly − Large	1	60	300	420	510	570	600
Small − Large	1	5	15	300	510	570	600

rithms using four different window size distributions. We consider query workloads consisting of seven window-join queries with the same query signature, but each having a different window size. While we ran experiments on many different distributions and sizes, here we report on results using four representative distributions (shown in Table 1. All of these distributions include windows ranging in size from 1 second to 10 minutes (i.e., 600 seconds).

In the *Uniform* distribution, windows are evenly distributed in the range from 1 second and ten minutes. The *Mostly-Small* distribution has window sizes skewed towards the smaller range while the *Mostly-Large* has windows skewed towards the larger end of the range. Finally, the *Small-Large* distribution has windows skewed towards both extremes.

Note that the arrival rate is exponential with mean λ set to 100 tuples/sec for each data stream in these experiments. We examine the impact of more bursty arrival patterns in Section 5.2. Here, we first describe the results obtained using the hash-based implementation of the algorithms, and then briefly report on the results obtained using nested loops.

5.1.1 Hash-based Implementations

Figure 7 shows the output response time *per output tuple* averaged over all of the windows for each of the four distributions (the average and maximum response times are broken down per window for each of the distributions in Figure 8). As can be seen in the figure, Greedy has the best average response time of the three algorithms, while LWO provides the second-best response time for all of the distributions except for (as might be expected) *Mostly-Small*. LWO favors larger windows at the expense of smaller ones. Since these averaged numbers tend to emphasize performance of the larger windows, LWO's overall performance here is fairly stable (we will look at performance for each of the window sizes shortly). Even by this metric, however, LWO is consistently outperformed by Greedy. Comparing Greedy and SWF, the reasons that Greedy does well overall are twofold. First, recall that Greedy's scheduling always chooses to work on the smallest outstanding window or partial window, which compared to SWF can result in satisfying larger queries in a shorter amount of time. There are some cases, however, where Greedy and SWF generate effectively the same scheduling steps for new tuples. Even in these cases, however, Greedy has the advantages that because it can predict the next scheduling step for outer tuples, it can release the output of the largest window earlier than SWF can.

We now drill down on these results to examine the behavior of the scheduling algorithms for the different window sizes in the distribution. This breakdown is shown in Figure 8. The top row of graphs in the figure show the average response time for each window size; the bottom row of graphs show the maximum response time observed during the run of the experiment for each window size.

As can be seen in all of the figures, LWO's performance is relatively stable across window sizes for each workload. This is expected since since in LWO, new tuples that need to join with the smaller windows will have to wait until the largest window is completely processed by an older tuple. As a result, the output response time for all the windows is approximately equal to the response time for the largest window. The slight increase observable when comparing smaller windows to larger windows in LWO stems from the fact that if a tuple arrives when the system is idle, it can immediately start joining with previously arrived tuples. For such tuples, the joins for smaller windows are not delayed by joins for the larger windows. This behavior is predicted by the formulas derived in Section 3.1.1. There, equation (1) clearly

shows that the largest term in the equation is the second term, $(m-1)S_{max}$, which involves the largest window size, whereas only a small effect is expected do to the individual window size, (S_i+1).

In contrast to LWO, both SWF and Greedy tend to provide faster response times for smaller windows than for large ones. The performance of these two algorithms in this regard is in fact, heavily dependent on the window distribution, so we address their performance for each distribution individually, below. Before doing so, however, we note that the maximum response times provide by the algorithms (shown in the bottom row of Figure 8) generally follow the trends (on a per window size basis) observed for the average response time. The key fact to notice however, is that there can be substantial variance in the response time for individual output tuples; in some cases, the maximum response time is one or two orders of magnitude worse than the average.

Turning to the *Uniform* window distribution (Figure 8 (a)), we can see that in this case, Greedy and SWF provide similar performance for all but the largest window. This is because, here, they generate the same scheduling order for new tuples (recall that the difference in response time for the larger window is due to Greedy's ability to release tuples early for that window). Both algorithms favor the processing of smaller window queries with new tuples over resuming the join of older tuples with larger window queries. This is clear from the incremental increase in the SWF and the Greedy as we move from smallest to largest windows. Again, these results validate our analysis in Section 3.2.1 where equation (2) shows that the average response time depends on the current and previous window sizes which is incrementally increasing as we move from smaller to larger windows.

For the *Mostly − Small* window distribution,Figure 8 (b), one would expect a good scheduling algorithm to be SWF, given that most of the windows are small. As was seen in the *Uniform* case, however, Greedy performs much like SWF for the small windows here, and has an advantage for the largest window. Note, however, that SWF's response time for the largest window is half as much in this case than it is for the *Uniform* case. This behavior is predicted by the previous analysis equation (2), where the response time for a window includes the size of both the current window and the previous window. Since the two largest windows are further apart here than in the *Uniform* case SWF's response time decreases here.

For the *Mostly − Large* window distribution,Figure 8 (c), one would expect a good scheduling algorithm to be LWO, given that most of the windows are large. In fact, LWO does reasonably well here, outperforming Greedy for all of the windows except for the two smallest. Greedy does an effective job of balancing the performance for small and large windows, here, with the result that as shown earlier, it provides slightly better performance averaged over all windows, than does LWO in this case. SWF on the other hand, performs extremely poorly here, as it continually preempts processing of the (many) larger windows to handle new incoming tuples.

Finally, for the *Small − Large* windows distribution, Figure 8 (d), neither the LWO nor the SWF scheduling algorithms is the best choice, since choosing one will increase the response time for windows on one side of the windows spectrum. The Greedy scheduling is the best choice in this case as it behaves similar to the SWF for small window queries (windows 15 seconds and lower) and similar to the LWO for large window queries (windows 300 seconds and up).

5.1.2 *Nested loop implementation*

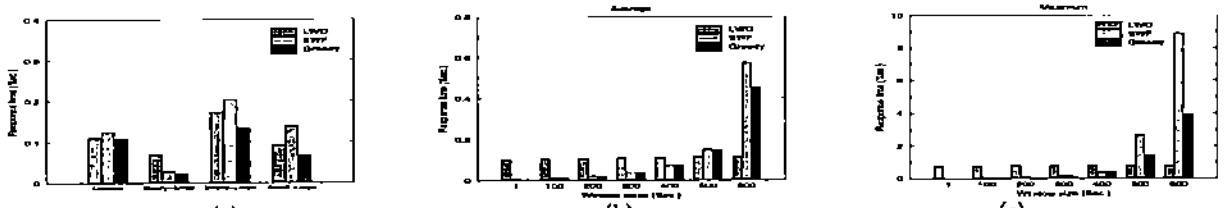
We also ran the above experiments using the nested loops implementations of the scheduling algorithms. In this case, due to the increased cost of join processing, we had to lower the arrival rate of the data streams to 15 tuples/sec in order to ensure that all algorithms could keep up with the incoming streams without dropping any tuples.

The average response time for the different window distributions (averaged over all window sizes) is shown in Figure 9(a). We also show the per window average and maximum response times for the *Uniform* distribution in Figures 9(b),(c) respectively. The results here clearly resemble those obtained with the hash-based implementations, with the obvious difference in magnitude of the response times. Compared to hashing, the response time for the nested loop algorithms is higher for all three scheduling approaches, as incoming tuples must be compared with more tuples from the other stream. Due to space constraints, we do not show the detailed results for the other window distributions using nested loops, as the story there is similar.

The conclusion of the previous experiments in Section 5.1.1 and Section 5.1.2 is that the Greedy algorithm provides the best overall average response time when compared to the LWO and the SWF and when using a variety of window distributions. For the SWF and the LWO algorithms there is no clear winner as their relative performance is highly dependent on the particular window distribution. In terms of maximum response time, the Greedy algorithm is always better than the SWF algorithm for large windows, although it has some irregularity for middle windows. This irregularity is mainly the result of switching back and forth to serve small as well as large windows. The LWO algorithm has a uniform maximum value over all the windows due to the fixed scheduling order used by LWO.

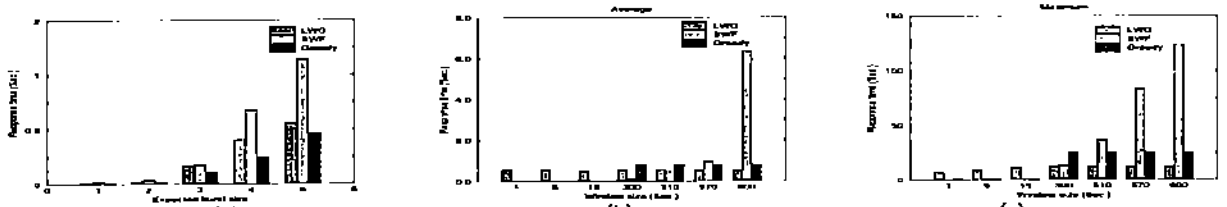
5.2 Varying the level of burstness

In the previous experiments, tuple arrival rates were driven by an exponential distribution. The analyses of the algorithms in Section 3 showed that their performance is highly dependent on the *burstiness* of the arrival pattern. To examine this issue more closely, we ran several experiments studying the behavior of the three algorithms as the level of burstess (i.e., tendency of tuples to arrive within short period of time) is increased for both streams. In these experiments, we generate the burst arrival of data streams using a pareto [8] distribution, which is often used to simulate network traffic where packets are sent according to ON OFF periods. In the ON periods a burst of data is generated and in the OFF periods no data is produced. The interval between the ON and OFF periods is generated using the exponential distribution with rate λ. The density function of pareto distribution is, $P(x) = \frac{\alpha b^{\alpha}}{x^{\alpha+1}}$, where $b \geq x$ and α



(a) (b) (c)

Figure 9: Response time with nested loop implementation.



(a) (b) (c)

Figure 10: Response time for different burst arrival sizes.

is the shape parameter. The expected burst count, $E(x)$, is $\frac{\alpha b}{\alpha - 1}$. For α much larger than 1, the expected value is almost one, and for α between 2 and 1, the expected value increases. We vary the expected burst size, $E(x)$ between one and five (and choose α accordingly). We also, modify the arrival rate between the ON periods, to provide a fixed overall average rate of, $\frac{\lambda}{E(x)}$. As we increase the level of burstness, more tuples wait to schedule their join with the other stream.

In this section we report on an experiment using the hash-based implementation for the three scheduling algorithms and considering the *Small − Large* windows distribution. The overall arrival rate is maintained at 100 tuples/sec per stream. Figure 10(a) shows the average response time (averaged over all window sizes). In the figure, we can see that as the burst size increases, the scheduling becomes more important; as bad scheduling decisions can increase the overall average response time dramatically. The Greedy scheduling outperforms all other scheduling for all of the burst sizes here. This behavior is more evident as we increase the expected burst size (e.g, at expected burst of sizes four and five, respectively). As shown in the Figure, the improvement of Greedy over LWO is as high as 80% (in the case of burst size of four).

Figures 10(b) and (c) show the average and maximum response time (respectively) per window for the *Small−Large* distribution using an expected burst size of five. Figure 10(c) indicates that with large burst sizes, the SWF scheduling algorithm has a response time of 125 seconds (approximately 2 minutes) for the largest window, whereas using the Greedy scheduling algorithm it is bounded by 25 seconds. These results demonstrate the fact that efficient scheduling is important to maintain a reasonable response time, particularly in unpredictable environments.

5.3 Memory Requirements

One concern about the SWF and Greedy approaches is that they might use excessive memory compared to LWO, due to their need to hold back some output tuples to preserve the proper ordering of the output stream. In order to determine the impact of this issue, we examined the maximum amount of memory required by each of the algorithms. While small differences are not likely to be important (given the low cost of memory these days), a large difference could have a negative impact on the data rates that could be supported by the various algorithms with out dropping tuples, for a particular memory size. We briefly present our experimental findings here.

For all of the scheduling algorithms, the JoinBuffer is needed to hold the tuples of each stream during the join processing. The maximum size of a single JoinBuffer is λw_{max}. SWF and Greedy also use an extra input buffer (a list of scheduling queues) to hold the new tuples from one stream until they complete their join with the other stream. LWO has a similar input buffer (a queue) to store the arriving tuples from one stream before they are actually used to scan the maximum window in the other stream. SWF and Greedy algorithms maintain an output buffer to sort the output before releasing it to the output data streams. The maximum size of the input buffer is a function of the maximum response time for a newly arriving tuple. In the experiments we reported on above the maximum response time was seen to reach 2 minutes in some cases (see Figure 10(c)).

When considering an arrival rate of 100 tuples/sec and a maximum window of size 600 seconds, the size of the JoinBuffer is approximately 60,000 tuples and the maximum input buffer size is 12,000 tuples, or 20% of the JoinBuffer size. This, however, is a worst case analysis. We experimentally obtained lower bounds for the maximum input buffer size, and found them to be less than 10% of the JoinBuffer size, for SWF.

Our conclusions are that the memory requirement for the SWF and the Greedy scheduling algorithms are roughly comparable to that of the LWO algorithm. We also measured the maximum size for the output buffer in case of the SWF and the Greedy algorithms. The maximum size for the output buffer in both the SWF and the Greedy algorithm was less than 3% the size of the JoinBuffer. This supports our conclusion that the memory requirement for the extra

input and output buffers in the SWF and the Greedy algorithms are negligible when compared to the JoinBuffer sizes.

6. RELATED WORK

Stream query processing has been addressed by many evolving systems such as Aurora [4], Telegraph [12] and STREAM [3] systems. The shared execution of multiple queries over data streams is recently presented in CACQ [12] and NiagaraCQ [7, 6]. In CACQ, they addressed the shared window join between multiple queries by using the largest window, similar to our first proposed algorithm. The join operation is implemented as a multi-way join and the window is defined in terms of number of tuples. CACQ also addressed the shared execution of selection queries (predicate indexing). The alteration of the query plan is dynamic (using Eddies [2]). Our research in this paper focuses on the shared window join and we provide several alternatives to schedule the join than the work in CACQ. The work in NiagaraCQ addressed incremental group optimization of multiple queries where queries are added and deleted to the system. The underlying data sources are remotely located (over the web). Their conclusion of pulling up the selection predicate to the top of the group execution plan, supports the use of shared join as we address in this paper.

Window join processing has been addressed in [5, 11]. Psoup [5] handles streamed queries over streaming data and provides a similar definition to ours for sliding time window. However, Psoup uses the same approach as was suggested in CACQ, mainly join using the largest window which is a limited approach as we showed in this paper. The recent work in [11] addressed the window join over two streams where the two arriving streams have different arrival rates. The authors suggest using asymmetric join (e.g., building a nested loop one stream and a hash table on the other stream), to reduce the execution cost. Our research is different as we consider the problem of sharing the window join execution between multiple queries.

Joining data streams is also addressed in [1], the authors identifies some queries over data streams that can be executed entirely using limited amount of memory. The authors consider the boundness of the terms in the projection and selection predicates to determine if the query can be executed in limited memory or not.

Scheduling a single join processing over non streaming data had been studied in [9, 10, 14, 15]. The ripple join switches from processing one input data stream to the other when the input source is blocked. The work in [10, 14, 15] address the scheduling of the hash join with limited memory (swapping part of the hash table to disk). The join scheduling is similar in spirit to our proposed research although the previous work was different as in addressing single join and non window processing.

7. CONCLUSIONS

Sharing the execution of window joins among multiple queries is a technique that can enhance scalability in stream query processing. We studied three scheduling algorithms (LWO and the two new algorithms SWF and Greedy) that prioritize this shared execution to minimize the average response time per query. The tradeoffs between LWO and SWF motivated the development of the Greedy algorithm. Under a variety of workloads and mixes of window sizes, the Greedy algorithm provides up to 80% improvement in average response time over the other two algorithms. The experimental results based on real implementation of the algorithms validate our analytical results. The experiments also illustrate that the gain in average response time of Greedy is at the expense of an additional memory overhead of less than 10%.

8. 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, May.*, 2002.

[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] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *28th VLDB Conference, Aug.*, 2002.

[5] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *28th VLDB Conference, Aug.*, 2002.

[6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE, Feb.*, 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] M. E. Crovella, M. S. Taqqu, and A. Bestavros. Heavy-tailed probability distributions in the world wide web. In *A practical guide to heavy tails: statistical techniques and applications, chapter 1, Chapman & Hall, New York, pp. 3–26.*, 1998.

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

[10] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. of the SIGMOD Conference.*

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

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

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

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

[15] T. Urhan and M. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of 27th VLDB Conference, September*, 2001.

[16] 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.