Scheduling for shared window joins over data streams

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

Michael J. Franklin§ UC Berkeley franklin@cs.berkeley.edu

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

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

Abstract

Continuous Query (CQ) systems typically exploit commonality among query expressions to achieve improved efficiency through shared processing. Recently proposed CQ systems have introduced window specifications in order to support unbounded data streams. There has been, however, little investigation of sharing for windowed query operators. In this paper, we address the shared execution of windowed joins, a core operator for CQ systems. We show that the strategy used in systems to date has a previously unreported performance flaw that can negatively impact queries with relatively small windows. We then propose two new execution strategies for shared joins. We evaluate the alternatives using both analytical models and implementation in a DBMS. The results show that one strategy, called MQT, provides the best performance over a range of workload settings.

1 Introduction

In many emerging applications, particularly in pervasive computing and sensor-based environments, *data streams* play a central role as 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, network monitoring, and enterprise application integration. Data streams break a number of the assumptions upon which traditional query processing technology is built, and thus, they require a rethinking of many fundamental database management techniques.

* The authors' work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, IIS-0209120, and EIA-9983249.

§ Michael Franklin's work was supported in part by the National Science Foundation under ITR grant IIS-0086057.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

One major difference that arises in data stream processing (compared to more traditional stored database management) is the notion of long-running Continuous Queries (CQs) 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 collection of standing queries raises the potential for aggressively sharing processing needed by multiple queries. Furthermore, the high data rates and tight responsiveness constraints in many streaming applications require that such opportunities for efficiency be exploited.

In this paper, we focus on a fundamental problem that arises in CQ processing over data streams. Namely, we investigate the problem of scheduling multiple *windowed joins* over a common set of data streams. As in traditional query processing systems, join is a fundamental operator. In streaming systems, joins allow data from multiple streams (e.g., different sensors or probes) to be combined for further processing, aggregation, and analysis. The role of joins in emerging CQ systems is further enhanced, however, due to the use of "selection pull up". As demonstrated in the NiagaraCQ system [4], the traditional heuristic of pushing selection predicates below joins is often inappropriate for CQ systems because early selection destroys the ability to share subsequent join processing. Given the high cost of joins (relative to selections), it is often most efficient to process a join once and then send those results to the selection operators. Similar arguments also hold for Group By and aggregation operators. Thus, systems like NiagaraCQ push joins down in query plans and process joins with a common signature (i.e.,the same input relations and join predicates) using just a single instance of that join. As a result, shared joins are often at the very core of query plans in CQ systems.

Emerging data stream processing systems add another component to the problem. Since data streams are typically assumed to be unbounded, traditional join operators would need to maintain unbounded state. As a result, query languages for data stream systems typically include a windowing predicate that limits the scope of the operators. The window size varies according to the semantics of each query. A naive approach would treat identical joins with different window constraints as having different signatures and would execute them separately, thereby negating the benefits of selection pull up and performing redundant join processing. Recent systems, such as CACQ [10] and PSoup [3], have sought to avoid this problem, and indeed do share processing of joins with identical signatures. As we discuss later, however, those systems adopted a scheduling strategy for such joins that discriminates against queries with small windows. Ironically, it is exactly such queries that are likely to have strict responsiveness constraints.

In this paper we make several contributions:

- We formulate the problem of shared window join processing and identify an important property than can be exploited in scheduling them.
- We present two initial scheduling algorithms for such processing that favor either small or large windows, and evaluate them analytically.
- Based on insights from this analysis, we develop a new algorithm called Maximum Query Throughput (MQT) that can work well across a range of window sizes.
- We describe our implementation of the three approaches in an existing DBMS and present results from a detailed performance study of the implementations.

The rest of the paper is organized as follows. Section 2 presents the model of window join, the problem definition and the related work. Section 3 describes the proposed scheduling algorithms. Sections 4 and 5 present the prototype implementations and the experimental results. Section 6 contains the concluding remarks.

2 Preliminaries

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). Streams are considered to be unbounded sequences of data items. Each data item in a stream is associated with a timestamp that identifies the time at which the data item enters the system. The data items of a single stream may arrive in a bursty fashion (i.e., a group of data items arriving within a short period of time) or they may arrive at regularly-spaced intervals. Examples of bursty streams include network monitoring streams, phone call records, and event-driven sensors. In contrast, pull-based sensors driven by periodic polling would produce a regular stream. Our discussion here focuses on bursty streams.

Queries over streams often exploit the temporal aspects 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 [13]. 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, Q_1, that continuously monitors the count of locations that have both humidity and temperature values above a specific thresholds within a one-minute interval could be specified as follows:

```
SELECT COUNT(DISTINCT A.LocationId))
FROM Temperature A, Humidity B
WHERE A.LocationId = B.LocationId and
A.Value > $Threshold_t$ and B.Value > $Threshold_h$
WINDOW 1 min;
```

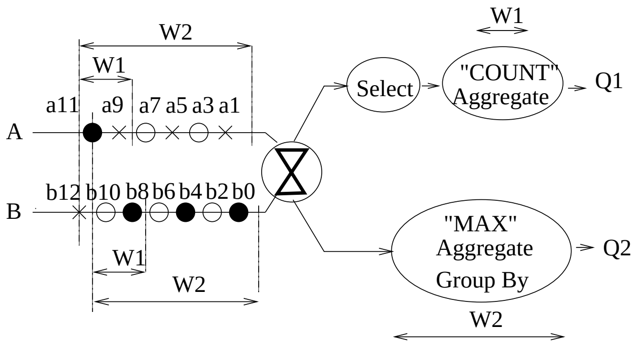
A second example query, Q_2, continuously reports the maximum temperature and humidity values per location in the last one hour interval as follows:

```
SELECT A.LocationId, MAX(A.Value), MAX(B.Value)
FROM Temperature A, Humidity B
WHERE A.LocationId = B.LocationId
GROUP BY A.LocationId
WINDOW 1 hour;
```

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. Note that the two example queries contain an equijoin with a common signature, but have significantly different window sizes (one minute and one hour).

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 [9] or with each pair of streams in a multi-way join [7]. 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; the algorithms can be applied to windows defined in terms of tuple counts in the same way.

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. Recently proposed stream query processing system, Aurora [1], proposes mechanisms to respond



Shared Window Join

Figure 1: The shared execution of two window joins.

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 completely. 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.

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 concurrent queries with the same signature (i.e., 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 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. The syntax of Q_1 and Q_2 were described in Section 2.1. 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., star, 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 minute. Q_2 performs a join between the same two streams A and B, using predicate p with window size w_2 = one hour. There is an obvious overlap between the interests of both queries, namely, the answer of the join for Q_1 (the smaller window) is included in the answer of the join for Q_2 (the larger window). We refer to this as the *containment property* for the join operation; that is, the join answer of any query is also contained in the join answer of the queries having the same signature with larger windows.

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

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 avoids such redundant processing.

The shared join produces multiple output data streams for each separate query. The output data streams are identified by their associated window sizes, and at least one query must be attached to each output data stream. The shared join operator is divided into two main parts: the join part and the routing part. The join part produces a single output stream for all queries and the routing part 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. 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.

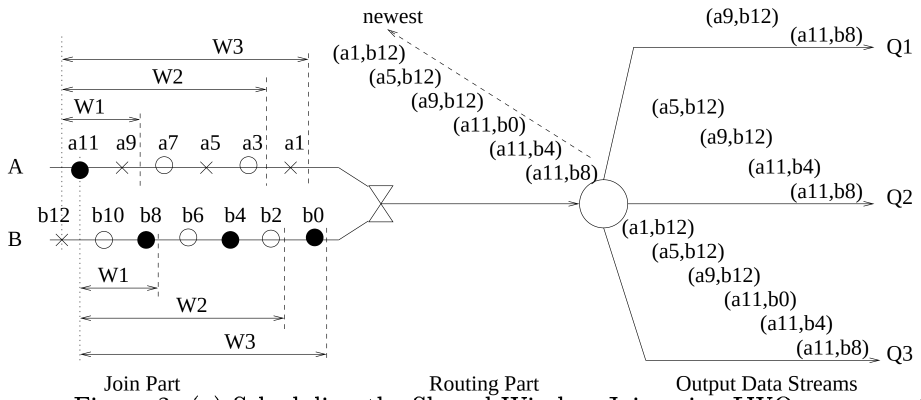
2. The response time penalty imposed on any query when a new query is included in a shared plan should be kept to a minimum.

In our example queries, changing the order of the output during shared execution (a violation of objective 1 above) could potentially produce different COUNT and MAX results than isolated execution. In addition, when the shared execution imposes a high response time penalty for one query (e.g., Q_1), that query's output could be significantly delayed. As we show in Section 3, the average response time per tuple for small window queries could increase from milliseconds (in isolated execution) to seconds, delaying crucial notifications, for example, that many sensors in some part of the data center are reporting a spike in their temperature and humidity values.

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

2.3 Related Work

Stream query processing has been addressed by many evolving systems such as Aurora [1], Telegraph [2] and STREAM [11] systems. The shared execution of multiple queries over data streams is recently presented in CACQ [10] and PSoup [3]. Both CACQ and PSoup address the shared window join among multiple queries by using the largest window, similar to our first proposed algorithm. Our research in this paper focuses on the shared



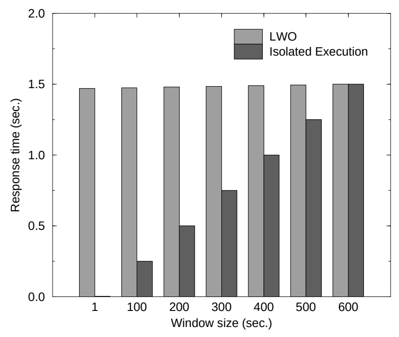


Figure 2: (a) Scheduling the Shared Window Join using LWO. (b) LWO versus Isolated Execution.

window join and we provide several alternatives to schedule the join beyond that used in CACQ and PSoup.

The recent work in [9] addresses 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 on 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 among multiple queries.

Scheduling the processing of a single join over non streaming data had been studied in [6, 8, 14]. Although, similar in spirit to the research we propose in this paper, scheduling individual joins does not address the issues raised by sharing and by window-based processing.

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 Maximum Query Throughput (MQT). LWO was implicitly used, but not elaborated upon in [3, 10]. 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 MQT algorithms are contributions of this paper.

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 (i.e., most recent) timestamp of the two tuples that form the join tuple.

All three scheduling algorithms presented in this section abide by the isolated execution property (Section 2.2). 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 other stream, from most to least recent arrival, or can be done using *hashing* as described in Section 3.4. 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 part 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 part 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 given in Figure 2(a). The figure shows a shared 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 part uses the largest window (w_3). 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 part. The routing part 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 part 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 sin A is checked against S_i tuples in B. Let $AT(a)$ and $CT(a)$ 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,

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

where the sum is taken over all output join tuples and $joinTuple_k$ corresponds to the tuple (a, b). Since $joinTuple_k$ is an output tuple of window w_i, then, $|AT(a) - AT(b)| < w_i$ and $AT(joinTuple_k) = max\{AT(a), AT(b)\}$. $CT(joinTuple_k)$ represents the time at which the output tuple is received by Q_i. For simplicity of the analysis, let $\alpha = 1$ (other values of α will not affect the analysis as the average is taken over all the output).

Let t_p be the time needed to check that a tuple pair, say (a, b), satisfies the join predicate and the window constraint $|AT(a) - AT(b)| < w_i$. Then, for window w_i, the first tuple of the m tuples will produce S_i output tuples with a total delay of $t_p + 2t_p + \ldots + S_i t_p$ 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,

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

To clarify this equation we plot the AvgRT for multiple queries while using the following values: $t_p = 1$ usec, $\lambda =$ 100 Tuples/Sec, m = 50 tuples. The windows are chosen to span a wide range (from 1 second to 10 minutes). Figure 2(b) 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[2] is AvgRT(Q_i) $= \frac{t_p}{2}(mS_i + 1)$. It is clear from the graph that the query with smallest window, i.e., Q_1 (with $w_1 = 1$ *sec.*) 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.

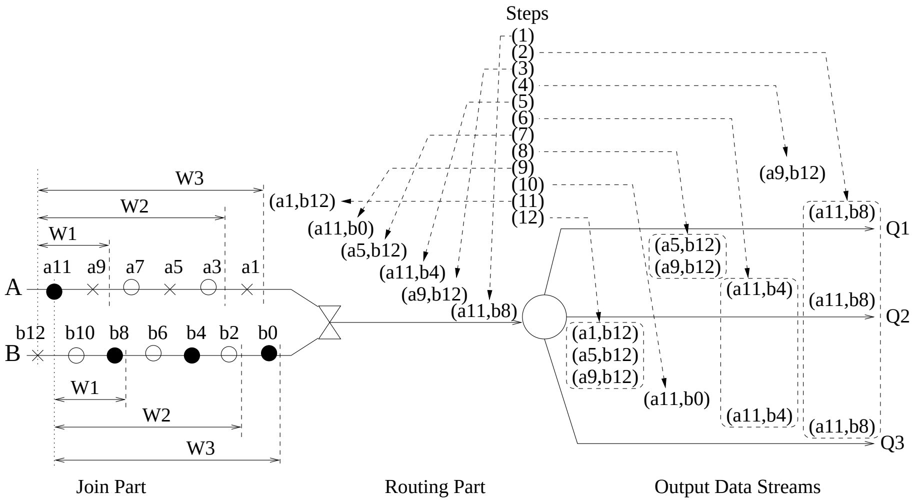
These analytical results are validated by experiments on an implementation of the algorithm in Section 5.1.

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. Under our basic assumption (Section 2.1) that the system can process all queries completely, tuples will eventually proceed to join with larger windows.

We illustrate SWF with the example in Figure 3(a). 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. Since tuple b_{12} will join window w_1 (the smallest window), b_{12} is scheduled immediately.

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



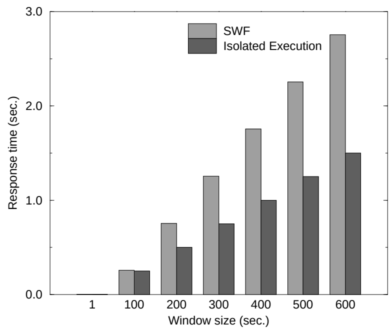


Figure 3: (a) Scheduling the Shared Window Join using SWF. (b) SWF versus Isolated Execution.

Tuple a_{11} has not finished its join with stream B so it is stored along with a pointer to tuple b_6. b_{12} now 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 in stream A. The process continues until tuple b_{12} joins with the partial window $w_3 - w_2$, of stream B. Figure 3(a) 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 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 it completes the new window.

Note that the output of the join part is *shuffled* compared to that of 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 part from that of LWO. The routing part must hold the output tuples and release them only when the outer tuples (a_{11} and b_{12} in our example) have completely scanned the corresponding windows.

Figure 3(a) 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 part 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 part 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).

Having described the operation of SWF, we now present the data structures and the detailed steps for the join and routing parts of the algorithm. For the join part, SWF scheduling algorithm uses the following data structures:

- joinBuffers, one for each input stream: joinBuffers are 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.

Given these structures, the join part of SWF can be described as follows:

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 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 part. 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 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. Also, in order to keep joinBuffer sizes small, the join part drops the old tuples in one stream that are outside the largest window. This process of tuple dropping is performed dynamically while the join is in progress.

The routing part of SWF is implemented as follows: A data structure, called the outputBuffer, is used to hold result tuples until they can be released. Step 4 of the join part sends the outer tuple along with the corresponding output tuples to the routing part. 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.

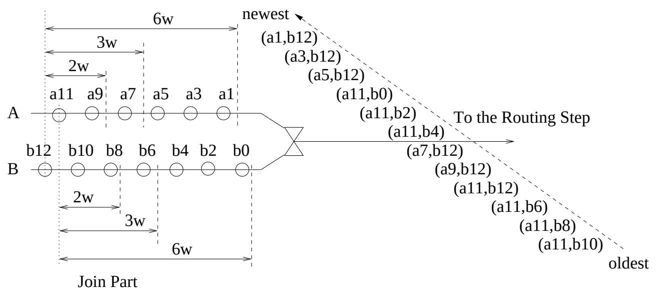
3.2.1 Analysis of Response Time

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:

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

Figure 3(b) 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.



$MaxQT(PW(w_i, w_j))$

	w_1	w_2	w_3
0	$\frac{1}{2w}$	$\frac{2}{3w}$ *	$\frac{2}{3w}$
w_1	-	$\frac{1}{w}$	$\frac{1}{w}$
w_2	-	-	$\frac{1}{3w}$

* $MaxQT(PW(0, w_2)) = max\{\frac{C_{01}}{pw_{01}}, \frac{C_{02}}{pw_{02}}\} = max\{\frac{1}{2w}, \frac{2}{3w}\} = \frac{2}{3w}$

Figure 4: Scheduling the Shared Window Join using MQT.

3.3 Maximum Query Throughput (MQT)

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 algorithm, which we call "MQT" for Maximum Query Throughput. Intuitively, MQT is more flexible than either LWO or SWF, choosing at any instant to process the tuple that is likely to serve the maximum number of queries per unit time (query throughput).

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 partial windows (the difference between the window it had already scanned and the next larger windows). Scanning one of these partial windows could actually serve more queries in less time than scanning the smallest full window. This leads us to consider a new selection criteria for scheduling tuples in the shared window-join. We present the following definitions before describing the basis of this selection.

Assume that the windows $w_1 \cdots w_N$ are sorted in increasing order of their size. Assume further that the tuple t_{old} arrives followed by the tuple t_{new}. t_{old} scans the windows from smallest to largest starting at w_1. The same is true for t_{new} when it arrives. Let window w_j be a window already completely scanned by t_{old}, and w_i be a window already completely scanned by t_{new} at a given point in time. Notice that it must always be true that $i \leq j$, otherwise, output tuples will not be emitted in the right order. So, the set of possible (partial) windows that the scheduler can assign to t_{new} depends only on i and j and is defined as: $PW(w_i, w_j) = \{pw_{ik} | pw_{ik} = w_k - w_i$, for $i+1 \leq k \leq j\}$. Two special sets of $PW(w_i, w_j)$ are $PW(0, w_j)$ (i.e., tuple t_{new} has not scanned any window) and $PW(w_i, \infty)$ (i.e., tuple t_{new} has no preceding tuple, in this case $PW(w_i, \infty) = PW(w_i, w_N)$, where N is the number of distinct windows).

Let C_i be the count of queries that will be serviced while scanning window w_i. $C_i = \sum_{l=1}^{i} Queries(w_l)$, where $Queries(w_l)$ is the number of queries with window w_l. The count of queries C_{ij} that will be serviced exclusively by scanning partial window pw_{ij} is $C_{ij} = C_j - C_i$. The time to scan a (partial) window is proportional to the size of the (partial) window. Therefore, we can use the ratio $\frac{C_{ij}}{pw_{ij}}$ as an estimate of query throughput for scanning the partial window pw_{ij}. The maximum query throughput for a tuple t if the tuple is allowed to scan any of its $pw_{ij} \in PW(w_i, w_j)$ is defined as: $MaxQT(PW(w_i, w_j)) = max\{\frac{C_{ij}}{pw_{ij}} | pw_{ij} \in PW(w_i, w_j)\}$.

The MQT algorithm schedules the tuple with the maximum value of $MaxQT(PW(w_i, w_j))$ among all the waiting tuples. $MaxQT(PW(w_i, w_j))$ depends on the relative order between two windows. Therefore the value of $MaxQT(PW(w_i, w_j))$ can be calculated and stored in a two dimensional matrix of fixed size N^2, where N is the number of distinct windows. The matrix changes only when a new query is added or an old query is removed from the shared window join.

We illustrate MQT by the example in Figure 4. In the figure, we have three queries with three different windows of sizes $w_1 = 2w, w_2 = 3w$, and $w_3 = 6w$, respectively. For illustration purposes, we assume that the arriving tuple will join with all tuples in the other stream ($\alpha = 1$). We also present in Figure 4 the $MaxQT(PW(w_i, w_j))$ matrix for the given windows' setting along with the detailed derivation for the value in entry $MaxQT(PW(0, w_2))$.

As shown in the figure, tuple a_{11} joins for the small window $2w$ and continues the join for the next window $3w$ since after finishing w_1, $MaxQT(PW(w_1, w_3))$ for a_{11} is larger than $MaxQT(PW(0, w_1))$ for b_{12}. MQT will switch back to b_{12} when finishing a_{11} with w_2 since by this time the $MaxQT(PW(w_2, w_3))$ for a_{11} is less than the $MaxQT(PW(0, w_2))$ for b_{12}. Finally, MQT will serve a_{11} with the partial window $w_3 - w_2$, followed by b_{12} with the partial window $w_3 - w_2$.

The steps for MQT are the same as those for SWF, except in selecting a tuple from the SchedulingQueues and in the routing part. In MQT, we traverse the list of the SchedulingQueues from largest to smallest windows. We choose the non-empty SchedulingQueue at window w_i that has the largest $MaxQT(PW(w_i, w_j))$ among all non-empty SchedulingQueues, where w_j is the window of the previous non-empty SchedulingQueue ($w_j = \infty$ if no such SchedulingQueue exists). Therefore, Step (2) in the SWF algorithm is deleted and Step (3) is modified to traverse the SchedulingQueues list searching for a tuple with the largest $MaxQT(PW(w_i, w_j))$. In the routing part for MQT, we can release the output tuples before the outer tuple completely scans the corresponding window. This means that in Figure 4, the output tuples (a_{11}, b_{12}) and (a_9, 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 were described assuming a nested-loop implementation of the join part. Here, we briefly describe a hash-based implementation of shared window-join using a symmetric hash join [15].

Each tuple in the hash table is a member of two linked lists. The first linked list includes all tuples in the hash table. The order of tuples in the list represents the arrival order of tuples in the sliding window. The second linked list includes the tuples that belong to the same hash bucket (have the same hash value). The size of the first linked list is equal to the maximum window size of the shared queries. Whenever a new tuple arrives, it is added at the head of the first linked list. In addition, the new tuple is linked to the list corresponding to its hash bucket. Tuples at the tail of the sliding window are dropped from the hash table when it is probed by arriving tuples from the other stream. The sliding window hash table structure requires two extra pointers per each tuple (compared to the traditional in-memory hash table) to maintain the first linked list. However, the sliding window hash table provides great flexibility in dropping expired tuples, even if they do not belong to the probed hash bucket. Therefore, the size of the hash table always reflects the size of the largest window.

Although the size of each hash bucket is relatively small, it is costly to scan the whole bucket (e.g., during LWO) to serve multiple window queries. However, based on our implementation inside PREDATOR [12], we found that the cost of producing output tuples constitutes a major part of the cost of join processing. Our experiments show that the production of output tuples can be as high as 40% of execution time. Thus, scheduling of tuples at the bucket level is still advantageous.

4 Prototype Implementation

In order to compare our three scheduling algorithms, we implemented them in a prototype database management system, PREDATOR [12], 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. To interface the query execution plan to the underlying stream, we introduce a *StreamScan operator* to communicate with the stream table 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 with the same window as the input to the next query operators. For the case of SWF and MQT, the shared window join operator creates a new SchedulingQueue if the query introduces a new window and updates the matrix in the case of MQT. 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, α, 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 part.

5.1 Varying Window Distributions

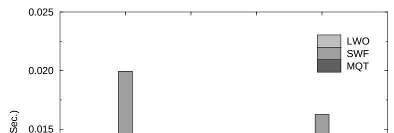


Figure 5: Average response time for all windows using different window distributions (hash-based).

In the first set of experiments we study the performance of our implementations of the LWO, SWF and MQT algorithms 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).

Table 1: Window Sizes (in seconds)

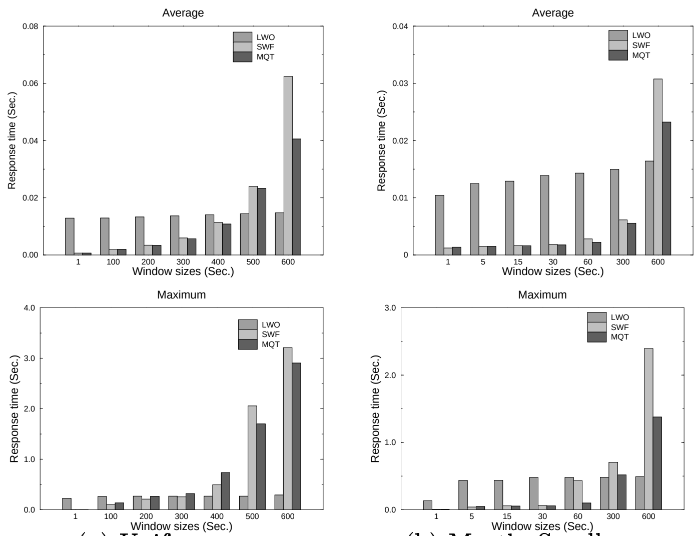
Dist.	w_1	w_2	w_3	w_4	w_5	w_6	w_7
Uniform	*1*	*100*	*200*	*300*	*400*	*500*	*600*
Mostly-Small	*1*	*5*	*15*	*30*	*60*	*300*	*600*
Mostly-Large	*1*	*60*	*300*	*420*	*510*	*570*	*600*
Small-Large	*1*	*5*	*15*	*300*	*510*	*570*	*600*

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 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. We briefly report on the results obtained using nested loops afterwards.

Hash-based Implementations
Figure 5 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 the first two distributions in Figure 6). The arrival rate, λ, is set to 100 tuples/sec. As can be seen in the figure, MQT 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 MQT. Comparing MQT and SWF, the reasons that MQT does well overall are twofold. First, recall that MQT'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 MQT and SWF generate effectively the same scheduling steps for new tuples. Even in these cases, however, MQT has the advantage 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 win-



(a) Uniform (b) Mostly-Small

Figure 6: Response time with hash-based implementation.

dow sizes in the distribution. This breakdown is shown in Figure 6. 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 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 due to the individual window size, $(S_i + 1)$.

In contrast to LWO, both SWF and MQT 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 the first two distributions, below. Before doing so, however, we note that the maximum response times provided by the algorithms (shown in the bottom row of Figure 6) 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 6 (a)), we can see that in this case, MQT 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 MQT'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 MQT 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 6 (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, MQT 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.

The conclusion of the previous experiments is that the MQT 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 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 MQT 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.

Nested Loop Implementations

We also repeated the experiment for measuring the output response time using different window size distributions, however 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.

The average response time for the different window distributions resemble those obtained with the hash-based implementations, with approximately order of magnitude increase than the values reported in Figure 5 and Figure 6. Due to space constraints, we omit the detailed performance figures.

5.2 Varying the Level of Burstiness

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 burstiness (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 [5] 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 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 burstiness, more tuples wait to schedule their join with the other stream.

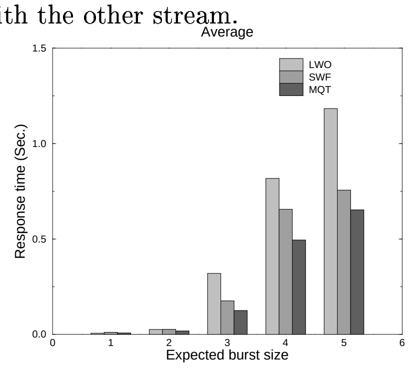


Figure 7: Response time for different burst arrival sizes.

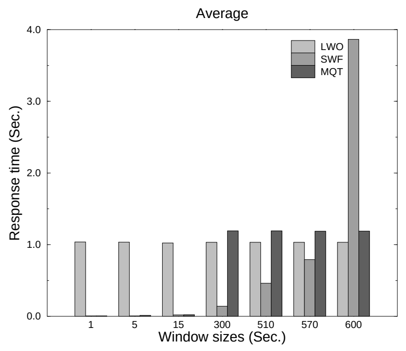


Figure 8: Response time per window size and for burst size equals five.

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 7 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 MQT 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 three, four and five, respectively). As shown in the Figure, the improvement of MQT over LWO and SWF is as high as 60% and 30%, respectively, (in the case of burst size of three).

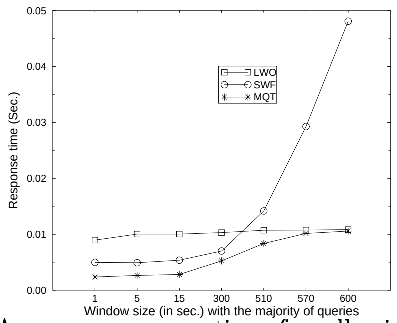


Figure 9: Average response time for all windows using different query distributions (hash-based).

Figures 8 shows the average response time per window for the $Small-Large$ distribution using an expected burst size of five. Figure 8 indicates that with large burst sizes, the SWF scheduling algorithm has a response time of 4 seconds for the largest window, whereas using the MQT scheduling algorithm it is bounded by 1 second. These results demonstrate the fact that efficient scheduling is important to maintain a reasonable response time, particularly in unpredictable environments.

5.3 Varying Query Distribution

In the previous experiments we assume a uniform query workload (single query per window size) over different window distributions. In this experiment we consider different skewness in query workload over a distribution of windows. Although, we present the results using the Small-Large window distribution, we obtained similar results when trying all the other windows distributions. We consider 80% of the queries share a single window, w_i, while the remaining 20% are uniformly distributed among the rest of the windows. We use hash-based implementations with same settings as in Section 5.1 and consider a pool of 30 queries. We vary w_i from w_1 to w_7 and report the results of average response time using each of the scheduling algorithms as shown in Figure 9. The MQT has the lowest average response time in all cases. When 80% of the queries are clustered at small windows we expect that SWF performs better than LWO. MQT outperforms SWF in this case. When the majority of the queries are clustered at large window, LWO performs better than SWF. However, MQT also schedules small window queries and provides the best average response time. The conclusion of this experiment is that, MQT adapts to a skewed query workloads and outperforms both LWO and SWF in terms of the average response time.

5.4 Memory Requirements

One concern about the SWF and MQT 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 without 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 MQT 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 MQT 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 with high burst sizes. 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 MQT scheduling algorithms are roughly comparable to that of the LWO algorithm. The maximum size for the output buffer in both the SWF and MQT 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 MQT algorithms are negligible when compared to the joinBuffer sizes.

6 Conclusions

Window joins are at the core of emerging architectures for continuous query processing over data streams. Shared processing of window joins is a key technique for achieving scalability and enhancing the operating range of such systems. We have described and evaluated three scheduling algorithms that prioritize such shared execution to reduce the average response time per query while preserving their original semantics. LWO was used previously, SWF and MQT were developed as part of the work described here. SWF directly addressed the performance flaw identified for LWO. MQT was motivated by the tradeoffs between LWO and SWF as identified by an analytical study of the two approaches. Experiments performed on an implementation of the three techniques in an extended DBMS under a variety of workloads and mixes of window sizes, validated the analytical results and showed that the MQT algorithm provides up to 60% improvement in average response time over the LWO algorithm. These experiments also demonstrated that the benefits of MQT become more pronounced as the burstiness of the input data streams is increased. The experiments also demonstrated that the benefits of MQT come at the cost of only a small increase in memory overhead.

References

[1] D. Carney, U. Cetintemel, M. Cherniack, and et al. Monitoring streams - a new class of data management applications. In *28th VLDB Conference, Aug.*, 2002.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, and et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *1st CIDR Conf., Jan.*, 2003.

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

[4] 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.

[5] 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.

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

[7] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proc. of the 15th SSDBM Conference, July*, 2003.

[8] Z. G. Ives, D. Florescu, M. Friedman, and et al. An adaptive query execution system for data integration. In *Proc. of the SIGMOD Conference*, 1999.

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

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

[11] R. Motwani, J. Widom, A. Arasu, and et al. Query processing, approximation, and resource management in a data stream management system. In *1st CIDR Conf., Jan.*, 2003.

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

[13] R. Sharma, C. Bash, and C. Patel. Dimensionless parameters for evaluation of thermal design and performance of large-scale data centers. In *Proc. of the 8th AIAA/ASME Joint Thermophysics and Heat Transfer Conference in St. Louis, June25.*, 2002.

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

[15] 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.