

# Evaluation of Rate-based Adaptivity in Asynchronous Data Stream Joins

Beth Plale and Nithya Vijayakumar

Computer Science Department  
Indiana University

## Abstract

Continuous query systems are an intuitive way for users to access streaming data in large-scale scientific applications containing many hundreds of streams. A challenge in these systems is to join streams in such a way that memory is conserved. Storing events that could not possibly participate in a join any longer wastes memory and limits scalability of the query processing system. This paper reports an experiment we conducted to validate an algorithm we developed for adaptive rate, adjustable join windows. We posit that a rate-based strategy can result in memory savings, can be sufficiently responsive to rapid changes in stream rates, and can execute with suitably low overhead. Based on the results, we conclude that the algorithm adds between 0.007% and 2.6% overhead, with significant gains in memory utilization possible depending on the particular workload.

Key Words and Phrases: data-driven applications, grid computing, continuous query systems, data streams, database query processing, meteorology, severe storm forecasting.

## 1 Introduction

Stream driven applications are growing in prevalence and importance. Examples include stock ticks in financial applications, performance measurements in network monitoring and traffic management, log records or click-streams in web tracking and personalization, data feeds from sensor applications, network packets and messages in firewall-based security, call detail records in telecommunications and so on.

The motivation for our work are dynamic data driven applications that ingest and react to information about their environment. Specifically, mesoscale meteorology simulations, such as WRF [13], are forecast models used to predict the occurrence of severe storms and tornadoes. It has been demonstrated that simulations can achieve higher levels of accuracy if they can draw data

on demand from regional observational sources (*e.g.*, regional NEXRAD Doppler radars). Computer science and meteorology researchers, collaborating in projects like LEAD [7], are building the infrastructure that enables this dynamic interaction with the weather. The simulations in LEAD, for instance, will be able to ingest current weather conditions from data streaming from sensors and instruments in real time.

The problem we investigate in this context is the synchronizing of data streams with a simulation. The simulation computes faster than real time, ingesting input conditions then computing until it reaches a certain state. If at that time no defining weather condition has been determined by the simulation, it will again draw the most recent weather conditions from the data streams. This cycle continues until a defining weather condition occurs or the simulation is terminated. The data streams used in meteorology simulation consist of a dozen or so observational data products that vary considerably in type, format and frequency of generation. For instance, meteorology aviation routine (METAR) reports are surface observations (versus satellite observations.) The report is small, often less than 1 kilobyte in size, is text-based, and is generated every two minutes from 270 National Weather Service sites and every hour from an additional 1000 sites. GOES data is visible and infrared imagery data generated every hour. The images arrive in a compressed binary format and are 4-5 MB in size, compressed. Sensor data will be generated from regional sensors mounted on cell phone towers. The first deployment is slated for Spring 2005 in Oklahoma City. The sensors will generate 125 KB events, binary encoded, twice a minute. Further details are given in Section 3

Meteorology researchers are more advanced than other geoscience disciplines in their handling of streaming data due in large part to the meteorologist's use of weather observations as the principal data products in their investigations. Weather observations are, by their time-based nature, streaming products. This long history has resulted in sophisticated production-strength data dissemination systems in use today. The Unidata Internet Data Distribution

System (IDD) [4] for instance distributes numerous surface and satellite observation products. IDD can be viewed as a topic-based publish-subscribe system that routes by means of an overlay network. The overlay network is US-based involving government and university resources of 150 institutions. One receives data streams from IDD by installing a publicly available client, called an LDM client, at their site.

Our work assumes the existence of a data dissemination system such as IDD to deliver weather data streams to the machine or cluster on which the simulations will run. Our work addresses the last mile, that is, the synchronization between stream and forecast simulation.

We propose a novel model of interaction between the simulation and data streams that is based on request-response query model. That is, when the simulation needs the “next timestep” from a set of streams, it issues a query to a continuous query system that is capable of satisfying that request in a timely manner. A common approach to synchronizing with a simulation is to interrupt the simulation, then push new data to it once the simulation has reached a consistent state. We believe the query approach is superior. When the model reaches a consistent state it specifically requests new data by means of a database query. Database queries are advantageous for several reasons. They are easy for a user to understand. Further, query processing can be made resilient to dropped or slow streams. This allows the model to continue, albeit under reduced conditions until the stream picks back up. Finally, most query languages are declarative. This means the query can be mapped efficiently to different back end systems in a transparent way. A simulation could request data from a data stream or request from a database, without change to the model-data interface [15].

In this paper we report the results of an experimental evaluation we conducted on a join window sizing algorithm proposed in [14]. The join window is key in database-oriented systems that execute queries over asynchronous streams. The algorithm addresses the challenge of choosing the correct size of a join window. The evaluation is in three parts. The first experiment is a microbenchmark of algorithm efficiency. The second experiment evaluates performance under a synthetic workload. While the workload is artificial, it still provides a feel for the memory savings that can be achieved. The third experiment attempts to capture performance under a realistic workload. We developed a workload generator that models the key meteorology data products of interest to our collaborators.

The contribution of this paper is a performance study of a rate-sensitive algorithm for dynamic adaptation of join window sizes for time-based joins. The tests include a realistic application-specific workload using stream characteristics derived from the application domain of mesoscale meteorology. The study makes a convincing argument for

the use of time-based join window sizes, and for dynamic adaptation of join window sizes in response to stream rates for streams that are highly asynchronous and have rate attributes that are domain dependent. The algorithm was implemented in the dQUOB system [17]. In this system, the algorithm adds between 0.007% and 2.6% overhead with significant gains in memory utilization possible depending on the particular workload.

The remainder of the paper is organized as follows. The following section discusses query processing over streams, time-based joins, and join windows. In Section 3 we discuss the meteorology streams that motivate our work. Section 4 describes the experiment. Section 5 describes related work. The paper concludes with future work in Section 6.

## 2 Query Processing Over Streams

### 2.1 Architecture

Continuous query systems are data driven systems that accept events in real time, and provide database query access to data in the data streams. dQUOB is an example; other examples are discussed in Section 5. A *query* is implemented in dQUOB as a directed acyclic graph with multiple entry points and a single exit point. Nodes are query operators (*i.e.*, select, project, join) and the arcs indicate direction of control flow, that is, the order in which operators are applied.

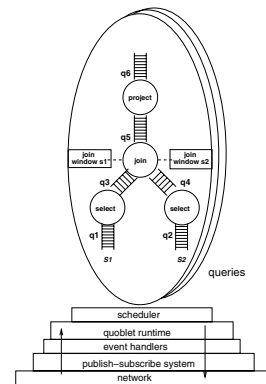


Figure 1: Architecture of continuous query system

A query is triggered whenever an event of interest arrives. An arriving event is queued at the query, then pushed through the query operators (*e.g.*, select, project, join, union) by means of a depth-first traversal of the query tree. As depicted in Figure 1, a query is scheduled by a scheduler built into the quoblet runtime. The scheduler queues events to the queries who registered for that event. Underlying the quoblet runtime is a publish-subscribe system that streams events from the data generation sources;

the runtime subscribes to the sources of interest. Event handlers handle the arriving events and pass them off to the runtime for scheduling.

To illustrate, an arriving event on event channel  $S1$  is queued at select  $\alpha_1$ . When control passes from the scheduler to the query,  $\alpha_1$  dequeues the event, processes it then queues the resulting tuple to queue  $q3$ . Control passes to the join operator. The join operator matches the arriving event with events in the  $S2$  join window, generating possibly multiple matches which are queued to  $q5$ . Control passes to the project operator which generates the outgoing tuple. The resulting tuple is published to the publish-subscribe system for downstream recipients.

## 2.2 Time-based Joins

A data stream is an indefinite sequence of time ordered events,  $\alpha = \langle e_1, e_2, e_3, \dots, e_n, e_{n+1} \rangle$  where  $timestamp(e_1) < timestamp(e_2)$ , etc. A query,  $Q1$  is a processing unit that takes one or more event sequences (e.g.,  $\alpha, \beta$ ) as input, and generates a new event sequence,  $\gamma$ , as output.  $Q1$  is event driven; that is, it is triggered by the arrival of a new event arriving for  $\alpha$  or  $\beta$ . An arriving event,  $e_i \in \alpha$  corresponds to 0 or more resulting events  $\gamma_k$ .

A join window is a subsequence of the input sequence ( $\alpha$  or  $\beta$ ). A join operator maintains one join window for each of the two input streams it is joining. The subsequence always contains the last arriving event in the sequence, and extends earlier in time to include some number of events. For example, for a join window size of two on event sequence  $\alpha$ , the join window consists of  $\langle e_n, e_{n+1} \rangle$ .

The join window is a necessary concept in continuous query systems because these systems support one or more forms of the database join operator. Since a data stream is an indefinite and potentially infinite sequence of events, any timely processing of a stream must be done on a subsequence of the stream. That subsequence is often called a *join window* or sliding window [16].

dQUOB supports a time-based, nested loop join operator. That is, it joins two streams on one of several notions of time in a manner as follows: suppose the existence of two streams,  $S1$  and  $S2$ , as depicted in Figure 1. Suppose further that a new event from  $S1$  arrive at the join operator, then

- $s2$ 's join window is scanned for tuples that satisfy the time-based match criteria; matching tuples queued to outgoing queue,
- $s1$  event is queued to "join window  $s1$ ",
- Oldest event is dequeued from "join window  $s1$ " and its space reclaimed if no further reference exists to the event,

- Rate sizing algorithm is executed. If yields new join window size then effect the change, and
- Control passed to project operator.

A time-based match occurs if the pair of events to be compared match plus or minus ( $\pm$ ) some epsilon,  $\epsilon$ . Specifically, suppose the two streams  $S1$  and  $S2$  have rates  $r1$  and  $r2$  respectively. An event  $i$  of  $S1$  will "match" an event  $j$  of  $S2$  IFF  $(S1_i.timestamp \text{ equals } S2_j.timestamp) \pm \epsilon$  where  $\epsilon$  is determined at runtime as follows: Determining a good size for the join window can

$$\epsilon = \max(r_1, r_2) / 2$$

be difficult. Approaches in literature vary. The window size can be fixed and static or can vary across streams; it can be a system-wide parameter set at startup, or can be specified by the user on a query-by-query basis by an extension to the query language. Our experience in applying a prototype continuous query system to several application domains (i.e., safety critical robot control [18], a global atmospheric transport model [10], and now severe storm forecasting) has led us to conclude that in order to work in a realistic setting, the join window size must be configurable by the user, and allowed to vary in size from stream to stream. User configurable is key because only the user has sufficient understanding of the latency and asynchronism of application-specific streams. Take observation weather streams as an example. If the user were querying information from the regional sensors located throughout the Oklahoma University campus, then any time skew that might exist between streams will largely due to skews between the clocks of the sensor devices. Network latency would likely be small and fairly uniform. Once meteorological data products are combined, however, clock skew and network latencies become dwarfed by the larger issues of generation rate and non-uniform interarrival rates.

We further posit that a join window size set based on time is far better suited to application specific data streams. The user has a much better feel for the temporal skew that exists between application streams, so specifying a join window as the maximum of the anticipated latency can be done with relative ease. Specifying the join window size as an integer count, on the other hand, has several problems. It is far less intuitive for a user, for one. The mapping from the time domain to an integer count can be difficult (our algorithm does this.) Finally, the cost of getting the size wrong can be great. Too small a join window size can result in a high rate of false negatives, that is, events that should have been paired on time but were not because the earlier arriving event had already been dropped from the window. A join window size too large wastes memory and hence limits system scalability both in terms of number of queries and number of streams.

The algorithm evaluated here takes a time interval as

input, then the join window algorithm dynamically maintains that interval for the duration of execution, dynamically adapting to changes in stream rates. “Slow streams” have smaller join windows so use less memory. “Fast streams” have larger join windows.

### 2.3 Dynamic Sizing Join Window

The intuition behind *Rate Sizing Algorithm (RS-Algo)* [14] is that two join windows are held at a fixed time interval and in relation to one another for the duration of the application, despite fluctuation in the stream rates. For applications with highly asynchronous streams, this optimization can improve memory utilization because the “slower” streams will consume less memory because fewer events are returned in the sliding window. For example, suppose a query joins two streams, one with a rate of 1000 events per second, and another of 1 event per second. A join window size kept at 2 seconds means the query must retain in memory all events received in the last 2 seconds. For a time-based sliding window, this equals to storage for 2002 events whereas for a fixed integer count strategy, 4000 events would need to be retained.

Specifically, RS-Algo maintains join windows at a size that can be expressed as a timestamp interval and adapts the sliding window size at runtime in response to detected changes in stream rate. By doing so, two gains are had. First, a timestamp interval enables the user to better reason about the trade-off between performance and increased incident of false negatives. Second, we achieve window sizes that mirror differences in stream rates almost as a byproduct. Unfortunately, timestamp information is often not available to a user at application startup, so we let the user specify an interval in wall-clock time instead, then map the interval into the timestamp domain during initial stream sampling.

```

rate_sizing(window_interval) {
  for all i concurrently {
    sample event stream[i] for duration of window_interval;
    barrier();
    max_timestamp_interval = last_event[i].timestamp -
                             first_event[i].timestamp;
    join_window_size[i] = (event_received[i] * window_interval)
                        / max_timestamp_interval;
  }
  change_window_size[i]();
}

```

Figure 2: RS-Algo dynamic rate sizing algorithm in pseudocode.

The algorithm is shown in Figure 2. Two streams participate in a join operation. The algorithm accepts a single input parameter, *window\_interval*, the interval of time

maintained in the sliding window. This parameter is specified by the user as part of the query, and represents the user’s best guess as to skew between streams in a query. When the query is first started up, sampling of all streams is done. Once completed, the maximum time stamp interval is computed. This step maps the time interval from wallclock time, which the user specified, to timestamp time, which more accurately reflects the state of the system. The join window size is then calculated, and the change effected if the difference is sufficiently high.

## 3 Meteorology Data Streams

The realistic workload used in this study is drawn from the meteorological research domain. The LEAD [7] project, in which our lab is involved, has identified 9 data products that are key to the meteorology domain. These products are described below and in Table 1. Product descriptions, current number of sources (*e.g.*, number of Doppler radars located throughout the continental United States) and approximate size of an event - be it a single volume scan from a radar, one aircraft textual report, or a single satellite image. Bandwidth and generation rates for the streams are given in Table 1.

- METARS-1st order – surface observations from Nat’l Weather Service. 270 sources, events are 1-5KB in size.
- METARS-2nd order – surface observations, ship and buoy. 1000 sources, 1-5KB events.
- Rawinsondes – upper air balloon soundings. 94 sources, 21-25KB events.
- ACARS – weather reports from commercial aircraft. 257 sources, 100-700KB events.
- Nexrad Level II – Doppler Radar data. 130 sources throughout the continental US, clear weather events are 163KB and storm events are 1.7MB.
- Nexrad Level III – Doppler Radar derived products. Same sources as Level II data, events are 2-20KB.
- GOES – visible and infrared imagery. 5 sources, 4.4MB events.
- Eta – NWS operational forecast model gridded analyses. 2 sources, 94-200KB events.
- CAPS sensors – regional radars mounted on cell phone towers. Approximately 9 located throughout a region, 0.5MB events

The meteorology streams provide a realistic scenario in which to test the rate sizing algorithm.

## 4 Experimental Evaluation

We experimentally validate our theory that adaptive rate, adjustable join windows can result in memory savings,

<i>Data Product</i>	<i>Gen Rate (per instrument)</i>	<i>Max. Bandwidth</i>
METARS 1st order	3 ev/hr	4,050 KB/hr (9 Kbps)
METARS 2nd order	1 ev/hr	5,000 KB/hr (11 Kbps)
Rawinsondes	1/12 ev/hr	195.8 KB/hr (0.435 Kbps)
Acars	10 ev/hr	1.8 GB/hr (3.9 Mbps)
Nexrad Level II	6 ev/hr(clear), 12 ev/hr(storm)	2.6 GB/hr (5.9 Mbps)
Nexrad Level III	6 ev/hr(clear), 12 ev/hr(storm)	31.2 MB/hr (69Kbps)
GOES	2 ev/hr	44MB/hr (97.7Kbps)
Eta	1/6 ev/hr	66 KB/hr (0.148 Kbps)
CAPS sensors	120ev/hr(clear), 60ev/hr(storm)	540MB/hr (1.2Mbps)

Table 1: Bandwidth and generation rates of the various meteorological streams.

while being responsive to rapid changes in stream rates, and with suitably low overhead. Our experiment consists of multiple parts. First, we microbenchmark the rate sizing algorithm. We then use a synthetic workload to understand adaptability and memory savings. We finally test a realistic workload based on meteorology streams.

An important measure used in the study is the elapsed time between the arrival of an event at the query processing server (called a *quoblet*) and the recognition of that event. However, behavior is often distributed across multiple remote sources, so time to recognition must include the time to collect all events making up a particular incident. The key performance metrics used in our study are service time and memory utilization. *Service time* is the elapsed time between the arrival of the first event of an incident and the generation of the tuple that recognizes it. *Average Service time* is the average of the service times taken over some number of result tuples. *Memory utilization* is measured as the total number of events resident in memory at any one time multiplied by the size of each event. All events are memory resident. An event is removed from memory when it is no longer needed by any of the queries. For events that participate in a join operation, events are retained in memory for as long as they are present in at least one sliding join window [17].

The experiment environment is as follows: the streams generator is distributed across 4 nodes of an 8 node Linux

cluster running Redhat 8.0. Cluster nodes are Xeon Intel dual 2.8GHz processors with 2GB memory. They are connected to the SAN via a 2Gbps Qlogic SANBlade 2300 Fibre Channel Host Bus Adapter. The query processing (quoblet) was performed on a dual processor Dell Precision WorkStation 450 2.8GHz processor with 2GB memory running RedHat 9.0. The network interconnect is 1000 Mbps Gigabit Ethernet.

## 4.1 Synthetic Workload

The synthetic workload consists of three synthetic data streams. Each stream carries data events of a single type. Specifically, the streams, D, R, and M are as follows:

- D events - large events, 50KB in size. Stream has fixed rate of 10 events/second.
- R events - small events of few bytes in size. Has variable rate of 10 events/second alternating with 1 event/second over 10 second intervals.
- M events - small events of a few bytes in size. Stream rate is 10 events/second.

The workload is issued against a single continuously executing query running in a quoblet runtime container. The duration of the run is 3 minutes. The workload to the query is two asynchronous streams, one with fixed rate and one with variable rate. The query accepts two input streams, D and R, and joins the streams together based on timestamp to produce the aggregate event  $\langle D, R \rangle$ . The input/output behavior of the query is illustrated in Figure 3.

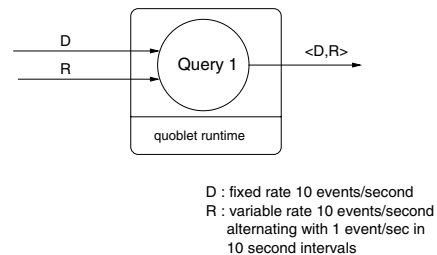


Figure 3: Input/Output behavior of query.

The query details are given in Figure 4. An arriving event is received by an event handler, which copies the event off the socket into the event buffer. It is removed from the buffer by a dispatcher (not shown) and scheduled at the queries who have previously registered an interest in the event. In the example, the dispatcher passes events that are of type D or R to the gate operator of query “Q3”. The gate operator is an implementation artifact to provide a single entry point to a query. An operator, called “Sample OP”, exists on each stream that feeds the join in order to sample the stream. The events, along with stream rates, are passed to the next operator, the join operator, shown in the large bubble. RS-Algo, when triggered, examines the

stream rates and calculates new window sizes. Window sizes may be adjusted in the next step. The time-based match pairs the arriving event, say  $D_i$  with all  $R_j$  in the  $R$  join window. The select operation filters event pairs that do not match on ID.

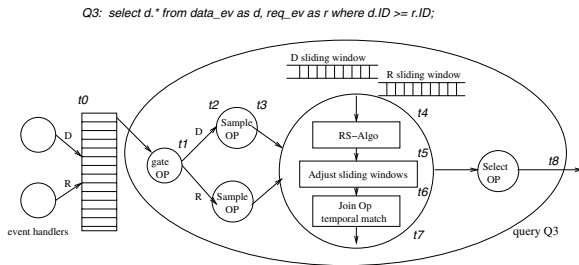


Figure 4: Query graph for SQL query shown at top of figure. Events arrive at left and are received by event handlers. Events are pushed through query graph depth-first. Instrumentation points depicted by ' $t_i$ '.

## 4.2 Microbenchmark

The microbenchmarks break down the overhead of the join window algorithm. The scenario, depicted in Figure 3, a single query run in a quoblet container accepts two input streams, D and R, and joins those streams together based on timestamp to produce the aggregate event  $\langle D, R \rangle$ . The rate sizing algorithm embedded in the query executes periodically to maintain an appropriate join window size for the pair of events. Based on a newly calculated join window size, the sliding windows for D and R may be adjusted. When the window size is reduced, “oldest” events are removed from the window and if not needed by other queries, deleted from the system.

Figure 4 depicts the instrumentation points used in the microbenchmark.  $t_0$  marks the instant an event is inserted into the event buffer. An event is selected for scheduling to query Q3; its arrival at the gate operator marks the instant measured by  $t_1$ . Entry and exit from a sampling operator is marked by  $t_2$  and  $t_3$  respectively. Hence  $t_3 - t_2$  is the sampling overhead. Stream rate information is appended to the event and pushed to the join operator. The time consumed in executing the rate sizing algorithm, adjusting the sliding window, and executing the join condition are given by  $t_5 - t_4$ ,  $t_6 - t_5$  and  $t_7 - t_6$  respectively. The total service time for an event in the system is  $t_8 - t_0$ . The microbenchmark results for the query and stream scenario described are shown in Table 2.

Algorithm overhead is computed as the sum of sampling time, algorithm execution time, and time to resize buffer. More complex queries incur sometimes significantly higher cost in join processing (Cartesian product execution), time spent in queues, and operator processing,

Elapsed time	Processing Interval	Avg. Elapsed Time (ms)
Wait time in input buffer	$t_1 - t_0$	0.0187
Sampling time	$t_3 - t_2$	0.0018
Algorithm execution time	$t_5 - t_4$	0.014
Time to resize join buffer	$t_6 - t_5$	0.0029
Wait time in join window	$t_7 - t_6$	5.70
Total Service time	$t_8 - t_0$	5.73

Table 2: Microbenchmark results for a single join query that joins two asynchronous streams, one of steady rate and the other variable.

but the algorithm overhead remains roughly constant. The average overhead for RS-Algo taken over the 3 three synthetic workloads is 0.5 ms. As a fraction of total service time for the cases we tested, the algorithm overhead varies between 0.007% and 2.6%.

The elapsed times shown in Table 2 are averages. While further study not reported here revealed that algorithm overhead is constant over different workloads, time spent in the input buffer (upon arrival) and in the join window can vary as a function of stream rates. This is illustrated in Figure 5.

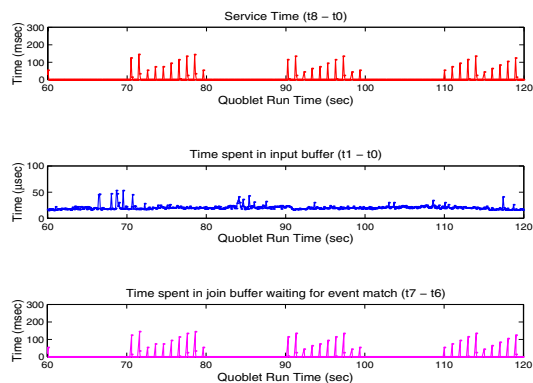


Figure 5: Breakdown showing time spent in input buffer and in join window.

The 10 second periodicity in the stream rate of R is reflected in the average time that D events wait in the join buffer (see lower graph.) Specifically, D’s average time in the join window increases as the rate of stream R drops to 1 event/second because event  $D_i$  is waiting for event  $R_j$  to arrive in order for the time based join to occur. Specifically, the join buffer in the figure is D’s join buffer. During the time R events are arriving at the same rate as D events, D events do not wait. When R events drop to 1 event/second, as in the intervals 60-70, 80-90, etc., the

time a  $D$  event spends in the join buffer increases.

### 4.3 Using Synthetic Workload to Examine Memory Utilization

We next investigate the impact of the RSAIgo algorithm on memory utilization, and do so by applying the synthetic workload to two queries: one query, shown in Figure 4, contains a single join operator, and the more complex multi-join query shown in Figure 7. The latter query generates the aggregate tuple  $\langle M, R \rangle$  that is dependent on upstream generation of the aggregate tuple  $\langle D, R \rangle$ .

Memory utilization for the simpler, single join query is plotted in Figure 6. The graph plots a 180 second run of a synthetic workload against the simple single join query described in Figure 4. The line that begins at 130 and quickly becomes steady at a join window size of approximately 250 depicts join window size for a steady stream of 50 KB events arriving at 25 events/second. The variable line captures a rapidly changing join window size in response to a stream that alternates between issuing requests at 25 events/second and 1 event/second at 10 second intervals.

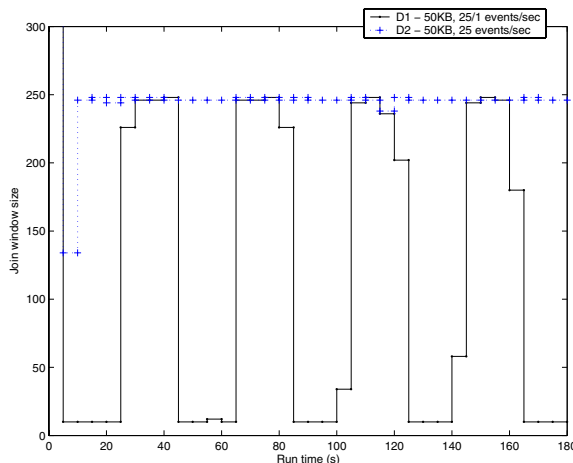


Figure 6: Join window size shown varying over a 180 second run. The query has a single join and the time interval maintained in the join window is 10 seconds.

The improvement in memory utilization for this particular workload is clear. When  $D1$  drops to a join window size of a few events, memory is released as the events that no longer fall within the desired time interval are removed. The time interval in this scenario is 10 seconds, meaning that the system maintains 10 seconds of events for each stream at all times.

The memory utilization can be computed by multiplying the join window size by the event size by the number of streams. For instance, at time 40 into the run, mem-

ory utilization of this query is 25,000 KB, (250 window \* 50KB \* 2 streams). At second 60, utilization drops to 12,550 KB. The overall memory savings for this query is roughly 25%. The benefits of the adaptive approach are realized when a query evaluation engine (the quoblet) is executing many queries at different stream rates and join window sizes. Then the cumulative effect of the per query memory savings demonstrated here become dominant.

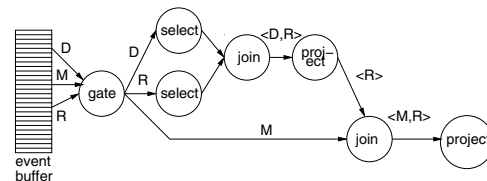


Figure 7: Multi-join query: accepts 3 data streams ( $D$ ,  $M$ ,  $R$ ) and executes two joins over the streams.

We next consider a more complex query case consisting of two join queries as depicted in Figure 7. The query receives 3 streams,  $D$ ,  $R$ , and  $M$  whose rate characteristics are described in Section 4.1. Note that  $D$  and  $R$  events are pushed through a select before being joined. The ordering of select operators before a join is a common query optimization heuristic because select operators act as filters, thus reducing the number of tuples that participate in the more expensive Cartesian Product. For better control over our measurements, we use selects that filter zero events. The join operator creates the tuple  $\langle D, R \rangle$  which it passes to the project operator. The latter, somewhat artificially, projects a new  $R$  tuple. This is passed to the second join which joins  $R$  and  $M$  to produce  $\langle M, R \rangle$  which is then projected to obtain the final tuple result.

### 4.4 Realistic Workload

We developed a realistic workload based on the meteorology streams of Table 3. We simulate a load of four data products: METARS, Acars, Nexrad Level III, and CAPS sensor data coming from multiple data sources. The generation rates are sped up by a factor of 8 from their normal rates. The Doppler radars and CAPS sensors are capable of operating in a storm mode where the generation rate is higher and event size larger than in clear air mode. The interarrival rate varies  $\pm 10\%$ . In reality, interarrival rates of meteorology streams vary considerably;  $\pm 10\%$  is an approximation used while we examine logs to characterize it more precisely. Storms are introduced into the workload at minutes 20, 80 and 140 and lasting 20 minutes in duration. The experiment executes for 3 hours.

At present the data sources are consolidated into a few streams. That is, the reports from all 27 METARS sources flow down a single channel. This is a current limitation.

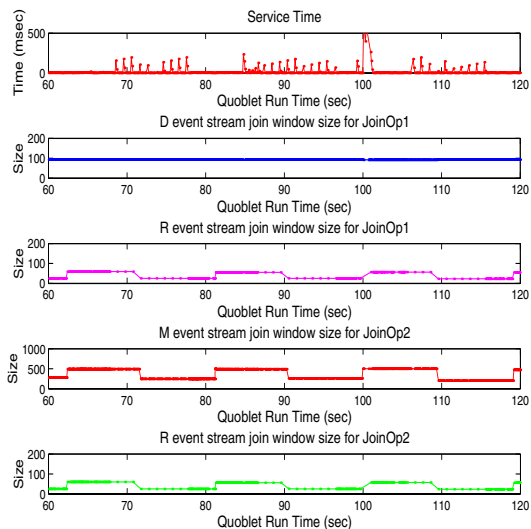


Figure 8: Join window results for query with two join operators. .

The publish-subscribe system requires that events within a channel have monotonically increasing timestamps, so in effect the generator imposes a total order on all the sources it represents. Monotonicity is not an unreasonable assumption since event channels are cheap to create, and often generation sources are spatially distant from one another. One could support unordered flows in an event channel by adding a module that would instill a total order on the events in a stream before events are scheduled. Ordering events in a stream is well understood [9]. The current implementation further precludes concurrency between events arriving from different sources. The workload generator is a standalone tool parameterized with the characteristics of one or more data products.

We first issue the realistic workload against a query that joins the four data products in the order as follows:  $join(join(join(METAR, Acars), CAPS), Nexrad)$ . The results for the first 20 minutes of execution are shown in Figure 9. The graph captures the startup oscillation that occurs during the first 10 minutes of the run as the rate sizing algorithm is adjusting to poor initial assumptions. The default join window size is 100. As it turns out, this is a relatively good guess. The default  $\epsilon$  on the other hand (the  $\pm$  range defining a “time-based match”) is set far too large: at 30 minutes. The large  $\epsilon$  means that too many event pairs satisfy the matching criteria and are passed on to the second join operator. This is why the plot for  $agg(metar, acars)$  shows the join window size suddenly shooting up to 1300 before slowly dropping back down. One can see from the startup figure that all streams start at this larger window size but within the first two minutes settle into more realistic window sizes. This demonstrates a second form

<i>Product</i>	<i>No. Srcs</i>	<i>Size (one instance)</i>	<i>Agg Gen Rate</i>
METARS	27	1KB (clear) 5KB (storm)	0.18 ev/sec (648 ev/hr)
Acars	30	100 KB (clear) 700 KB (storm)	0.39 ev/sec (1440 ev/hr)
Nexrad Level III	13	2 KB (clear) 20KB (storm)	0.170 ev/sec (624 ev/hr) 0.340 ev/sec (1248 ev/hr)
CAPS sensors	10	16KB (clear) 64KB (storm)	0.26 ev/sec (960 ev/hr) 1.30 ev/sec (4800 ev/hr)

Table 3: Realistic Workload characteristics: Product name, number of sources generating data flows, size per single instance, aggregate generation rate.

of memory savings. That is, if the user sets an arbitrarily large join window size, and there is no mechanism to adjust it once execution begins, the window will stay arbitrarily large, resulting in poor memory utilization.

The remainder of the execution, plotted in Figure 10, demonstrates that the window sizes settle out into a regular pattern. Notable behavior in this rather complex graph is the rapid response by the algorithm to storm modes at minutes 20, 80, and 140. The memory savings accrued by the approach is due to smaller join window sizes under clear weather conditions.

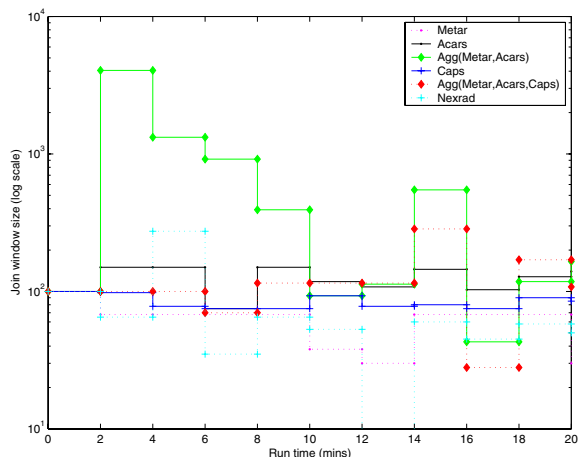


Figure 9: Realistic workload: Depicts startup oscillation as algorithm adjusts from a starting window size of 100 and  $\epsilon$  of 30 minutes to a steady window size hovering around 100 and an  $\epsilon$  of less than 1 second.

We then reordered the query so that the fastest

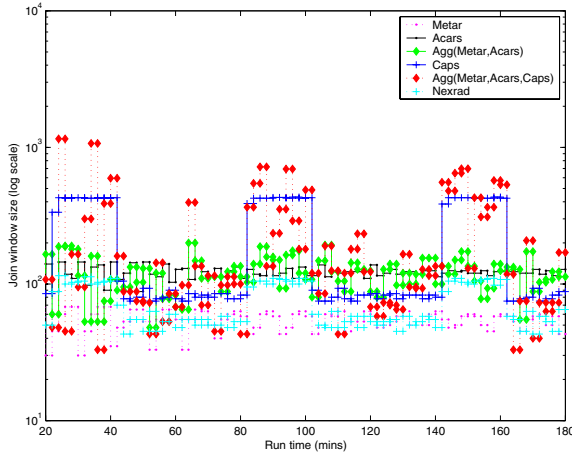


Figure 10: Realistic workload for the remainder of the run. Storms present at 20-40, 60-80, and 140-160 are evident by increases in join window sizes for the aggregate streams (streams produced as result of join.)

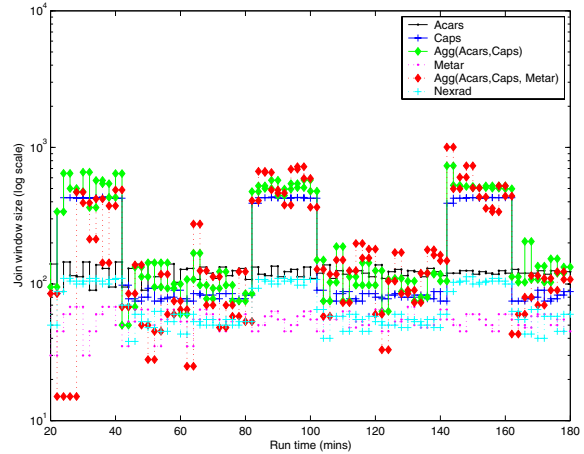


Figure 11: Realistic workload interval from minute 20 to minute 180. Shows detail of steady state.

stream (the CAPS stream) participates in the first join:  $join(join(join(Acars,CAPS),METARS),Nexrad)$ . The results are plotted in Figure 11. The storm surges, introduced at minutes 20, 60, and 140, are clearly visible. In comparison of Figures 10 and 11, the queries are distinguishable by the density of the triangles located above the 400 mark during storm mode in Figure 11 compared to a far sparser set of triangles in the other. The density is caused by the CAPS stream. When CAPS participates in the first join, the memory usage by that stream is higher than when it participates in the second join. These results would suggest that faster streams need to be joined later in the query.

## 5 Related Work

**Continuous Query Systems.** Aurora [1] is a continuous query system designed to run on a single server. It uses TCP sockets as its underlying communication protocol and defines a communication protocol data format that is closely tied to the format of Aurora’s internal queue format. NiagraCQ [5] is a continuous query system that runs against web XML sources, intermediate files, or on local disk. It supports a variant of XML: XML-QL. Queries are triggered by either timer events or file modifications. The NiagraCQ project has done interesting work in rate-based optimizations and characterizing stream rates through query operators. Its file orientation makes it less interesting for scientific streaming demands. The Stanford STREAM [3] project supports queries over a database containing data streams and traditional rela-

tions. The system architecture contains many architectural similarities with our prototype, dQUOB. However, it is a fully centralized model, and appears to have no available code from which to explore it further. Eddies [2] executes queries over widely distributed information resources, such as massively parallel database systems. It achieves dynamic operator ordering with an event-driven model to query evaluation. Eddies targets query optimization in a database setting, but shares the same goal as dQUOB of being responsive to changes in the environment. The Fjords data-flow architecture [12] focuses on database queries to process streaming data from networks of ad hoc collections of sensors. It defines the zipper join, which “zippers” together synchronous streams based on a time interval. We discovered in our work with application domain streams, however, that synchronized streams are the exception rather than the rule. The Continual Queries system [11] computes and return the difference between current query results and prior results of changing web pages. This approach complements our work, which is optimized to continuously execute and return full results of a query in a highly efficient manner. Gehrke [19] pushes aggregation queries into a sensor network. Because sensor nodes consume power whenever they must activate the radio transmitter to send a message, the aggregation operator allows them to collect messages from downstream nodes in the ad hoc network, then aggregate the messages into a single message that is transmitted upstream. The work was tested in a simulation environment.

**Data Flow Computing.** The GATES [6] project takes a data flow model view of stream processing, and focus their efforts on encoding computational elements in the flow as grid services capable of responding to commands to adapt their behavior to changes in their environment. Compo-

sition is more difficult in the data flow model than in the stream query model because the flow model has no well defined algebra. User interaction with a data flow system is a programming problem that the user must solve. The Active Data Repository [8] evaluates SQL-like queries to satisfy client needs. However, queries are evaluated at the source over a physical database. This is in contrast to our domain of continuous query execution over data streams [15].

## 6 Conclusions and Future Work

This paper focuses on optimizations for improved memory utilization under the large scale, asynchronous streams found in on-demand meteorological forecasting and other data-driven applications. Specifically, we experimentally evaluate the RS-Algo algorithm for dynamically adjusting sliding window sizes used in database joins to reflect current stream rates. The average overhead of the RS-Algo algorithm over the three synthetic workloads was found to be between 0.007% and 2.6% of total service time.

Our current efforts are focused on refining the workload to be more realistic. In particular, we are obtaining logs of arrival times for the various data stream products described in this paper in order to more precisely characterize the deviation in interarrival rate within the streams. We are also packaging the stream generator for release. Future work will include comparing dQUOB with the rate algorithm to a system such as Telegraph under conditions of the realistic workload described in this paper.

## References

- [1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. In *Very Large Database (VLDB) Journal*, 12(2):120–139, August 2003.
- [2] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*. ACM Press, 2000.
- [3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*. ACM Press, June 2002.
- [4] Mitchell S. Baltuch. Unidata’s internet data distribution (IDD) system: Two years of data delivery. In *Proceedings of the Thirteenth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, Anaheim California, February 1997.
- [5] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 379–390. ACM Press, 2000.
- [6] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. GATES: A grid-based middleware for processing distributed data streams. In *Proceedings Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*. IEEE Computer Society, June 2004.
- [7] Kelvin K. Droegemeier, V. Chandrasekar, Richard Clark, Dennis Gannon, Sara Graves, Everette Joseph, Mohan Ramamurthy, Robert Wilhelmson, Keith Brewster, Ben Domenico, Theresa Leyton, Vernon Morris, Donald Murray, Beth Plale, Rahul Ramachandran, Daniel Reed, John Rushing, Daniel Weber, Anne Wilson, Ming Xue, and Sepideh Yalda. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*, Seattle, WA, 2004.
- [8] Renato Ferreira, Tahsin Kurc, Michael Beynon, Chialin Chang, and Joel Saltz. Object-relational queries into multidimensional databases with the Active Data Repository. *Journal of Supercomputer Applications and High Performance Computing (IJSA)*, 1999.
- [9] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.
- [10] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concur. Pract. Exper.*, 8(9):639–666, November 1996.
- [11] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering, Special issue on Web Technologies*, January 1999.
- [12] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *International Conference on Data Engineering (ICDE)*, 2002.
- [13] J. Michalakes, S. Chen, J. Dudhia, L. Hart, J. Klemp, J. Middlecoff, and W. Skamarock. Development of a next generation regional weather research and forecast model. In Walter Zwielfhofer and Norbert Kreitz, editors, *Developments in Teracomputing: Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*, pages 269–276. World Scientific, 2001.
- [14] Beth Plale. Leveraging run time knowledge about event rates to improve memory utilization in wide area data stream filtering. In *Proceedings Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 171 – 176, Edinburgh, Scotland, August 2002. IEEE Computer Society.
- [15] Beth Plale. Using global snapshots to access data streams on the grid. In *Lecture Notes in Computer science*, volume 3165. Springer Verlag, 2004. 2nd European Across Grids Conference (AxGrids).
- [16] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *Proc. 9th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2000. IEEE Computer Society.
- [17] Beth Plale and Karsten Schwan. Dynamic querying of streaming data with the dQUOB system. *IEEE Transactions in Parallel and Distributed Systems*, 14(4):422 – 432, April 2003.
- [18] Beth (Plale) Schroeder, Sudhir Aggarwal, and Karsten Schwan. Software approach to hazard detection using on-line analysis of safety constraints. In *Proceedings 16th Symposium on Reliable and Distributed Systems SRDS97*, pages 80–87. IEEE Computer Society, October 1997.
- [19] Yong Yao and Johannes Gehrke. Query processing for sensor networks. In *First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.