

RS-Algo: an Algorithm for Improved Memory Utilization in Continuous Query System under Asynchronous Data Streams

Nithya Vijayakumar and Beth Plale

Computer Science Department
Indiana University
Bloomington, IN

IUCS TR 601

Abstract

Continuous query systems are an intuitive way for users to access data streaming data in large scale scientific applications containing hundreds if not thousands of streams. 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.

In this TR we provide detailed results of measurements of the RS-Algo algorithm which dynamically adjusts the sliding window sizes used in database joins to reflect current stream rates. We tested under various queries and stream rate conditions using a synthetic workloads and using the dQUOB continuous query system. We examine service time and memory utilization. The second result chronicles a smaller experiment comparing two algorithms used during runtime to schedule events. We evaluated the performance of the existing algorithm, First Come First Serve (FCFS), against a modified version of Earliest Job First (EJF) and examined the impact on memory utilization.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Computer Systems Organization**]: Performance of Systems – Performance attributes; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Experimentation, Performance

Additional Key Words and Phrases: data-driven applications, grid computing, continuous query systems, database queries architecture

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.

Applications that process data streams often provide access to the streams by means of database queries [15, 3, 11, 20]. These are called *continuous query systems* because stream processing is by means of long lived SQL queries that execute continuously, triggered by event arrivals. Experience we have gained in the application of our continuous query system, dQUOB [15], to scientific applications has led us to the following observations:

- In scientific streaming applications, the events making up the data streams are often large, into the megabyte range,
- Data streams rates vary significantly relative to one another, and
- Stream rates change over time as the application and environment undergo changes.

Continuous query systems often execute in core, that is, data resides completely in memory. Scientific applications impose a burden on continuous query systems because event sizes are large, often in the MB range, so scalability to thousands of streams can be limited by available memory. In response to the demands of scientific applications, we have developed the *Rate Sizing Algorithm (RS-Algo)* [14] for dynamically adjusting sliding window sizes used in database joins to reflect current stream rates. Specifically, in dQUOB, as is common in continuous query systems, events are retained in the system as long as they are needed, for as long as they participate in, say, a join operation. *Stream joins*, (or fusions [16]), join two streams together on some condition, often a time-based condition. These joins are often executed over a *sliding window*, a time interval of events in the event stream. Events are retained in the system for as long as they remain in the sliding window. RS-Algo takes as input a time interval representing the skew between two streams and maintains the sliding windows at that interval throughout the life of the stream. By maintaining the window size as a time interval, memory savings can be realized particularly for highly asynchronous streams because the space allocated for slow streams will be proportional to the slow rate of arrival.

The contributions of this paper are twofold: we provide results of an experimental evaluation we performed of the RS-Algo rate sizing algorithm under three synthetic workloads and using the dQUOB continuous query system [15]. We examine service time and memory utilization. Among our findings, we found RS-Algo to have lower overhead than anticipated, which opens opportunities for more frequent triggering. More frequent triggering can result in reduced latency between detected change in stream rate and resulting adjustment of sliding window size. We are currently validating our results under realistic workloads.

The second result chronicles a smaller experiment comparing two algorithms used during runtime to schedule events. We evaluated the performance of the existing algorithm, First Come First Serve (FCFS), against a modified version of Earliest Job First (EJF) and examined the impact on memory utilization. In [14] we hypothesized that EJF should outperform FCFS, and reasoned by means of examples. In this paper we present experimental results to confirm that hypothesis.

The remainder of the paper is organized as follows. The following section motivates our work by means of a scientific meteorology application. It also introduces dQUOB, the continuous query system that serves as the foundation for our experimental work. In Section 3, we contrast our work to related work. In Section 4, we discuss the rate window sizing algorithm and in Section 5, the performance evaluation of the scheduling algorithm. The paper concludes with future work in Section 6.

2 Motivation

The motivation for our work is cyberinfrastructure support for mesoscale meteorology research being done in the context of the LEAD [7] project. Meteorology researchers are currently lacking the infrastructure to do on-demand ensemble (or group) runs of forecasting models such as WRF [12] that are able to incorporate regional observational data about current weather conditions into a forecast model run. These current localized weather conditions are detected by regional instruments such as NEXRAD Doppler radars and sensors. Today these data products are distributed to the research meteorology community by means of general data distribution systems that stream compressed data chunks to sites distributed nationwide. The work described in this paper deals with the “last mile” issue of bringing streams of regional observational data to the forecasting models so that the right data is available to the model in a timely manner when the model needs it (that is, when the early conditions of a mesoscale phenomena are detected.)

We model the *data stream* in LEAD as an indefinite sequence of timestamped events. The dQUOB

system model conceptualizes a data stream as a relational table over which a restricted set of SQL can operate. The data streams are implemented as channels in a publish subscribe system such as EChO [8], NaradaBrokering [9], and Siena [5].

Query execution can be viewed intuitively as dropping a query into a channel. The query executes over the incoming stream and generates a result stream. Specifically, a publish subscribe application can be viewed as a directed acyclic graph, $V = \{E, G\}$ where E are edges between nodes, and G are providers, consumers, or both. An intermediate node is a consumer of input relations and a provider of output relations. Queries are dropped into intermediate nodes located in the graph, and serve to consume input relations and generate new output relations. Queries serve such functions as filtering the stream, aggregating values over a stream, or combining streams. Some of the relational operators (*i.e.*, join, temporal select, and aggregation operators) operate over entire tables in relational databases. To enable these operators to operate over indefinite sequences, we use a *sliding window* over which a query is continuously executed. We identified the sliding window in 1997 [17]; it is in widespread use today.

3 Related Work

The seminal work on continuous queries was done in the early 1990's by Terry *et.al* [19] and by Snodgrass [18]. An overview and general architectural framework for query processing in the presence of continuous data streams is discussed in [3]. The STREAM project [3] advocates support for data streams as an integral part of a DBMS. The authors developed a database management system capable of executing continuous queries over stored data streams and stored relations. The emphasis of the work is on rich language (SQL) support and assumes stream data is located in a central repository. Our work, on the other hand, assumes distributed remote streams and accepts a subset of SQL as a tradeoff for efficient query execution.

Pipelined query processing techniques to sliding window queries and algorithms for correctly evaluating window-based duplicate elimination, Group-By and Set operators are discussed in [10]. Our work focuses on improvement in memory utilization of sliding window queries in the context of dQUOB. Each quoblet in dQUOB has a single thread of execution and events are handled sequentially. Pipelined query processing may be interesting should the demands of our application merit this kind of functionality. Arasu *et. al* [1] consider memory requirements of stream queries by means of an algorithm for determining memory consumption for a query for all possible instances of the data streams. For queries that cannot be evaluated in bounded memory, the paper provides an execution strategy based on constant sized synopses of data streams. This work is complementary to ours. Babcock [2] addresses the problem of representative sample sizes when sampling streaming data. Our rate adjustment algorithm samples to determine stream rate, hence this work could potentially refine our approach. Stampede [16] investigates support for interactive stream-oriented multimedia. Though the underlying sources are data streams, much of the work in Stampede is directed toward a cluster parallel programming system that supports creation of multiple address spaces in the cluster and an unbounded number of dynamically created application threads within each address space. Stampede provides high-level data sharing abstractions that allow threads to interact with one another without regard to their physical locations in the cluster, or the specific address spaces in which they execute.

Mayur's work [6] is focused on devising an array of stream statistics over sliding windows including sum/average, histograms, hash tables, and frequency moments. Our work is not in the formulation of stream statistics, but in collecting information so that we can keep join windows in synch relative to each other and to the stream rates. Viglas [20] proposes a rate based approach and optimization framework to maximize the output rate of query evaluation plans. By estimating the output rates of various operators as a function of the rates of their input, the authors are able to identify plans with maximal rate over time and plans that reach a specified number of results as soon as possible. This work is complementary to ours and could be of value in refining our solution to asynchronous rates as addressed in this paper.

4 Rate Sizing Algorithm

Events that are retained in the system long after they are needed simply because they still exist in the sliding window of one or more join operators cause needless memory consumption. The intuition behind RS-Algo is that if the two sliding windows of a join are held at a fixed time interval they can fluctuate in size in relation to stream rates. For applications having highly asynchronous streams, such as are found in LEAD, this optimization can significantly improve memory utilization because the “slower” streams will consume less memory because fewer events are held 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. If the sliding window size is kept at 2 seconds, the query must retain in memory all events received in the last 2 seconds. For a time-based sliding window, this is 1002 events whereas for a fixed integer count strategy, 2000 events would need to be retained.

From our experience in applying dQUOB to various applications, it became apparent to us that users had difficulty reasoning about integer counts but could reason about time. For instance, a scientist knows that NEXRAD Doppler radars can have a skew of up to 2 minutes, relative to each other, so for a query that joins two Doppler radar streams, a sliding window of 2 minutes will be sufficient. The fixed size sliding window has other problems as well. A fixed join window size could be determined experimentally, but often experimental execution of an application is not practical. The rate could also be set by the user, but the risk of error is too high. If the user selects a size too large, quoblet scalability is compromised. If the size is too small, the risk of false negatives increases (*i.e.*, a query execution that should have evaluated to true instead evaluates to false because one of the participating events has fallen off the end of the sliding window and is no longer in the system.)

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 positions 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 1: Pseudocode for RS-Algo dynamic rate sizing algorithm.

RS-Algo, shown as pseudocode in Figure 1, executes over the two streams participating in a particular join operation. The two input streams are sampled for a time interval to determine respective stream rates. The new join window size is calculated based on a time interval. Thus the join window sizes for “slow” streams are allowed to grow and shrink in response to real data rates. Specifically, the algorithm accepts a single input parameter, *window_interval*, the interval of time maintained in the sliding window. The parameter is also used as the interval over which sampling occurs to determine stream rates, but a separate sampling rate could be used. The key parameters of the algorithm are as follows:

- Sampling duration – time interval over which stream is sampled.
- Sampling frequency – frequency at which re-sampling takes place to determine new stream rates.
- RS-Algo trigger – trigger condition that invokes algorithm.

For the experiments reported in this paper, the values chosen for the parameters were naive choices to enable us to better understand performance. Sampling is done continuously and any time a rate change is detected, RS-Algo is triggered.

In the remainder of this section we describe an experimental evaluation of RS-Algo using the dQUOB system [15]. We quantify the overhead of the sampling that is needed, algorithm execution time, and cost of effecting the change. Memory utilization is compared to the count based approach where join windows are of fixed size throughout execution.

4.1 Experiment Setup

The first query of our experiment is a simple query, shown in Figure 2. The query joins events from streams D and R that match on timestamp (implicit in the join operator). A query is triggered in dQUOB whenever an event of interest arrives. An arriving event is pushed to 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, starting at the gate node. Specifically, an arriving event is received by one of the event handlers, who copies the event off the socket into the event buffer. It is removed from the buffer by a dispatcher (not shown) and pushed to queries who have previously registered an interest in the event. In the example, the dispatcher passes D and R events to the gate operator of query “Q3”. A rate sampling operator, called “Sample OP”, exists on each stream that feeds the join. The rates are fed to the join operator, called “Join OP”. The algorithm computation takes place within the join operator, where new window sizes may be calculated, and window sizes may be adjusted. Filtering by the select operator removes all event pairs that do not match on ID.

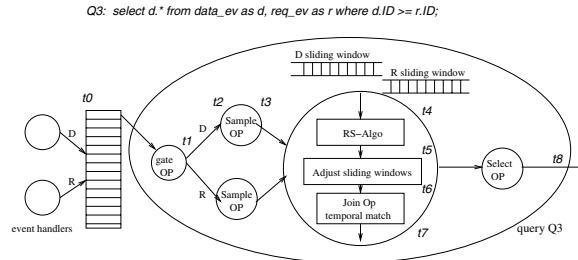


Figure 2: Simple query (Q3-simple) consisting of a join followed by a select. The query graph also details components of the RS-Algo and performance time intervals.

Figure 2 also captures the timing points used in the performance evaluation of RS-Algo. t_0 marks the instant an event is inserted into the event buffer. Once an event has been selected for scheduling at query Q3, its arrival at the gate operator marks 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. Embedded in the join operator is the rate sizing algorithm that executes periodically to determine 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. 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 second query used in the evaluation is Q4, shown in Figure 3. Q4 is a more complex, multi-join query consisting of two join, two select and two project operators. Query Q4 is used to better understand the impact of operator placement on join stream rates. That is, the join that generates $\langle M, R \rangle$ is upstream of the join that generates $\langle D, R \rangle$. What is the impact of position on the stream rates for the later join windows?

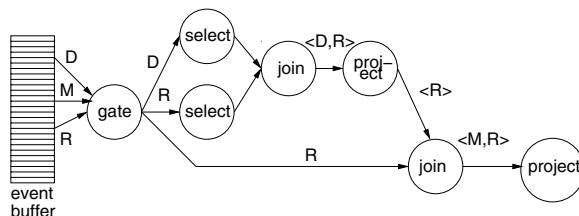


Figure 3: Q4-complex: Multi-join query

The experiment environment is a Xeon, Dell PowerEdge $2 \times 2.8\text{GHz}$ with 2 GB RAM running Red Hat Linux 9. The queries and runtime are written in C++, and queries execute as directed acyclic graphs of C++ objects. The query execution engine is layered on top of dquobEC, an in-house typed-channel publish/subscribe system written in C++ that uses PBIO [4] as its underlying binary network format. Events arriving at the quoblet are copied into a user space buffer. Events are all 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 [15]. The following are our performance metrics:

- Join window size for synchronous and asynchronous streams
- Service time, $t_8 - t_0$
- Memory utilization under rate sizing algorithm
- Overhead of sampling and rate sizing algorithm

4.2 Analysis of Rate Sizing Algorithm

We use three synthetic workloads in our evaluation of RS-Algo. All workloads execute for a total time of three minutes. Each workload consists of three streams. A stream carries events of a single type. These types are: D events which are 50 KB in size, R events which are a few bytes in size, and M events which are also a few bytes in size. The workloads are as follows:

- Case 1-async: Two asynchronous streams of constant rates. D event rate of 50 events/sec. R event rate of 1 event/sec.
- Case 2-sync: Two synchronous streams of constant rates D event rate of 50 events/sec. R event rate of 50 event/sec.
- Case 3-variable: Two asynchronous streams, one with fixed rate and one with variable rate. D stream rate of 10 events/sec. R stream rate alternates every 10 seconds between 10 events/sec and 1 event/sec.

For the analysis, we execute the three workloads against each of the two queries, Q3-simple and Q4-complex. A workload is run against a continuously executing query for several minutes. We plot the interval between second 60 and second 120 of the 3 minute run. As a point of comparison for quantifying the gains in memory utilization under dynamic rate sizing, we have implemented a version of join window processing that uses a fixed join window.

The results for the simple query under synchronous streams, Case 1-async, is given in Figure 4. The total service time of a query, shown in the topmost graph, includes two primary delays: time spent waiting in event buffer ($t_1 - t_0$), and time spent in the join sliding window ($t_7 - t_6$) before a match occurs between

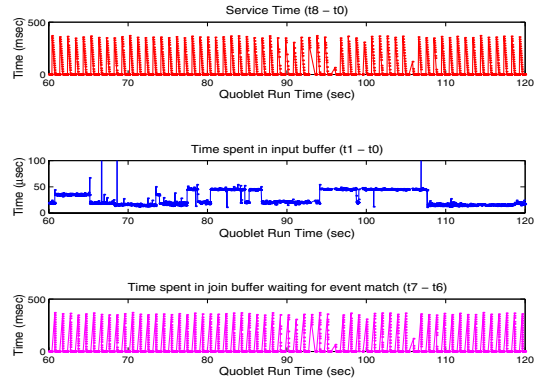


Figure 4: Case 1-async: Service time distribution for Q3-simple.

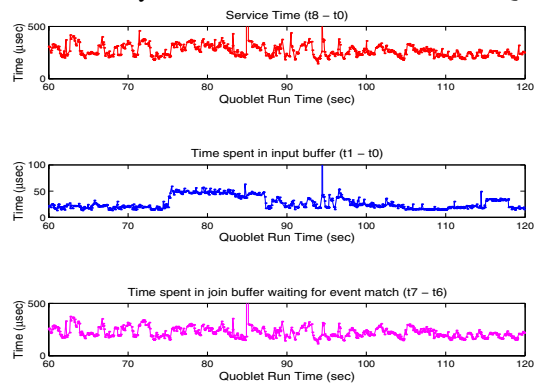


Figure 5: Case 2-sync: Service time distribution for Q3-simple.

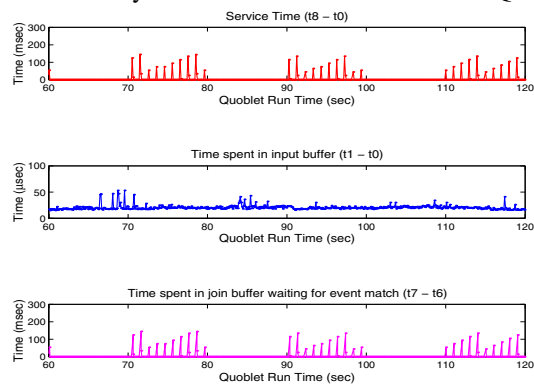


Figure 6: Case 3-variable: Service time distribution for Q3-simple.

the waiting event and a newly arrived event on the second stream. These are shown in the second and third graphs of the figure. We have observed that when the two streams are highly asynchronous but the query is in steady state, that is, the query processing time matches the interarrival rate. Time spent in the sliding window constitutes the major portion of the service time. This case is depicted in Figure 4. When query processing time is greater than event interarrival rate, then the primary delay is in the event buffer (not shown.)

The waiting time of a D event in Figure 4 depends on how soon after its arrival arrives an R event with a matching timestamp. The average service time shown in Table 1 (a) for events of type D is 70.89 ms, most of this spend in the join sliding window (70.85 ms.)

Average values	Q3	Time (ms)
Wait time in input buffer	t1 – t0	0.0267
Sampling time	t3 – t2	0.0017
Algorithm execution time	t5 – t4	0.0011
Time to resize join buffer	t6 – t5	0.0027
Wait time in join buffer	t7 – t6	70.85
Service Time	t8 – t0	70.89

(a) Time values for Case 1

Average values	Q3	Time (ms)
Wait time in input buffer	t1 – t0	0.0314
Sampling time	t3 – t2	0.0022
Algorithm execution time	t5 – t4	0.0018
Time to resize join buffer	t6 – t5	0.0033
Wait time in join buffer	t7 – t6	0.233
Service Time	t8 – t0	0.283

(b) Time values for Case 2

Average values	Q3	Time (ms)
Wait time in input buffer	t1 – t0	0.0187
Sampling time	t3 – t2	0.0018
Algorithm execution time	t5 – t4	0.0014
Time to resize join buffer	t6 – t5	0.0029
Wait time in join buffer	t7 – t6	5.70
Service Time	t8 – t0	5.73

(c) Time values for Case 3

Table 1: Summary of service time under 3 loads. RS-Algo overhead is sum of “sampling time”, “algorithm execution time”, and “time to resize join buffer.”

The results for the simple query under synchronous streams (Case 2-synch) is given in Figure 5. Note that in this case the Y axis of service times and join buffer times are in microseconds instead of the millisecond unit of Figure 4. As graphed in Figure 5, top graph, and as shown as an average in Table 1 (b), service time is minimal under synchronous streams for all cases tested. The waiting time in the input buffer is longest of all cases tested, and this is due to the high rate of both streams.

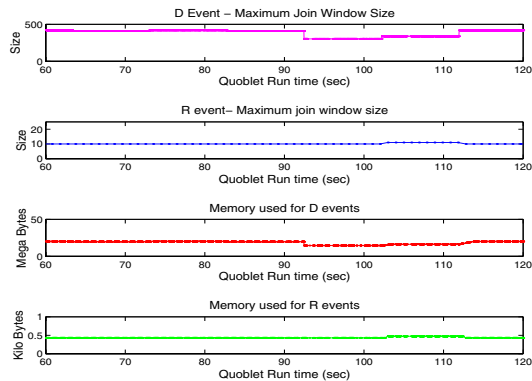


Figure 7: Case 1-async: Memory utilization and join window size for query Q3.

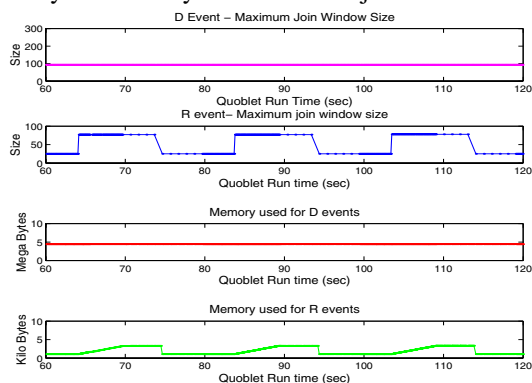


Figure 8: Case 3-variable: Memory utilization and join window size for query Q3. The stream rates are asynchronous and constant and the difference in rates is large.

Figure 6 plots the service time for Q3-simple under the final workload, Case 3-variable. The 10 second periodicity in the stream rate of R is evident in the average time D events wait in the join buffer (see lower graph.) Specifically, D’s average time in the join window increases when the rate of stream R drops to 1 event/second.

The overhead imposed by the RS-Algo algorithm can be seen by examining Tables 1 (a), (b) and (c). Overhead attributable to the algorithm is the sum of sampling time, algorithm execution time, and time to resize buffer. The average overhead for RS-Algo taken over the 3 three synthetic workloads is 0.05 ms. Taken as a fraction of total service time, algorithm overhead varies between 0.007% and 2.6%.

Memory utilization is plotted in Figures 7 and 8. The join window size for streams D and R are plotted in the top two graphs of each figure. As is evidenced by Case 3-variable, the rate sizing algorithm responds almost immediately to the change in stream rate. This is desirable, and coupled with the fact that overhead of the algorithm is fairly low, presents the opportunity for frequent triggering of the algorithm. We are currently working on a realistic workload based on LEAD stream rates that will allow us to explore this further.

Improvements in memory utilization achieved by RS-Algo can be most easily seen by comparing the maximum join window size for D to the join window size for R in Figure 7. If a integer count had been used, the plotted lines of the top two graphs (D and R join window size) would be at 500. But instead, the plot of R stays around 10 events. This difference is the memory savings. For Case 1-async the savings is 97.98%. For Case 3-variable where the streams alternate between synchronous and asynchronous with

respect to each other, the savings is less, 57.73% over the fixed window size solution. Savings in terms of bytes of memory is less meaningful because our workload sets the slow stream as the stream of small events (256 bytes). Had the “slow” stream contained large events, as would often occur in practice, the savings in terms of bytes would be more significant.

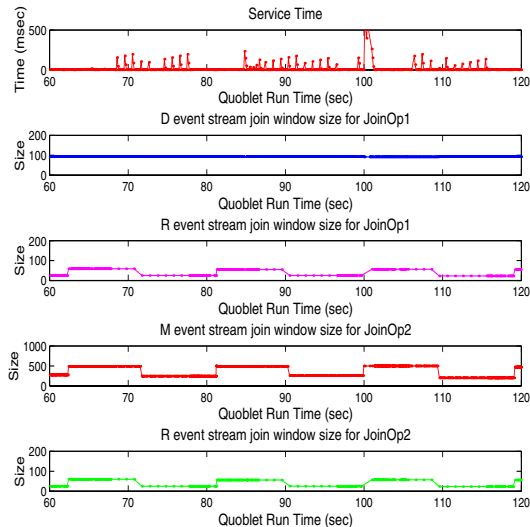


Figure 9: Examination of rate for upstream join, JoinOp2, compared to downstream join, JoinOp1 using variable workload, Case 3-variable.

Finally, in Figure 9 we examine the impact of join operator placement within a query on the size of the sliding windows for that join operator. Plotted are the service time and the join window size for query Q4-complex under the variable workload, Case 3-variable. Plotted are join window size for the two join operators of query Q4. Referring back to Figure 3, JoinOp1 refers to the downstream operator that generates $\langle D, R \rangle$ whereas JoinOp2 is the upstream operator that generates $\langle M, R \rangle$. In comparing the two graphs for R, one sees little difference. This is to be expected since both streams are still at “wire speeds”. The graph for M on the other hand, reflects the rather substantial impact of downstream operators on stream rates.

5 Event Scheduling

We showed by means of an example in [14] that overall query processing time (service time) can be improved by employing an Earliest Job First (EJF) scheduler at the event dispatcher instead of the currently employed First Come First Serve (FCFS). FCFS performs well when streams are synchronous and non-bursty because it processes events in the order of arrival. But under asynchronous streams, particularly when one of the streams suffers a delay, events from the faster stream will block in the query and accumulate, waiting for an event from the slower stream to arrive. Earliest Job First (EJF), which selects from the input buffer the event with the earliest timestamp, performs better under these conditions.

We measure the elapsed time between the arrival of an event at the stream processing server (called a *quoblet*) and the recognition of that event. However, often the behavior to be recognized occurs over distributed remote sources in which case recognition time includes the arrival time of all the events that make up a particular incident. The performance metrics used to evaluate the scheduler algorithm are service time and average service time. *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.

A *query* is an acyclic directed 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. The workload used to evaluate the scheduling algorithms consists of three synthetic data streams, that is, streams that carry data events of a single type as follows:

- D events - virtual memory and CPU statistics for a host resource. Large events of 50KB in size.
- R events - user request events that filter D events. Small events of few bytes in size.
- M events - used to simulate additional workload at the quoblet. Small events of a few bytes in size.

The quoblet executes two black-box queries whose inputs and outputs are shown in Figure 10. Execution of query Q1 upon the arrival of a D event results in one tuple, $\langle D, R \rangle$, being generated. Q1 contains a single user-defined function that has an execution time of approximately 100 milliseconds. Execution time for the query not including the user defined function is a few milliseconds. The second query, Q2, performs a simple select on arriving events. Selectivity of the select operator is 1.0. It executes a user defined function of 100 milliseconds in duration. Local ordering is preserved within a stream as a guarantee of the underlying publish/subscribe system. Runs are conducted on a local area network in order to obtain better control over rates and delays.

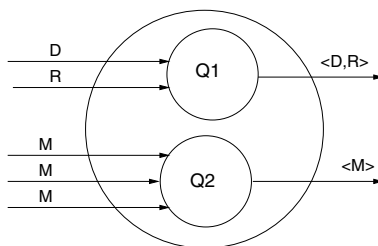
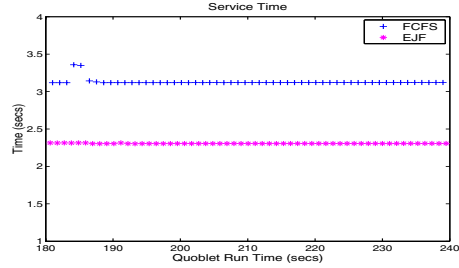


Figure 10: Input/Output behavior of the two queries used in quoblet.

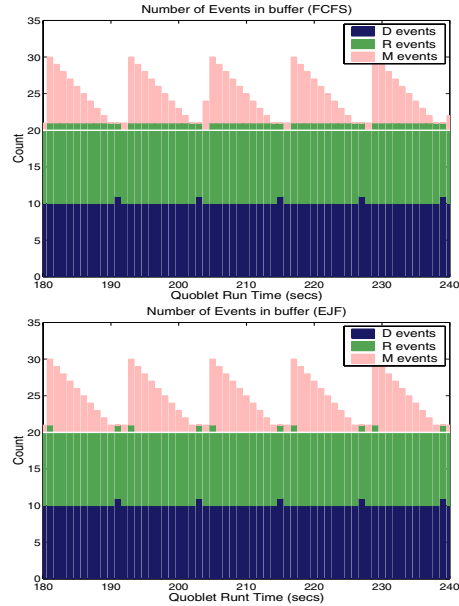
In our experimental setup for EJF, join window size is set to 10 events. That is, under steady state conditions a maximum of 10 events will be held in the history buffer of any join operator. D events arrive at a rate of 1 event/sec. R events also arrive at the rate of 1 event/sec but are delayed by 2 seconds. Total runtime of a quoblet is the time period for which input streams are supplied to the quoblet. The total quoblet runtime in our experimental evaluation is 5 minutes. To display the finer details, the graphs show data collected from a snapshot of 1 minute duration.

Intuitively, when applied to scheduling events for stream joins, EJF will provide overall reduced service times compared to FCFS when streams are asynchronous due to factors such as delays in transit, such as exist in WAN environments, and the quoblet is under sufficient load so that queuing of arriving events occurs. For example, suppose observational and model events are generated at same rate, but network temporarily partitions and the observational events are delayed in transit. During the delay events from another source arrive with timestamps later than delayed observational event stream. Then under EJF when the partition is repaired, delayed events will be processed the moment they become available to the scheduler.

EJF scheduling obtains a minimum overall service time by scheduling a delayed R event as soon as it arrives and does so by selecting the earliest event from a set of ready events having the smallest timestamp value. Figure 2 (a), compares service time for Q1 under the quoblet configuration depicted in Figure 10, using FCFS and EJF policies.



(a) Service time for query Q1.



(b) Number of events for query Q1.

Figure 11: Service time and number of events in memory for the quoblet executing query Q1 under FCFS (top) and EJF algorithms (bottom).

Table 4 shows the average service time for query Q1, using FCFS and EJF scheduling as shown in Figure 11 (a). Across the cases we tested, service time improvement was 25% to more than 50%. Particularly, when user defined functions impose larger load on the quoblet, or the rate of M streams were higher, EJF produces best improvements in service time.

Figure 4 (b) demonstrates that in the case of FCFS, R events are waiting while M events are serviced. This is visible by examining the number of times where R events appear above the white line that appears at count 20 on Y axis. In EJF, M events are made to wait when delayed R events arrive and are serviced. This is evidenced by the fact that there are only a handful of times where R events appear above the white line for EJF in Figure 11(b).

6 Conclusions and Future Work

Continuous query systems are an intuitive way for users to access data streaming data in large scale scientific applications containing hundreds if not thousands of streams. This paper focuses on optimizations for improved memory utilization under the large scale, asynchronous streams found in on-demand meteorolog-

Description	FCFS	EJF
Average service time (sec)	3.13	2.31
Average number of D events in quoblet	9.94	9.94
Average number of R events in quoblet	10.59	9.87
Average number of M events in quoblet	3.87	4.60

Table 2: Service time and number of events. Average values under the two scheduling algorithms.

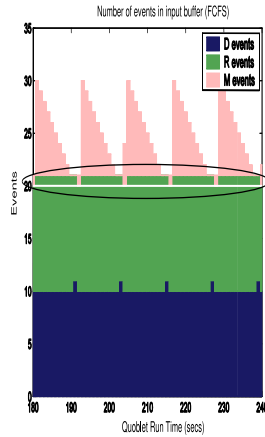


Table 3: Detailed blowup of number of events in input queue under FCFS scheduling. Key difference is number of 'R' events in the circled region.

ical 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. Improvements in memory utilization attained through use of the RS-Algo algorithm reach a 97.98% reduction in the space required for the slower stream of an asynchronous pair. Where the streams alternate between synchronous and asynchronous with respect to each other, the savings is less, 57.73% over the fixed window size solution. Our current effort is focused on developing a realistic workload based on our knowledge of LEAD streams for use in further evaluating RS-Algo.

Our results have shown that FCFS and EJF are comparable when stream delays are rare and steady state at the quoblet is the norm. But EJF has a slight performance edge when streams are highly asynchronous and a quoblet is under load.

In addition to developing a realistic workload based on meteorological data streams, our future work is focused on refining values for the key parameters of the rate-based window sizing algorithm. Specifically,

Sample duration. Sample duration is the time duration over which a stream is sampled. In Shapiro [13] Kolmogorov's statistic is for estimating selectivity by means of sampling data. Since we are only interested in determining stream rate, a simpler metric than selectivity, our sample size should be smaller.

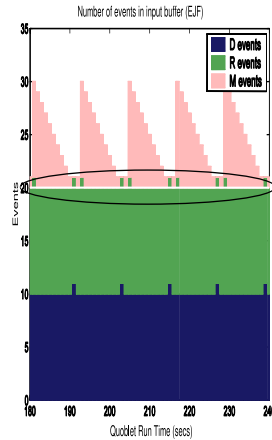


Table 4: Detailed blowup of number of events in input queue under EJF scheduling. Note that under the number of 'R' events in the circled region is smaller for EJF than it was for FCFS.

Selecting sampling frequency. Sampling frequency is the frequency at which re-sampling is done to determine new stream rates. Our experimental evaluation revealed that stream sampling is cheap, on the order of 1 - 2 microseconds, hence we can run it frequently if needed.

Rate Sizing algorithm trigger. The rate sizing algorithm trigger invokes execution of the rate sizing algorithm that adjusts the join window sizes for the two streams participating in that particular join. The algorithm may be sensitive to abrupt and drastic changes to streams of MB events. We are working on capturing that scenario with additional workloads.

References

- [1] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 221–232. ACM Press, 2002.
- [2] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.
- [3] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. In *International Conference on Management of Data (SIGMOD)*. ACM Press, 2001.
- [4] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Supercomputing 2000*, November 2000.
- [5] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
- [6] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms(SODA)*. ACM Press, 2002.
- [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] Greg Eisenhauer. The ECho event delivery system. Technical Report GIT-CC-99-08, College of Computing, Georgia Institute of Technology, 1999. http://www.cc.gatech.edu/tech_reports.
- [9] Geoffrey Fox and Shrideep Pallickara. An event service to support grid computational environments. *Journal of Concurrency and Computation: Practice and Experience. Special Issue on Grid Computing Environments.*, 2002.
- [10] Moustafa A. Hammad, Walid G. Aref, Michael J. Franklin, Mohamed F. Mokbel, and Ahmed K. Elmagarmid. Efficient execution of sliding window queries over data streams. Technical Report Technical Report CSD TR03-035, Computer Science Department, Purdue University, 2003.
- [11] 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.
- [12] 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 Zwiefelhofer 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.
- [13] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *ACM SIGMOD Conference*, pages 256–276. ACM Press, June 1984.
- [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, Edinburg, Scotland, August 2002. IEEE Computer Society.
- [15] 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.
- [16] Umakishore Ramachandran, Rishiyur Nikhil, James Matthew Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede: A cluster programming middleware for interactive stream-oriented applications. 14(11), November 2003.
- [17] 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.
- [18] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems(TOCS)*, 6(2):157–195, 1988.
- [19] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *International Conference on Management of Data (SIGMOD)*. ACM Press, 1992.
- [20] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. ACM Press, 2002.