

Leveraging Run Time Knowledge about Event Rates to Improve Memory Utilization in Wide Area Data Stream Filtering

Beth Plale

Computer Science Dept.
Indiana University
Bloomington, IN 47405-7104
plale@cs.indiana.edu

Abstract

The dQUOB system conceptualization of datastreams as database and its SQL interface to data streams is an intuitive way for users to think about their data needs in a large scale application containing hundreds if not thousands of data streams. Experience with dQUOB has shown the need for more aggressive memory management to achieve the scalability we desire. This paper addresses the problem with a two-fold solution. The first is replacement of the existing First Come First Served (FCFS) scheduling algorithm with an Earliest Job First (EJF) algorithm which we demonstrate to yield better average service time. The second is an introspection algorithm that sets and adapts the sizes of join windows in response to knowledge acquired at runtime about event rates. In addition to the potential for significant improvements in memory utilization, the algorithm presented here also provides a means by which the user can reason about join window sizes. Wide area measurements demonstrate the adaptive capability required by the introspection technique.

1 Introduction

The dQUOB system [15] leverages the popularity, time tested usefulness, and unarguable efficiency of SQL queries through its application to data streams found in wide-area, data intensive, distributed and parallel scientific applications such as [2, 5, 8, 14, 17]. The system model conceptualization views a set of data streams as a relational database. This *data streams as database* abstraction enables a user to request data from the temporal event streams by means of SQL queries. The execution model can be viewed as a set of continuously executing queries that are logically grouped and

selectively placed in data streams at the data source, at the recipient, or at points between. This conceptualization provides an intuitive way of thinking about streams, about combining streams and about creating new views of the data. The implementation is lightweight SQL query execution middleware that does not require nor operate over a database management system. The dQUOB system is built on top of the publish/subscribe event channel implementation, ECho [6].

Our experience with using dQUOB to manage the big-data event streams that occur in grid computing has given us the perspective to identify meaningful performance optimizations. Specifically, it became obvious that due to the large size of the events traditionally found in scientific applications (200K - 2 MB), good memory utilization is key to overall performance and scalability. Since memory utilization is directly related to the number of events that linger around waiting to participate in joins, efforts at optimization need to focus on the join processing.

Join processing in our domain is the act of combining two event streams based on one of two notions of time. Since events pass through the query evaluator in real time, the evaluator must make instant decisions as to whether to accept or discard an event. This is easy for queries containing simple select statements. But because useful queries contain more complex operators like joins and temporal selects, and further, streams are not often synchronous because of issues like clock skew and delayed generation rate, some events must be retained. This paper deals with the question of the number of events to retain. Specifically, how large should be the join window?

A *join window* is similar to a sliding window; and determines the size of the event sequence that is retained in memory for a particular join operation. In the earlier system, the join window size was specified by the user at

startup and it applied uniformly to all event streams and join operations. But the solution was flawed because the cost of a wrong guess is simply too high. Too small a window increases the likelihood of false negatives; too large a window consumes memory, impedes throughput, and limits scalability.

Our solution for improved memory utilization is two-fold. The first focuses on improving the service time of a query, that is, reducing the time interval from the arrival of the last event of a set of events required by the query to when the query generates a result. We show average service time can be reduced by replacing the existing first come first served (FCFS) scheduling algorithm with an Earliest Job First (EJF) algorithm.

The second optimization focuses on a more informed selection of join window size through an introspection technique. The existing one-size-fits all approach is too prone to error, or more likely, to huge inefficiencies in memory utilizations. Instead, we developed an algorithm for setting and adapting the join window size based on run time information about the data stream rates. Through event stream sampling, we estimate initial window sizes, then through an adaptive scheme employing exponential averaging, we monitor arrival rates and adjust the join window sizes as appropriate. Because optimized window sizes are computed from stream information alone, the decisions are global to the quoblet¹ then only enacted locally at the queries. The solution accommodates differences in event generation rates such as would occur when a stream is slow in starting up or with a low arrival rate.

The abstract proceeds with a definition of the several notions of time typically encountered in streaming applications and of a characterization of query cases. The potential for performance improvements in service time of an EJF scheduling policy is given as a proof by example in Section 3. Section 4 introduces the algorithm for rate sensitive join window size prediction. Section 5 provides performance measurements that demonstrate the ability of a small set of query rules to detect and adapt to changes in the data streams. Since the rate sensitive join window algorithm is implemented using the query mechanics, the measurements serve as early, partial proof of the algorithm. Measurements that specifically quantify memory utilization savings are in progress. The wide area measurements in and of themselves reveal interesting differences between LAN and WAN performance for data streaming applications. The database community has recently taken renewed interest in continuous queries of the kind that form the heart of dQUOB. This and other related work is discussed in

¹A *quoblet* is a logical grouping of queries that constitutes an executable unit.

Section 6. The paper concludes with a brief summary and future work in Section 7.

2 Time Notions and Query Cases

For purposes of discussion, a *logical timestep* is the application's notion of time. A global atmospheric model that begins its simulation with ozone concentrations from September 01, 1992 at 1200 hours, assigns Sept 01, 1992 1200 hours as the logical timestep of its first timestep. The *timestamp* then is the CPU clock time at which some event, such as the output of a logical timestep, occurs. Finally, *generation time* is the instant at which an event is pushed or streamed from its source. Trace data is by its nature used as a data source post mortem; its timestamp records information about when it was created; generation time captures the time at which a trace file is being used post mortem. We make the simplifying assumption that timestamp and generation time are the same. Our grounds are that the meaningful time values for decision making in a streaming application are the logical timestep, required to tie events back to the application time domain, and current time, required to react to changes in the current environment.

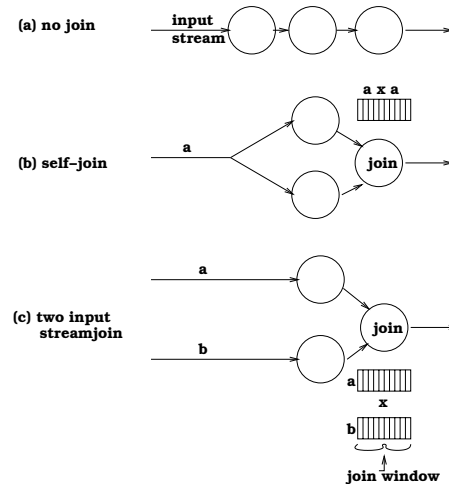


Figure 1. Basic join query types.

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. As shown in the basic cases of Figure 1, events enter the graph at the left and exit at the right. In the simplest case ((a) in the figure) a query contains only selects and projects. It accepts a single event stream and either filters out events, or transforms and forwards

them. A node with two incoming edges such as is shown in (b) is a join node. Joins are a Cartesian product with a time-based condition. Intuitively, two events satisfy a join if they “happen at the same time” where time is either logical time or timestep time. The query (b) contains a self-join, that is, a join that performs a Cartesian Product over one event stream, that is, over the events in the join window ‘a x a’. The third query case, (c), defines a join over two event streams, ‘a’ and ‘b’, or more precisely, over the join windows of ‘a’ and ‘b’. It is this latter class of queries to which our work described in this paper applies. In the presence of asynchronous streams, the window size of a slow stream, say ‘a’, could be made considerably smaller than that for a fast stream, say ‘b’.

3 Event Scheduling Algorithms

Evidence suggests that query service time, and hence memory utilization, can be better served by an Earliest Job First (EJF) scheduling algorithm instead of the currently employed FCFS algorithm. FCFS serves well for streams that are reasonably synchronous and non-bursty. But when event generation rates vary, latencies are large (as in a WAN environment), and traffic patterns are bursty, it is our observation that events from the faster stream will block in the join and accumulate, waiting for an event from the slower stream to arrive. In this section we examine the cost benefit of using an EJF scheduling algorithm.

The event scheduler is an asynchronous thread that selects the best event from the recently arrived events and dispatches it to the queries that have registered an interest in the event type. Concurrent arrivals are handled by channel-specific handlers in the quoblet. The handler retrieves the event off the TCP socket and stores it to a global queue. The event scheduler then is a global scheduler that is executed upon the disposition of every event, in order to select the next event for processing.

The overarching goal of the dQUOB system is to recognize incidents that occur in a distributed system, and to do so as soon as possible after their occurrence. An *incident* is loosely defined as a set of events that occur at the same time that when taken together describe the incident or its preconditions. A new nuclear meltdown is an incident; the causal events are stuck valve, higher than normal pressure in the boiler, rising ambient air temperature, etc. There is little a quoblet can do to reduce wide area latencies and differences in generation rate that will delay incident recognition, but the quoblet can minimize service time. So the metric we establish to evaluate event scheduling algorithms is service time. *Service time* is traditionally defined as the elapsed time between the arrival of a request at a server and the sat-

isfaction of that request by the server. The service time for incident recognition, which differs slightly in that it is dependent upon input from multiple sources, is the elapsed time between the arrival of the final event of the incident and the generation of the tuple that recognizes it.

We claim that an Earliest Job First (EJF) scheduling policy minimizes overall service time by giving preference to events occurring earlier in time. The notion of earliest is determined by selecting the next event from a set of ready events having the smallest timestamp value. This means the recognition of these complex, multi-source conditions does not suffer delay by a policy that favors newer but unrelated, and hence, irrelevant events. We defend our assertion with an example, but are in the process of evaluating performance of both approaches.

To illustrate, we compare the service time for FCFS and EJF over a sample set of events and a sample quoblet consisting of two simple queries. In our example, see Figure 2 (a), events arrive in one second intervals (*i.e.*, the inter-arrival rate is an integer multiple of one second,) query processing time is one second long, and event generation is instantaneous. At time t_3 three events are received: a_3 and c_3 which are timestamped t_3 (assuming instantaneous arrival) and d_0 which is timestamped t_0 (and was delayed in transit). The two queries are named AB and CD, see (b). Each query consists of a single join operation; query AB joins event streams ‘a’ and ‘b’ while query CD joins ‘c’ and ‘d’. The service time for the incident described by c_0 and d_0 is the elapsed time between the arrival of d_0 and the generation of the tuple pair $\langle c_0, d_0 \rangle$. Under FCFS scheduling, generation of the tuple pair $\langle c_0, d_0 \rangle$ occurs at the beginning of timestep t_9 , see (c). Under EJF, generation occurs at the beginning of t_4 implying that d_0 was serviced immediately upon arrival. The average service time for the example under the two scheduling policies is:

$$FCFS_{avgSvcTime} = (6+8+10+11)/4 = 8.75sec/event$$

$$EJF_{avgSvcTime} = (1 + 1 + 1 + 2)/4 = 1.25sec/event$$

The example illustrates the viability of EJF scheduling in a burst condition, that is, when server processing has fallen behind. The results hold under steady state conditions as well, though are less pronounced.

A weakness of the EJF algorithm is the potential for starvation. We argue that given the time sequence nature of the objects to be scheduled, starvation is not possible. Suppose an event α arrives at the scheduler and experiences starvation. Because events are timestamped and scheduled earliest timestamp first, this means that every event already at the scheduler has a timestamp ear-

lier than α . Further, every event that arrives after α also has an earlier timestamp than α . Because timestamp is equal to generation rate, every event but α must have been generated before α . That means α is the last event generated by the application, and as such will eventually be scheduled.

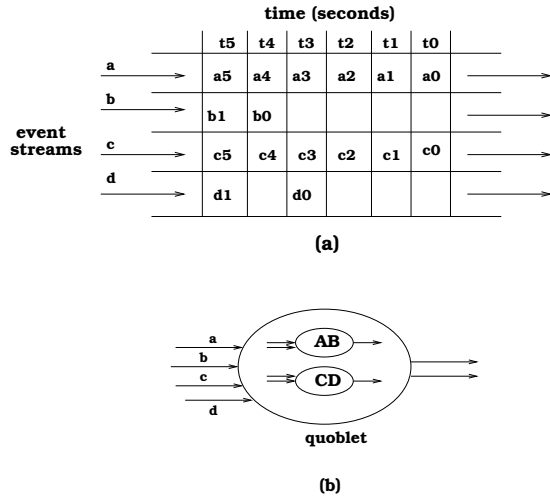


Figure 2. Event service times under First Come First Serve (FCFS) and Earliest Job First (EJJF) scheduling policies.

4 Rate Sensitive Join Windows

Experience with the dQUOB system in scientific computing where megabyte events are not unheard of, has highlighted the need for improved memory utilization through intelligent allocation of memory for join windows. We address the problem with an introspection technique. Introspection is an architectural paradigm that mimics adaptation in biological systems [10]. It can be viewed as a continuous cycle of computation – observation – optimization. Computation is normal operation. Observation and optimization, on the other hand, moni-

tor computation, detect behavior through analysis of observations, then effect a response that then becomes part of the next computation.

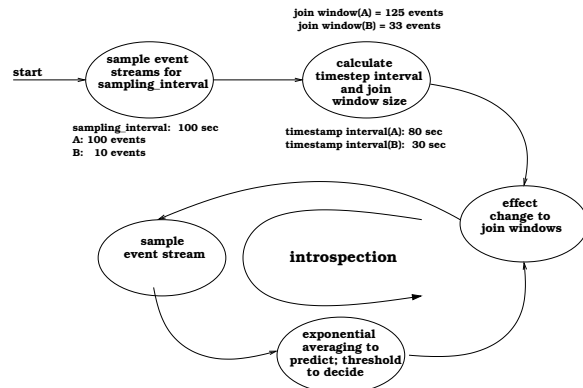


Figure 3. Rate Sensitive Join Window Size Algorithm.

Our overall goal is to maintain join windows at a size that can be expressed as a timestamp interval. By doing so, two gains are had. First, it positions the user to better reason about the trade-off between performance and increased incident of false negatives. For instance, if the user knows the application will be running in a WAN that typically experiences latencies on the order of several minutes, then a larger starting value can be selected. Second, we achieve window sizes that mirror differences in stream rates almost as a byproduct. Unfortunately, timestamp information is often not available at application startup, so we let the user specify an interval in wallclock time instead, then map the interval into the timestamp domain during initial stream sampling.

We enact the algorithm as two phases: a startup phase where initial reasonable join window sizes are established, followed by ongoing inspection to adjust. This two-phase cycle is depicted in Figure 3.

Determination of initial join window sizes is shown as the first three states of Figure 3 and in more detail in Figure 4. The algorithm accepts a single input parameter, *sampling_interval*. *sampling_interval* is a time interval defined by the user, and determined by the user thinking about the potential skew that might exist between application streams. Where latencies are long, an interval of 300 (seconds) might be reasonable. The algorithm samples over this interval, though it would be trivial to sample over a longer period. It samples all event streams concurrently. Upon completion the timestamp interval for every stream is computed from the first and last timestamps received during the sampling interval. The barrier synchronizes the threads before the maximum timestamp interval is determined; this last

step identifies the fastest stream. The join window size is then computed for each stream relative to the fastest stream.

```

at_startup(sampling_interval: integer) {
  for all i concurrently {
    sample event stream[i] for duration of sampling_interval;
    barrier();
    max_timestamp_interval = last_event[i].timestamp -
      first_event[i].timestamp;
    join_window_size[i] = (events_received[i] * sampling_interval)
      / max_timestamp_interval;
  }
  effect_change[i];
}

```

Figure 4. Pseudo code for join window algorithm.

To illustrate, shown in Figure 5 are two input streams A and B having synchronous arrival rates. Suppose the user specifies a sampling interval of 30 seconds. Event arrival times are shown across the top for the 30 second sampling interval. During the interval, 11 events arrive on A and three arrive on stream B. Timestamp_interval for streams A and B are 10 seconds and 2 seconds respectively. There is obvious latency in both streams, but A is selected as max_timestamp_interval; that is, A is the fastest stream. The join window sizes for A and B are then calculated at 33 and 9 respectively. For the fast stream, A, a window size of 33 means the window is sized to hold 30 seconds of events in timestamp time. The slow stream window size of nine represents an equivalent time interval relative to the fast stream. The window sizes reflect the ratio of events received to total sampling time; the smaller the ratio, the smaller the join window. The purpose of the timestamp interval computation is to map the users wallclock notion of time into a time domain more closely tied to the application.

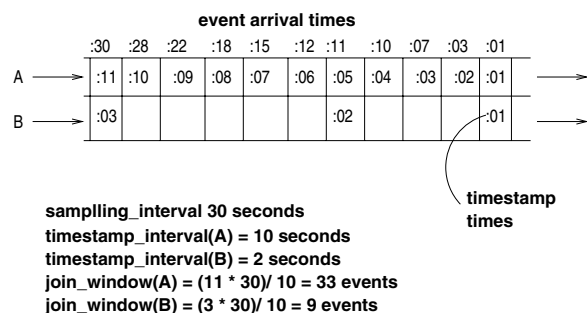


Figure 5. Example for Join Window Algorithm.

The join window algorithm is based on the assumption that the interval expressed in the timestamp domain will always be smaller than or equal to the interval in the sampling domain. For instance, in Figure 5 we sampled over a 30 second interval, and both event streams had intervals less than 30 seconds. But suppose a source were to advance timestamp its events, that is assigns timestamps that are valid in the future. It may do this in order to deliver its event stream and retire. However, our earlier assumption that generation time is the same as timestamp precludes this kind of behavior.

Enacting a change in join window size at the individual queries requires the following conditions be met:

- Events belonging to the same stream are monotonically increasing in timestamp,
- When a join window is reduced in size, the events that are removed are the oldest events,
- While a window size adjustment is being undertaken, no query evaluation can be taking place (for the query in question), and
- Events flushed from a join window cannot automatically be freed as they may still be participating in partially complete results within the query.

Monotonically increasing events are guaranteed by the underlying event channel layer. An event stream in the dQUOB middleware maps to an event channel in the communication layer, and event channels have a single source. Flushing oldest events first is accomplished efficiently because oldest events reside in order at the head of a linked list. Effecting a change while the query is quiescent is accomplished with existing query support for management-level commands. A query services management commands during quiescent times. Parallelism exists between queries but not within a query, so when a query is servicing a management command, it does so sequentially. Preserving partial results is achieved through reference counts to ensure an event is not released (destroyed) until its reference count goes to zero.

The ongoing *introspective phase* of the algorithm employs exponential averaging to predict the next join window size based on past observation. The averaging algorithm makes its predictions by looking for changes in the arrival rate of the stream. Specifically the exponential averaging function

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n$$

is defined in terms of T_n as the number of events received over the most recent interval determined by sampling_interval. S_1 is the number of events received over the initial sampling interval, and α is expected to be large (i.e., ≥ 0.8) giving weight to more recent observations.

In order to abate constant minor adjustments to join window sizes, the transition to the 'effect change' state of Figure 3 is taken only when the predicted value exceeds a threshold. Omitted for brevity is a discussion of how threshold is determined and its impact on performance. Omitted also is a discussion of effecting the join window size change; particularly the consequence of a shrinking window.

Introspection is implemented with existing mechanisms, that is, by creating additional queries with the sole task of housekeeping. The startup phase and introspective phase can be implemented in two queries, one per phase. The ability of queries to implement simple finite state machines, and effectively transfer from state to state, is demonstrated albeit briefly in the next section. Exponential averaging leverages a new extension to the dQUOB SQL language; namely, support for user-defined functions invocable from within the query.

5 Adaptivity and Wide Area Network Measurements

Our first experiment tests the adaptivity of queries in the presence of changes in the underlying execution environment. We push a workload of 540 75K events from a source to a sink through a quoblet filter. We vary the filtering strength of the quoblet at points in the workload and look at the impact on end-to-end latency as measured between source and sink. The test was run on a LAN in a controlled setting (*i.e.*, quiescent network, lightly loaded machines) on several Sun Ultra 30 247 MHz workstations running Solaris 7 and connected via 100 Mbps switched FastEthernet.

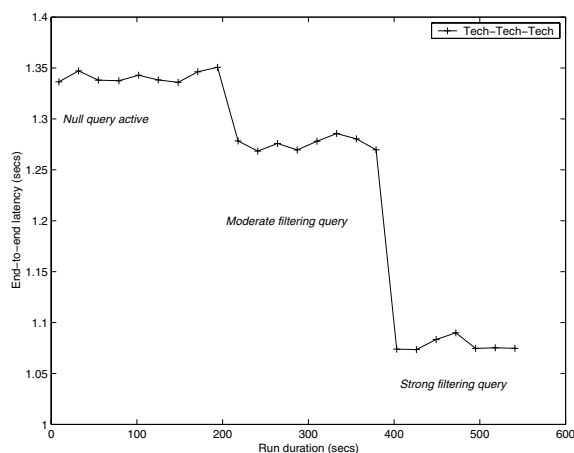


Figure 6. Quoblet potential for responsiveness to changes in environment.

We see in Figure 6 that end-to-end latency is responsive to filtering strength. More importantly for our discussion, the behavior in Figure 6 is implemented as three queries. The phase transitions at roughly 200 and 400 seconds are query responses to events from the underlying network monitoring infrastructure. The graph demonstrates the ability of queries, if specified correctly, to implement a small finite state machine. This is important because our join window algorithm leverages this behavior.

We then move the same experiment to a WAN and test with the quoblet at the source and at the sink. We compare the WAN results to the baseline measurement from the LAN measurements. The latter appears as the '-x-' plot along the bottom of Figure 7. The hosts are a Sun Ultra 30 at Georgia Tech and an 8 processor Onyx 2, R10000 running IRIX64 6.5, at the Albuquerque High Performance Computing Center (AHPCC).

We anticipated worse behavior when filtering is done at the sink, the '-+-' plot, and certainly saw it in terms of longer end-to-end times, but what was unexpected and reoccurred repeatedly is the wide variations in latencies at the data points we plotted. We are currently repeating these experiments in a more iterative fashion (growing out from LAN to IUB -> IUPUI WAN, then beyond), but from these measurements we obtained the interesting result that pushing the quoblet closer to the source results in performance that more closely matches LAN behavior.

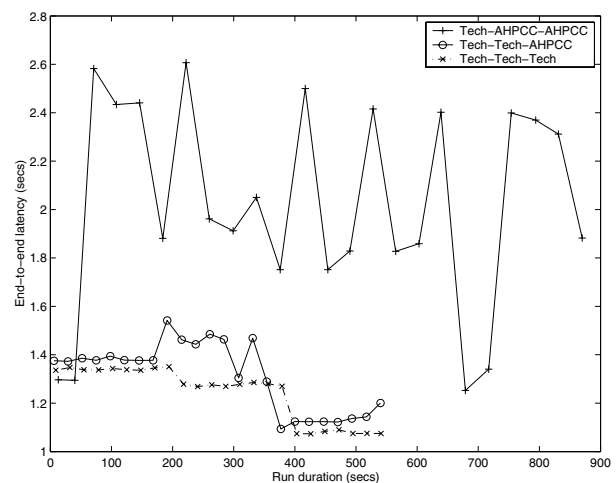


Figure 7. WAN measurement (Georgia Tech -> AHPCC)

6 Related Work

Seminal work on continuous queries was done in the early 1990's by Terry *et. al.* [19] and by Snodgrass [18]. *Continuous queries* are conventional database queries that are issued once and henceforth run continually over the database.

The STREAM project advocates continuous queries to support data streams as an integral part of a DBMS. The authors in [4] define an architecture for processing continuous queries and identify a number of research topics in the area including a processor for continuous queries. The STREAM work focuses on rich language (SQL) support and assumes streams and queries are directed to a single location. Our work stresses portability and distribution of query processing points and accepts a subset of SQL as a tradeoff for efficient query execution. TAPESTRY [19] defines the class of queries that work for continuous semantics: monotone queries. Intuitively a monotone query is a query in which nothing can be later undone in a database that will invalidate earlier results. The starting point of our work is a subset of SQL that yields monotone queries. The

Fjords data-flow architecture [12] focuses on a similar problem of database queries to process streaming data from networks of ad hoc collections of sensors. The authors define a new type of join, the zipper join, to meet the needs of combining event streams by "zippering" together streams according to a time interval. The join is applicable when streams are synchronized. However, we have discovered in our work that synchronized streams are the exception rather than the norm.

Eddies [3] executes queries over distributed information resources, such as massively parallel database systems. It shares basic technical ideas with our problem, although the general approach is different. Eddies uses an adaptive query engine to process conventional (one-time) queries efficiently under unpredictable environments. Ours is a continuous query approach. The Active Data Repository [7] evaluates SQL-like queries to satisfy client needs. However, queries are evaluated at the source over a physical database. This contrasts our work which evaluates queries over datastreams, providing portability for queries and support for queries over streams that originate at different sources. The Continual Queries system [11] is optimized to compute the difference between current query results and prior results. It then returns the delta of the two queries. This approach complements our work, which is optimized to continuously execute and return full results of a query in a highly efficient manner.

In parallel and distributed computing, temporal SQL has been used for performance analysis of distributed

and parallel computations [18, 13, 9]. Queries are issued one-time against the post-mortem performance data [18]. Our goal, on the other hand, is analysis at run-time, which puts greater demands on efficiency. Further, by pushing analysis into the data stream [13], one can reduce the amount of data that must be ultimately stored, and the period over which it must be stored.

Run-time detection in data streams has been addressed with fuzzy logic [16] and rule-based control [1]. We argue that the more static nature of these approaches make them less able to adapt to changing user needs and changes in data behavior.

7 Conclusion

The dQUOB system conceptualization of *data streams as database* provides an intuitive way for users to think about their data needs in a large scale application containing hundreds if not thousands of streams. This paper focuses on optimizations being undertaken to address a performance and scalability bottleneck identified in the system, namely, memory management. Our solution is two-fold: a more suitable event scheduling algorithm, and an algorithm that leverages run-time information knowledge about event rates using variable length join windows.

An important focus for future work is to extend the join window size line of reasoning to include probability assessment. As mentioned in the introduction, too small a join window increases the likelihood of false negatives while too large a window consumes memory, impedes throughput, and limits scalability. Our goal is to explore the assignment of a probability of false negatives for a chosen window size. We can then, for example, respond to a window size setting of 1 second with a warning such as "Beware, the likelihood of false negatives for a 1 second window size is 90%."

References

- [1] A. Afjeh, P. Homer, H. Lewandowski, J. Reed, and R. Schlichting. Development of an intelligent monitoring and control system for a heterogeneous numerical propulsion system simulation. In *Proc. 28th Annual Simulation Symposium*, Phoenix, AZ, April 1995.
- [2] Paul Avery and Ian Foster. GriPhyN: Grid physics network. 2001.
- [3] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*, 2000.

- [4] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. In *International Conference on Management of Data (SIGMOD)*, 2001.
- [5] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific data sets. *J. Network and Comput. Appl.* (to appear).
- [6] Greg Eisenhauer, Fabian Bustamante, and Karsten Schwan. Event services for high performance computing. In *Proc. 9th IEEE Intl. High Performance Distributed Computing (HPDC)*, Los Alamitos, CA, August 2000. IEEE Computer Society.
- [7] 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.
- [8] Ian Foster, Joseph Insley, Gregor von Laszewski, Carl Kesselman, and Marcus Thiebaut. Distance visualization: Data exploration on the grid. *Computer*, 32(12):36–43, December 1999.
- [9] Carol E. Kilpatrick and Karsten Schwan. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. In *IEEE International Conference on Computer Languages*, March 1990.
- [10] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, , and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Ninth Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
- [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] David Ogle, Karsten Schwan, and Richard Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.
- [14] Beth Plale, Volker Elling, Greg Eisenhauer, Karsten Schwan, Davis King, and Vernard Martin. Realizing distributed computational laboratories. *Int. J. Parallel and Distributed Systems and Networks*, 2(3):180–190, 1999. ACTA Press, www.actapress.com.
- [15] 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.
- [16] Randy Ribler, Jeffrey Vetter, Huseyin Simitci, and Daniel Reed. Autopilot: Adaptive control of distributed applications. *IEEE International High Performance Distributed Computing (HPDC)*, August 1999.
- [17] Shava Smallen, Henri Casanova, and Francine Berman. Applying scheduling and tuning to on-line parallel tomography. In *ACM/IEEE Supercomputing 2001*, Los Alamitos, CA, 2001. IEEE Computer Society.
- [18] Richard Snodgrass. A relational approach to monitoring complex systems. *IEEE Transactions on Computers*, 6(2):156–196, May 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)*, 1992.