

Dynamic Querying of Streaming Data with the dQUOB System

Beth Plale, *Member, IEEE*, and Karsten Schwan, *Senior Member, IEEE*

Abstract—Data streaming has established itself as a viable communication abstraction in data-intensive parallel and distributed computations, occurring in applications such as scientific visualization, performance monitoring, and large-scale data transfer. A known problem in large-scale event communication is tailoring the data received at the consumer. It is the general problem of extracting data of interest from a data source, a problem that the database community has successfully addressed with SQL queries, a time tested, user-friendly way for noncomputer scientists to access data. By leveraging the efficiency of query processing provided by relational queries, the dQUOB system provides a conceptual relational data model and SQL query access over streaming data. Queries can be used to extract data, combine streams, and create new streams. The language augments queries with an action to enable more complex data transformations such as Fourier transforms. The dQUOB system has been applied to two large-scale distributed applications: a safety critical autonomous robotics simulation and scientific software visualization for global atmospheric transport modeling. In this paper, we present the dQUOB system and the results of performance evaluation undertaken to assess its applicability in data-intensive wide-area computations, where the benefit of portable data transformation must be evaluated against the cost of continuous query evaluation.

Index Terms—Data-intensive computations, grid computing, data streams, publish-subscribe event channels, SQL, relational data model, database query processing.

1 INTRODUCTION

1.1 Background

DATA-INTENSIVE parallel and distributed computations have undergone dramatic increases in scale along numerous dimensions, including numbers and types of data sources, numbers of users, and problem sizes. In the scientific domain, large-scale data-intensive applications exist in computational biology, tomography [33], remote visualization [13], remote instrument control [31], and distributed data analysis [28], [8]. Beyond the scientific domain are rich media collaboration and pervasive computing. The scaling parallels and leverages the recent explosive growth of computers in everyday life. The Internet and high-bandwidth connectivity now reach a significant portion of the population; wireless communication and hand-held devices extend the reach even further. Clusters assembled from commodity PCs make it possible for institutions to provide a major collective distributed computational resource, as demonstrated by the US National Science Foundation Distributed Terascale Facility (DTF).

The impact on data-intensive computation has been dramatic. Scientists no longer consider remote location of a data set to be a barrier to its inclusion as a data source. *The Grid* [14], [15] addresses the expanding computational base of distributed multiuser workstation clusters and super-

computers with a middleware infrastructure providing services such as communication, security, scheduling, and resource location.

Within this class of data-intensive applications is a subclass of distributed applications characterized by their use of data streaming for communication. Rich media and remote visualization are well-known examples, but data streams can exist between any loosely coupled, autonomous components that communicate frequently and asynchronously. In all such applications, events flow from provider to one or more consumers. Events have no size restrictions, are often timestamped, and contain information about the behavior or state of a computational entity, physical instrument, computing environment, or user. An event stream can be initiated by a data provider (push model) or by a consumer (pull model). Publish-subscribe event systems such as ECho [11], Gryphon [4], Siena [7], and NaradaBroker [16] support subscription to events based on name or content. Data streams have been treated by others in [12], [5], [22].

A problem in data streaming applications surfaces when the applications scale in terms of number of data providers and data consumers and in richness of information exchanged. Specifically, needs mismatches begin to occur. *Needs mismatch* exists when the data sent by the supplier is not precisely the amount or in a form needed by the user. For instance, scientific data generated by an atmospheric transport model may require a Fourier transform prior to rendering. Similarly, a user may not be interested in all 3D grid data generated, but instead in aggregate or statistical results such as a 2D plot of a one-month trend. Our work addresses the needs mismatch problem.

• B. Plale is with the Computer Science Department, Indiana University, Lindley Hall, Rm 215, 150 S. Woodlawn Ave., Bloomington, IN 47405-7104. E-mail: plale@cs.indiana.edu.

• K. Schwan is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280. E-mail: schwan@cc.gatech.edu.

Manuscript received 24 Sept. 2001; revised 15 Aug. 2002; accepted 6 Nov. 2002.

For information on obtaining reprints of this article, please send e-mail to: tpls@computer.org, and reference IEEECS Log Number 115061.

1.2 Approach and Contributions

Earlier work by our group established the benefit of encapsulating needs mismatch computations into logical tasks embedded within a data stream [30], [6], [20], [25]. Our work extends this notion by casting the needs mismatch problem as a relational database problem. By creating a relational data model view of data streams and a prototype system, we can provide to the user an API that is largely the SQL query language.

Our approach is to view data streams as a relational database [32] and, through this conceptualization, gain access to the SQL query language plus an intuitive way of thinking about data streams. Specifically, we view data streams as data sources that can be operated on via a relational database query language. For instance, a time-based join of two streams results in a third stream that is a product of the two. The abstraction provides additional value through the declarative nature of the query language, which enables automated stream optimizations not possible with procedural language approaches.

On its own, the notion of *queries over data streams* is a useful abstraction, but, in our work with scientific applications, we discovered that we can enhance the transformation power of the rules by coupling a query and a complex, user defined function such that, when a query evaluates to true (returns a nonnull set), a user defined procedure is triggered that performs a task such as a Fourier transformation. The rule construct, based on triggers in active databases [36], leverages the dual strengths of queries for joining event streams, filtering, and creating (i.e., materialized) event types and application-specific algorithms for the mathematical computation needed to transform the data. Furthermore, by uniform treatment of all data sources as relations, a user can write a single query that is simultaneously responsive to network conditions, changing user needs, and application behavior. Finally, through its formal foundation in relational calculus, SQL has evolved as a highly efficient query language with proven query optimization heuristics and wide public adoption and acceptance.

Application of the work, first to safety-critical systems [29], then to scientific applications [30], has provided early confirmation of the usefulness of the abstraction. Examples from our work with a distributed global atmospheric transport model demonstrate that meaningful queries can be stated in the subset of SQL that we support. Experimental evaluation of the prototype system, dQUOB, shows that query processing can be accomplished efficiently for realistic stream rates.

1.3 Key Challenges

Continuous query processing over large-scale streaming data imposes unique challenges on database query processing. The contribution of this work is to address the following three challenges in the context of high-performance computing applications. The first is to question the applicability of traditional query heuristics when applied to the vastly dissimilar environment of streaming data. With traditional databases, the cost metric by which query evaluation performance is evaluated emphasizes minimized disk access. That is, plan generation selects, from among multiple query

plans, the one that minimizes total disk access times. With disk latency in the millisecond range and CPU clock speeds in the nanosecond range, the metric is viable. Queries over data streams, on the other hand, are evaluated over events that are delivered to the query evaluation engine over a network interface. Hence, a more relevant metric is one that minimizes total response time.

The second challenge is to minimize query latency. Query latency is the interval of time between when a condition over distributed components occurs and when it is detected by a query. To minimize latency, queries are executed continuously and over a snapshot of the data streams. The challenge exists in that time-based operators (i.e., join and temporal select) require some amount of event retention. The correct size for retained event sequences is a tradeoff between wasted memory and higher incidence of false negatives.

The third challenge arises out of the absence of actual data values during query optimization, which, in consequence, pushes part of the optimization into being performed at runtime. The manner in which query optimization and plan generation is performed under these new conditions, and in a suitable runtime representation for a query such that it is amenable to runtime optimization, while, at the same time, executing at rates in the microsecond range form the third and final challenge of the work.

1.4 Overview

In the following section, we discuss related work. The examples in the paper are derived from a distributed visualization that is discussed in Section 3. A high-level user view of the system is given in Section 4. Section 5 addresses the second challenge, that is, the problem of partial views over data streams. Section 6 discusses the architecture and runtime system, both of which are critical to the third challenge, the need for reoptimization at runtime. Section 7 addresses the remaining challenge of efficiency of query evaluation and reassessment of traditional query heuristics in a streaming environment. We conclude with a summary and discussion of future work.

2 RELATED RESEARCH

Continuous queries of the kind that form the heart of dQUOB have their history in database research. Seminal work in the area was done at Xerox Parc in the early 1990s by Terry et al. [35]. They coined the term “continuous queries” to mean conventional database queries that are issued once and, henceforth, run continually over the database. The seminal contribution of the paper is in its definition of the type of database and the class of queries that work for continuous semantics. There has been recent renewed interest by the database community in continuous query systems. A good review of current work appears in [3]; several of the projects are discussed here.

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. That is, the execution order of operators is determined by factors such as the arrival rate of tuples. Eddies targets query optimization in a

database setting, but shares the same goal as dQUOB of being responsive to changes in the environment. The STREAM project [3] advocates support for data streams as an integral part of a DBMS. It focuses on rich language (SQL) support and assumes streams and queries are directed to a single location. Our work, on the other hand, stresses portability and distribution of query processing points and accepts a subset of SQL as a tradeoff for efficient query execution. The Fjords data-flow architecture [24] focuses on 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, which “zippers” together streams according to a time interval. The zipper join can be applied to synchronized streams. As we discovered in our work, however, synchronized streams are the exception rather than the rule.

Temporal (or valid-time) databases have been applied to performance monitoring of parallel and distributed computing in work done by Snodgrass [34], Ogle et al. [25], and Kilpatrick and Schwan [21]. Queries are issued postmortem against the performance data [34]. Our goal, on the other hand, is analysis at runtime, which puts greater demands on efficiency.

Runtime detection in data streams has been addressed with other approaches, including fuzzy logic [31] and rule-based control [1]. We argue that the more static nature of these approaches makes them less adaptable to changing user needs and changes in data behavior. The Active Data Repository [12] 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. The Continual Queries system [23] is optimized to compute and return the difference between current query results and prior results as applied to detecting changes in 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.

3 MOTIVATION

In an increasingly connected world, a growing number of data-intensive applications use data streaming for communication. Here, we explore one example, that of distributed scientific visualization.

Atmospheric scientists interact with, and visualize results from a coupled parallel and distributed global atmospheric transport model and a parallel chemical model [28]. The transport model simulates movement of ozone in the upper atmosphere subject to factors such as horizontal and vertical winds; the chemical model simulates ozone interaction with short-lived chemical species (e.g., N_2 , NO_3 , CH_3O_2). Temperature and species concentration are exchanged at the end of every logical timestep. The transport and chemical models produce an *event* at every timestep consisting of ozone, temperature, winds, and the short-lived species data. The event is timestamped and pushed to consumers who have explicitly subscribed to the event channel. The consumer of the visualization streams is a visAD [19] visualization client executing on a remote high-end graphics workstation.

Even this relatively simple model of two producers streaming data to a single consumer illustrates the needs mismatch problem discussed in Section 1. The transport model uses a spectral representation of the data, whereas the visualization and chemical model use a 3D grid representation. The scales of the two models differ, forcing reconciliation prior to visualization. Problems are exacerbated in the presence of additional clients. A second client may desire a 2D graph plotting a trend over one month of data and a third, having limited compute resources, may desire to be notified when an event of interest occurs such as the ozone density over the south pole dropping below a threshold. The needs mismatch problem grows with each additional producer and consumer. Below, we provide a simple example of an SQL query to illustrate how a database query language can address the needs mismatch problem.

Queries in dQUOB are technically more than just a query. They are actually If-Then rules [36], where the IF clause contains the SQL query and the THEN clause contains an action, a user-defined function that is executed when the result of the query is a nonnull set. The combined construct leverages the strengths of SQL queries for filtering and limited aggregation (i.e., sum, max, and min) and the action for mathematical manipulation of the data set.

As an example, the rule C:1 shown below, accepts three event types, all listed in the FROM clause: data events from the atmospheric model as *Data_Ev*, a user request as *Request_Ev*, and a performance monitoring event as *Perf_Ev*. The condition in the WHERE clause evaluates to true if the data event from the model is at least partially contained in the region bounded by the min and max latitudes provided by the user and the current network latency is better than a threshold specified by the constant *MAX_ERT*. If the query is satisfied, the function “ppm2ppb” is triggered and passes the events that satisfied the query (one of each event type). The function “ppm2ppb” converts species concentrations from parts-per-million to parts-per-billion.

```
CREATE RULE C:1 ON Data_Ev, Request_Ev, Perf_Ev
IF
  SELECT Data_Ev
  FROM Data_Ev as d, Request_Ev as r,
  Perf_Ev as p
  WHERE
    (d.lat_min >= r.lat_min or
     d.lat_min <= r.lat_max) and
     d.level_min >= 30 and
     p.latency <= MAX_ERT and p.aid = r.aid
THEN
  FUNC ppm2ppb
```

4 HIGH-LEVEL SYSTEM VIEW

Applications for which dQUOB filtering is particularly well-suited include data flow applications [12] such as scientific visualization, grid applications having external sensors or scientific instruments as input sources [9], and online monitoring for performance or hazard detection [32].

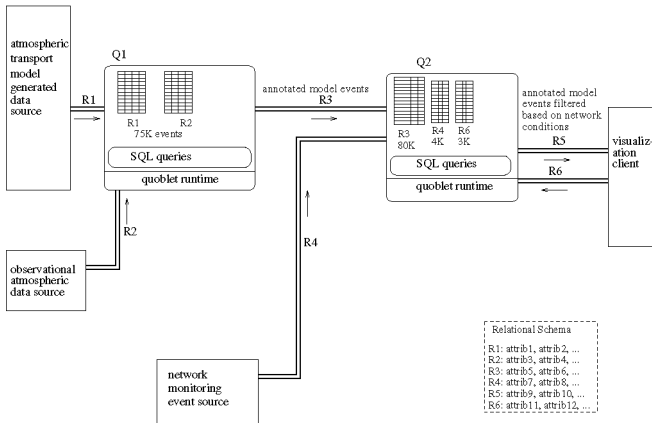


Fig. 1. dQUOB data stream for visualization client.

These streaming applications can be generalized to distributed and heterogeneous data providers and consumers who communicate frequently through timestamped events. Data providers generate streams of events and data consumers receive the events; a component can play the simultaneous role of provider and consumer. The dQUOB system facilitates development of streaming applications by providing an API for creating components that transform, aggregate, and filter data in the intermediate stages between data provider and consumer. Systems like dQUOB, based on database models, are referred to as *continuous query* systems because the queries execute continuously, upon every arrival of an event.

Fig. 1 depicts a visualization flow instrumented with dQUOB components added for flow processing. The atmospheric transport model generates an event at each logical timestep. The observational atmospheric data source streams satellite observation data used to validate the transport model. Before visualization can occur, the transport model data must be matched to the observational data, must undergo transformation from the spectral domain to the grid domain, and, finally, must undergo filtering based on the user's request for a particular region of data. The intermediate computation is handled by dQUOB quoblets. A *quoblet* is a task, either thread or process, that executes one or more query rules over the data streams. Fig. 1 shows two quoblets, Q1 and Q2. Q1 contains one query that joins the model generated stream, R1, and observational stream, R2. The result is annotated model events that are pushed downstream on event channel R3. The tables "R1" and "R2" inside Q1 are sliding windows over their respective data streams. Quoblet Q2 filters annotated model events based on infrequent network monitoring events received on channel R4 and on infrequent user requests for a particular atmospheric region of data received on R6. The results are forwarded to the client on event channel R5.

The application is described by a single schema, as shown in the bottom right of the figure. This schema is the sole way in which a quoblet understands the format of events it receives and generates and knows to which event channels to subscribe. It would be reasonable to group the queries of Q1 and Q2 into a single quoblet, but separating them provides an opportunity for additional consumers to connect directly to the annotated model channel. A quoblet

containing multiple queries would have multiple downstream event channels.

The "data streams as database" abstraction can now be described more succinctly. Queries are organized by quoblets as the computational unit. A query can run at one or more quoblets; a quoblet can run any number of queries concurrently. Queries execute continuously over events arriving at quoblet. A query that is satisfied yields an event that flows on a downstream data stream (similar to a relational database query that yields a relation as a result). A single event in event systems terminology is equated to a tuple in database terminology; similarly, data stream is equated to a relation (or table). Queries added to a quoblet begin processing upon the arrival of the next event. That is, events are not retained by a quoblet for historical purposes. Such a feature has negative implication on memory demands, though it could be accomplished with a logfile.

The streaming environment discussed above imposes a unique set of requirements on database query processing. One of these, the need for query responsiveness and the challenge that it creates for time-based operators, is discussed next.

5 SLIDING JOIN WINDOW

A join operation is the Cartesian product of two tables subject to a condition.¹ But, in a streaming application, it is only after an application terminates that a full table of data is available. The requirement that queries be responsive precludes doing post mortem analysis. Thus, query evaluation must be performed on the fly over an interval of the event stream. To address the need, in [32], we introduced the notion of a sliding window over the event streams that we call a join window. A *join window* is an event sequence of relation R for the join operator J1 of query Q. The difficult question is the size of the join window. For joins such as ours in which two events are joined if they "happen at the same time," the optimum join window size depends upon such factors as the skew of the producer clocks, latency of the network, and asynchronous behavior of the streams.

In the early version of the system, the join window size was determined at startup as an integer count and the count was applied uniformly to all event streams and join operations. But, the solution was flawed for two reasons. First, an integer size is too low-level a notion for a user to reason about stream flow rates. Second, the cost of a wrong guess is simply too high. Too small a window increases the likelihood of false negatives [23]; too large a window consumes memory, impedes throughput, and limits scalability.

Based on experience with data streams in scientific computing, we concluded that a more viable approach would be to base join window sizes on a user specified time interval. Our algorithm sets initial join window sizes based on a user-specified time interval, then through an introspection technique that monitors the stream rates to maintain the window sizes at a desired size based on the user specified time interval.

1. A table can also be joined with itself.

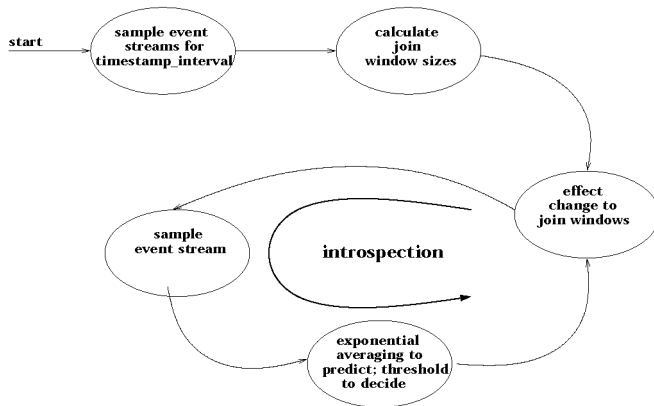


Fig. 2. Join window algorithm. The first three states are the startup phase. The Introspection phase that follows employs monitoring and prediction to detect significant changes in stream rates upon which join window size change is made.

Specifically, our overall goal is to maintain join windows at a size expressed as a timestamp interval irrespective of changes in stream rates. This approach offers two gains. First, it positions the user to better reason about the tradeoff 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 seconds, then a larger starting interval can be selected. Second, window sizes based on a time interval self-adjust themselves to match the asynchronism between two streams. A fast stream that receives 1,000 events/second will retain 1,000 events for a join window size set at “1 second,” whereas a slow stream generating five events/second will require storage for only five events. Storage demands as a result become more sensitive to stream rates, and overall scalability of the system improves.

Determination of the initial join window sizes is shown as the first three states of Fig. 2. The algorithm accepts a single input parameter, a user defined timestamp interval, that represents the users desired join window interval. The timestamp interval is used as the initial sampling interval, though it would be trivial to sample over a larger interval were the timestamp interval too small to obtain a valid sample. Upon completion of sampling, join window sizes are computed relative to the fastest stream. The ongoing introspective phase employs stream sampling and exponential averaging to predict the next join window size based on past observation. In order to abate constant minor adjustments to join window sizes, the transition to the “effect change” state is taken only when the predicted value exceeds a threshold. The algorithm is presented in more detail in [27].

6 SYSTEM ARCHITECTURE

A key challenge to the system architecture is a query representation and deployment strategy that satisfies the requirements of portability, efficiency, and optimizability. Wide area applications are long running and dynamic; the latter due, in part, to the volatility in available network bandwidth. In order to be responsive, our system must

support query addition and removal on-the-fly. A long running application is also more likely to undergo a major mode transition or change in application behavior. To be responsive to changes in data patterns, our system must support query reoptimization because shifts in data modalities can impact query performance. A query optimal under one mode is not necessarily optimal under another. The architecture addressing these needs is discussed in this section.

The system architecture consists of the following three components, which are discussed in this section: The dQUOB client is the vehicle through which the user creates queries. The dQUOB server is a centralized, application-wide service and repository that accepts queries from users and deploys them at quoblets. The runtime system, or quoblets, are the executable entities that actually transform the data stream.

6.1 Query Generation and Deployment: The dQUOB Compiler and Server

The dQUOB client is essentially a query compiler. The compiler parses SQL query, optimizes the query through application of optimization heuristics, and generates a portable, intermediate representation of a query plan.

The choice of a representation for the query plans is driven by the needs of portability and efficiency of the final representation. To meet the efficiency demands of high-performance scientific computing, the final representation must be compiled code (as opposed to interpreted). But, queries must be portable as well, capable of deployment at any time, anywhere. Our approach was to develop a library that resides at a quoblet. The library consists of object classes from which operator objects (i.e., select, project, and join) are instantiated and linked as a directed graph. Since any scripting language satisfies the need for portability and Tcl interfaces well with the C language in which the library is written, we selected Tcl. The compiler generates a script for a query plan that consists of calls to the dQUOB library; the order of calls is guided by the compiler’s internal relational calculus representation of the query. Scripts are portable and small; a script for a query is roughly one-tenth the size of the compiled code representation [30]. The query plans are stored to a repository for retrieval at runtime.

We considered alternative representations. The query compiler could generate procedural language source code at the target machine, then a target compiler could generate object code that could be dynamically linked at the quoblet. The JIT compilers serve this purpose, but JIT compilers are generally specific to the Java language. Or, the object code could be cross-compiled at the client and stored to a code repository. Either alternative has the advantage of removing the need for the dQUOB library, but the former is specific to Java, which we rejected for performance reasons, and the latter lacks support for on-the-fly query modification.

The dQUOB server is a relational database repository with an HTTP interface. The repository stores the query scripts plus supporting relational schemas and metadata.

6.2 Runtime: Quoblets

A quoblet is essentially a runtime for executing queries. As such, it performs event handling, graph traversal, query

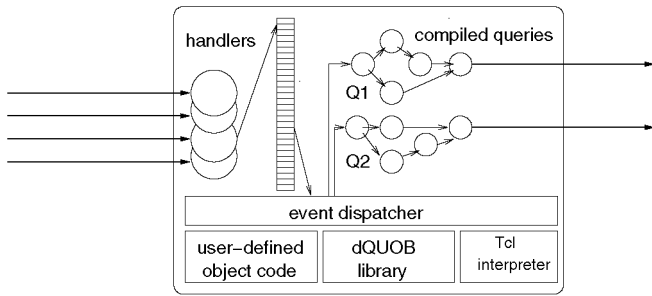


Fig. 3. Quoblet architecture.

instantiation, and reoptimization. The components of the quoblet are shown in Fig. 3. Event handlers handle incoming streams, one handler per stream. Events are copied into memory, represented in the figure as a large vector. The event dispatcher routes incoming events to the queries that have registered an interest in the event type and initiates query evaluation. The user defined object code is the user defined mathematical functions. A function is dynamically loaded into a quoblet at the time a query referencing that function is added to the quoblet. The dQUOB library and Tcl interpreter operate in concert when a query script is added to the quoblet. The script consists of Tcl calls to the dQUOB library. A script, when executed by the Tcl interpreter, creates operator objects and links them together to form the query graph. For instance, a script may contain a command to create a select operator. The script calls the dQUOB library, the latter of which instantiates a select object that is parameterized by the script to execute a comparison. Parameters are specified down to the level of the logical or temporal operand to use. The query, once instantiated, registers itself with event dispatcher and begins to receive events.

A query is represented at runtime as a directed acyclic graph (DAG) where a node is one of select, project, join, or action, and an arc is flow of control. The query graph has multiple entrance points and a single exit point. If the user has specified an action for the query, the action is always the last node to be executed. Events are pushed through the query, that is, an operator knows to whom to direct its output. Sets of events satisfying the query are forwarded downstream to another query in the quoblet, a query in a different quoblet, or to a consumer. A query is executed by a depth-first traversal of the query graph. The strength of a query representation as a graph is the opportunity for dynamic reoptimizability; that is, two select nodes can be swapped by updating a couple pointers. A query heuristic, discussed in Section 7.3, leverages this feature to reorder select nodes at runtime. On the downside, a query graph representation carries the additional overhead of graph traversal; the overhead is quantified experimentally in Section 7.4. Queries themselves are nodes in a larger graph of queries that the dispatcher executes by depth-first traversal.

7 EXPERIMENTAL EVALUATION

Continuous execution of queries over event streams, where events vary in size from a few hundred kilobytes to several

hundred megabytes and streams are frequently asynchronous is a novel scenario. In order to establish the viability of the dQUOB system as a solution for continual querying, we undertook a wide range of measurements that we discuss in this section. They include measurements to determine the efficiency of query evaluation, to quantify the benefit of individual optimization heuristics, and to quantify the overhead of the graph representation of queries when compared to hard-coded, static queries.

7.1 Microbenchmarks

We undertook microbenchmarking to establish the execution time of the individual query operators: select, project, and join. The measurements are obtained using a workload of 16,872 application events generated by an autonomous robotics simulation; the experiment was conducted on a Sun UltraSPARC 1. The microbenchmark results were gathered for a “typical” query consisting of 13 operators, three of those join operators. All joins execute a Cartesian product and time-based condition over a sliding window of size 120. The event size is 75K.

The results are as follows: *projection 8.6ms, selection 1.6ms, and join 0.5ms*. The relatively high cost of the projection operator relative to the others is a worst case, incurred when a project serves in the role of materializing a view. In this situation, it must allocate space for a new event and copy attribute values from existing events into the materialized view. *Join* has a misleading name as a microbenchmark. The operation that is measured is the time to evaluate a logical expression over two events, one from each stream. A true join is a Cartesian product over two tables and evaluation of a condition over each pair; measurements of the latter are highly dependent on the size of the two participating tables, so are not considered here.

7.2 Query Optimization in Application Setting

We undertook to determine query processing costs as a percentage of total quoblet time in a realistic application setting. The query is nontrivial in number of operators and its purpose is to filter global atmospheric transport events based on atmospheric pressure and region. The user supplied action performs a units conversion on each data point in a 3D grid slice. We consider user algorithms like these, which perform a simple arithmetic operation on each grid point, to have midrange computational needs.

Fig. 4 shows the breakdown of quoblet computation time into percentage spent executing the query and the action for three different event sizes. The charts in the unoptimized column reveal that query processing costs as a percentage of the whole varies widely, depending on the computational cost of the user supplied action. At the 612-byte event size, query processing represents 98 percent of the total time. This finding is not surprising as a 612-byte event contains no species data, thus no units conversion computation is performed. At the largest event size, the event contains nine species, so all but 612 bytes of attribute information must be converted. In this case, the query represents only 10 percent of total execution time. The second column shows query execution time for the same query, but in its optimized form.

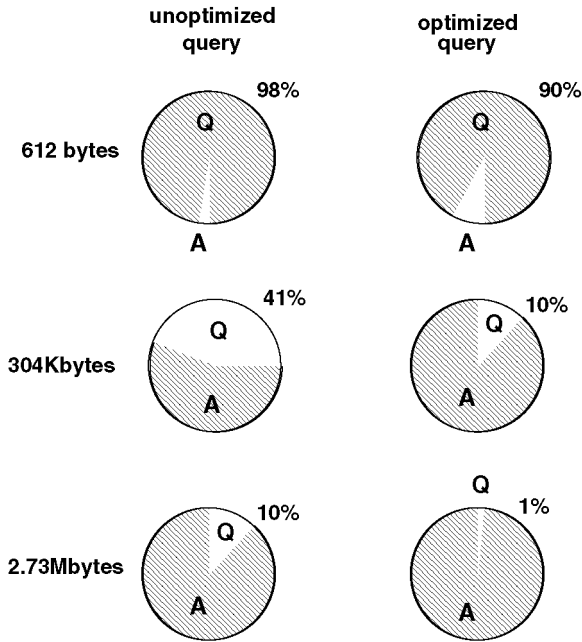


Fig. 4. Breakdown of quoblet execution time by query (Q) and action (A).

The results show that, in an application setting with meaningful queries, realistic event sizes, and midrange computational needs, optimized queries consume a small portion of total execution time. Further, the results show that optimized queries result in significant decreases in total processing time and, as such, testify to the value of a declarative language for filtering tasks because the declarative query can undergo optimization. The optimization heuristics that we considered are discussed next.

7.3 Query Optimization Heuristics

A declarative query language is highly desirable when users are application scientists because it is very unlikely that application scientists will put the time and effort into hunting for the optimal way to state a request. This section considers four algebraic heuristics that we revisit in the context of a data streams environment. Algebraic heuristics directly manipulate a query tree (similar in structure and purpose to a parse tree), producing a new tree that results in a lower cost metric value than the former tree.

7.3.1 Pushing Selects

Pushing selects down the query tree is a well-known database optimization heuristic. Our experimental evaluation of this optimization technique uses a quoblet containing one application-realistic query that evaluates over a data stream generated by an autonomous cooperating robotics simulation. The workload consists of 16,872 application specific events. The experiment was conducted on a cluster of Sun UltraSPARC 1s connected via a 100Mbps switched Ethernet. The unoptimized query is one that might be specified by a naive user before optimization is applied. The optimized query is the result of pushing selects down the parse tree wherever possible.

Table 1 shows the execution times for query execution over a single event for optimized and unoptimized queries. The second row shows the average number of operations executed per event, where an operation is one of selection, projection, join, or overhead. The third row breaks execu-

TABLE 1
Optimization I: Pushing Selects Down the Parse Tree

<i>Push Selects</i>	<i>Optimized</i>	<i>Unoptimized</i>
execution time (secs)	0.14427	1.08329
number ops	2	51
% by op (ovhd/project/select/join)	.49/0/.51/0	.02/0.11/.87

tion down into percentage by operator type. For instance, in the unoptimized case, 2 percent of the operations are overhead, 11 percent are selection, and 87 percent are join. Optimization yielded an 87 percent reduction in query execution time and a 96 percent reduction in number of operations largely credited to reduced number of joins. As can be seen in the breakdown by type, in the unoptimized version, 87 percent of the operations are attributed to join, whereas that number is reduced to zero in the optimized version. Selects that filter 100 percent of the events (so that none reach the join operation) is a best case.

7.3.2 Factorization

To determine the performance gains achievable through internode optimizations, we evaluated the heuristic of factoring a common subexpression from two or more queries into a separate query. Intuitively, this should result in performance gains by decreasing the number of times the subexpression must be evaluated. Our experiment compared two queries having a common subexpression duplicated across the queries to three queries, two having the common subexpression removed, and a third containing the subexpression. The experiment was run on a cluster of Sun UltraSPARC 1s connected via a 100Mbps switched Ethernet. The workload is the autonomous robotics stream described earlier.

The results, expressed on a per event basis, appear in Table 2. From the execution times, one can see that there is a slight performance benefit to factoring common subexpressions. But, the second row reveals an inconsistency. Total number of operations per event drops by more than half in the factoring case. A proportional drop in execution time does not match this reduction in number of operations. Insight to the problem can be gained by the breakdown by operation type. In the "no factoring" case, project constitutes 7 percent of the total operations, whereas, in the filter case, it accounts for 33 percent. The disparity indicates that the gain achieved through a reduced number of select operations is largely consumed by increased projection costs, which we have determined through the microbenchmarks to be costly. We conclude that factoring a common subexpression is not a beneficial heuristic to use on queries that have a common address space. Distributed queries could benefit from

TABLE 2
Optimization II: Factoring Out a Common Subexpression

<i>Factoring</i>	<i>Factoring</i>	<i>No Factoring</i>
exccution time (secs)	0.22191	0.23224
number ops	1.97	4.28
% by op (ovhd/project/select/join)	.33/.33/.33/0	.31/.07/.62/0

pushing a common subexpression upstream of the quoblet resulting in reduced compute cycles and bandwidth needs. We expect such a heuristic to further improve performance in the distributed case.

7.3.3 Pushing Projects

The projection operator has diminished usefulness in a data stream environment. Projection generally serves two purposes: to reduce the size of the participating tuples and to form new events (i.e., materialize views) from participating attributes. When tuples are disk resident and large, it is viable to reduce individual tuple size whenever possible because reductions in the amount of data transferred reduce disk access times, SCSI channel traffic, and latencies, which comprise a significant portion of query execution time. Hence, much effort in database query optimization is directed at choosing a query plan that minimizes these costs. Given the large memories present in today's systems, the fact that events arriving at the quoblet arrive on an NIC, and the relatively high cost of the projection operation, we rejected the heuristic of pushing projects down the parse tree.

7.3.4 Reordering Selects

The final query optimization heuristic we consider requires statistical information gathering. In a traditional database, gathering statistical information is relatively easy since the data resides in tables. But, in a data stream environment, events are not available at compile time. Thus, optimizations involving statistical metrics, specifically selectivity, must be deferred to runtime. A *selectivity* is a probability assigned to a particular select operation. For example, if a select compares the value of atmospheric pressure to "5" and there are 37 atmospheric "levels," then the probability that the select will evaluate to true is 1/37. The selection heuristic is to push the select operators with smaller selectivities down the query tree.

dQUOB computes selectivities at runtime through data stream sampling and depth-first histograms. Selectivity estimation is based on theoretical work done in the early 1980s by Piatetsky-Shapiro and Connell [26]. Reoptimization is accomplished by on-the-fly reordering of the operators making up a query. A key strength of the work is the ability to reorder operators efficiently and without compromising correctness. There are two reasons. First, relational calculus operators are associative, so an incorrect operator ordering cannot compromise correctness. At worst case, an ordering less optimal than its predecessor will be selected. Second, each node in the executable query graph is an independent, side-effect-free function. The ease with which reoptimization can be done must be contrasted to work in code movement where blocks of code are reordered to improve efficiency [18].

Query optimization techniques must be rethought in a data stream environment. Heuristics known to yield performance gains in a database may not yield gains in data streaming. While pushing selects down the parse tree yielded significant improvements in query evaluation time, pushing projects had a detrimental impact on performance. Factorization was also rejected because the cost of executing the additional project operator overshadowed any benefit of a reduced number of select operations. The final optimization, reordering based on selectivities, has the potential to be quite beneficial.

TABLE 3
dQUOB versus "Static" Query Cost Per Data Model Event

<i>dQUOB Query</i>			
<i>Join buffer size</i>	<i>Overhead (μs)</i>	<i>Query (μs)</i>	<i>Total (μs)</i>
1	34.07	87.00	121.07
10	32.35	87.31	119.66
120	32.43	84.56	116.99
<i>'Static' Query</i>			
<i>Join buffer size</i>	<i>Overhead (μs)</i>	<i>Query (μs)</i>	<i>Total (μs)</i>
1	16.65	2.78	19.43

7.4 Overhead of Dynamic Runtime Query Representation

We assess performance implications of our decision to represent a query as a directed graph by comparing the performance of a dQUOB query against a baseline case. The general representation of dQUOB queries as directed graphs of operators allows fast reoptimization and dynamic instantiation, but at the expense of additional overhead. To quantify that overhead, we specify a moderately complex query (i.e., seven selection, two join, and one projection operators) and compare two implementations. The first implementation uses a dQUOB-style graph representation, and the second uses a C++ function compiled into the quoblet. The latter is referred to in Table 3, as the "static query" since changes to the query necessitate code recompilation.

Query processing costs are averaged over a stream of 120 application events to obtain a per event processing cost. There is no user-supplied action. The environment is a cluster of single processor Sun Ultra 30 247MHz workstations running Solaris 7 and connected via 100 Mbps switched FastEthernet. The quoblet, provider, and consumer reside on separate workstations. The test case quoblet receives events on three event channels and writes events to one channel. Events generated by the atmospheric model are 304K bytes in size and are generated at every logical timestep. The client and performance monitor generate smaller events infrequently on the other two channels at random intervals.

The dQUOB query processing cost is given at the top of Table 3 for three sizes of join windows. The results are broken out by quoblet overhead and query evaluation time. The "Total" column, which is the sum of the Overhead and Query columns, shows the time to execute a dQUOB query over a single atmospheric data event. As can be seen, this is in the 120 μ s range. Additional sample queries also tested confirmed this number. Compared to the total execution time for a static query in the 20 μ s range, one can conclude that dQUOB query processing incurs a worst case overhead of approximately 6x.

One may note that, in the dQUOB query case, the smaller join window performs more poorly than the larger sizes. This is counter-intuitive because the join operator is essentially Cartesian Product over two "tables," so larger window size should yield worse per event processing time. The explanation is two-fold. First, we optimize join processing by making the distinction between *snapshot*

events or behavior of an application at an instant, such as is generated by the atmospheric model and *interval events*, events describing behavior existing over a duration, such as is generated by the client and performance monitor. When one “table” in a join contains interval events, only the most recent interval event participates in the join. Thus, under continuous queries where evaluation is triggered by every event arrival, Cartesian product over an event pair where one of the pair is an interval event requires at most one tuple evaluation. The effect is a flattening of the execution time growth curve as the join window grows. Second, the decrease in execution time shown at the larger buffer sizes reflects less frequent storage reclamation, reclamation which is performed whenever an event is no longer needed. At size 1, storage reclamation is performed continuously as events are being released constantly.

7.5 Wide Area Network Measurements

The utility of dQUOB is derived from 1) its ease of use in terms of the user’s ability to easily express complex behavior over multiple, diverse, long lived data streams, and 2) the performance advantages gained by using dQUOB which are accentuated in bandwidth-constrained environments (e.g., WAN). This section demonstrates its utility by evaluating the adaptability of a set of cooperating queries over time subject to changes in the underlying network. We additionally demonstrate the impact of location of the cooperating queries in both a LAN and WAN setting.

7.5.1 Adaptivity

A strength of the *data streams as database* view is that any event stream, irrespective of source, fits into the database view. Users can write a single query that includes such diverse sources as network resource information [37], [10], [17], application data, and user-specific requests. We have observed that a user’s response to resource availability (e.g., end-to-end latency) can often be described by a small finite state machine. By describing each state in the finite state machine with a separate query, our approach can achieve FSM control. Queries executing concurrently provide the state transitions needed to dynamically respond to changes in the environment. Specifically, transition between states (i.e., queries) is possible because queries can be written very naturally as the conjunction of an application condition and a check on a resource state. If the resource state check fails, the entire query fails irrespective of the outcome of the remainder of the query. The implication is that a query can effectively be turned on and off by the receipt of a resource state change event.

Our first experiment tests the adaptivity of queries in the presence of changes in the underlying network. We employ a quoblet containing three queries: a “no filtering” query called “null query,” a query with moderate filtering capability, and one that applies strong filtering. The impact on end-to-end latency over time is depicted in Fig. 5. At the outset, “null query” is active. “Null query” does no filtering, so, during the first phase (180 events), computation cost is dominated by the action routine. At logical timestep 180, the receipt of a network monitoring event causes “null query” to be deactivated and the “moderate filtering” query to be activated. The newly active query filters out a small number of atmospheric pressure levels from every 3D grid slice resulting in a drop in event size from 75K to 55K. At timestep 370, a second network event is received, this time indicating significant degradation in

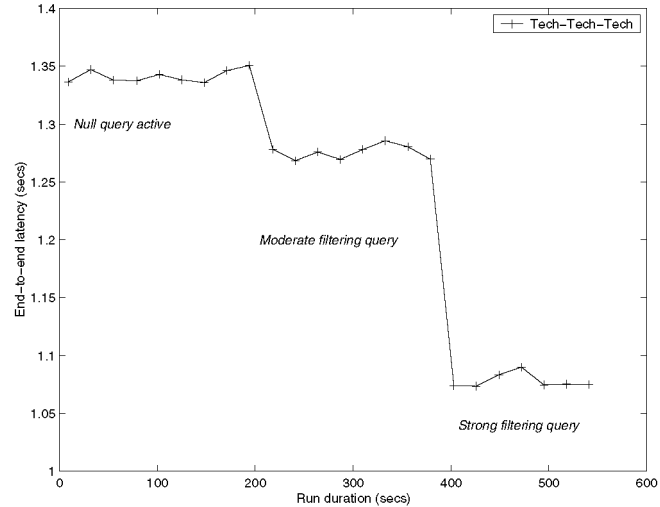


Fig. 5. Quoblet potential for responsiveness in a dynamic environment.

end-to-end latency. The event triggers the deactivation of the moderate filtering query and activation of the “strong filtering query,” the latter of which performs strong filtering whereupon 2K events are generated.

The test scenario employed a workload of 540 global atmospheric model events (540 logical timesteps) plus a smaller number of user requests and end-to-end latency updates. The experiment was run over a local area network on a cluster of Sun Ultra 30 247 MHz workstations running Solaris 7 and connected via 100 Mbps switched FastEthernet.

Two concluding points can be made. The test was run in a local area environment, thus the dominant variable in average end-to-end latency is quoblet computation time. As we show below, in a WAN environment, other variables overshadow quoblet computation time. Second, note the abrupt quoblet transition in response to state change events. Because of the binary nature of queries that contain a check on resource state, queries can be instantly “turned on” and off.

7.5.2 Impact of Quoblet Location

The second experiment evaluates the impact of quoblet location in a WAN environment and employs resources at Georgia Tech and the Albuquerque High Performance Computing Center (AHPCC). Using a model of single supplier, single consumer, and quoblet positioned between, we undertook to compare the benefits of the quoblet residing at the source to that of its residence at the client. These are, respectively, Tech-Tech-AHPCC and Tech-AHPCC-AHPCC in Fig. 6. The Tech-Tech-Tech plot is the baseline case from Fig. 5. The experiment uses the 540 event workload from the global atmospheric model described earlier. The AHPCC machine is an 8 processor Onyx 2, R10000 running IRIX64 6.5, 1 G RAM/node, and connected within AHPCC by FastEthernet. The Georgia Tech machine is a SunUltra 30 with gigabit Ethernet. The link between Georgia Tech and AHPCC is Internet 2.

The measurements show a benefit of pushing data manipulation closer to the source. In Fig. 6, with the quoblet located at the client, the average end-to-end latency is 0.614 events/second. With the quoblet at the source, end-to-end latency is an increase 0.998 events/second. It can be seen in the figure that pushing the quoblet toward the source results in performance that more closely patterns

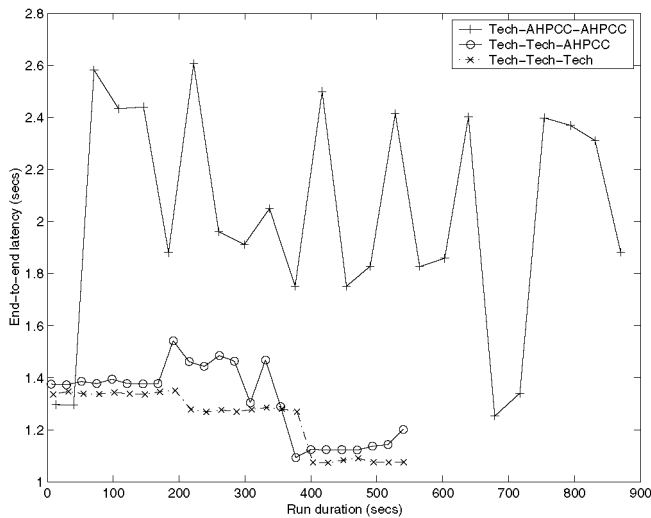


Fig. 6. WAN measurement (Georgia Tech → AHPC).

itself after the application run on the LAN. The erratic results for the case where filtering is done at the consumer occurred in repeated experiments. We believe the results are a function of traffic control in TCP; however, we did not optimize TCP configuration for these experiments. The results of Fig. 6 are encouraging indications that filtering early in the data stream can be effective in reducing execution times directly under dQUOB control and, perhaps more interestingly, in ensuring more predictable overall wide-area behavior.

8 CONCLUSION

We introduced in this paper an alternate way of thinking about data streams, that is, as entities capable of being queried with a relational database language, and have introduced dQUOB, a system for creating and executing queries over data streams. By conceptualizing data streams in distributed peer-to-peer applications as a relational database, one can view extracting data as the specification of one or more queries. Through database joins, streams can be combined and, through materialized views, new streams can be created. We feel a strength of the conceptualization is its naturalness.

The dQUOB system consists of a compiler through which the user creates queries, a server that deploys queries, and a runtime consisting of a library, an event handler, a query evaluation engine, and queries. Measurements of the runtime performed in LAN and WAN settings have confirmed the efficiency of queries, the benefit of query optimization, and the advantage of portability in decision making.

In ongoing work, we are considering alternate schemes for extremely large data events. Runtime performance suffers under large (MB) events because the quoblet copies the full event into user space. Depending on the data representation and on the amount of data touched by the user-defined functions, a full copy may not be needed. For instance, a lazy copy could copy the minimal attributes needed to execute the query, then copy the remaining data only after it is determined that the event satisfies the query.

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 a probability assessment for the likelihood of false negatives for a selected window size. We can then, for example, respond to a window size setting of one second with a warning such as "Beware, the likelihood of false negatives for a one second window size is 90 percent."

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their insightful comments on this paper. The end result is far stronger for their effort.

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," *Proc. 28th Ann. Simulation Symp.*, Apr. 1995.
- [2] R. Avnur and J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *Proc. Int'l Conf. Management of Data (SIGMOD)*, 2000.
- [3] S. Babu and J. Widom, "Continuous Queries over Data Streams," *Proc. Int'l Conf. Management of Data (SIGMOD)*, 2001.
- [4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Storm, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '99)*, 1999.
- [5] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A Component Based Services Architecture for Building Distributed Applications," *Proc. IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 2000.
- [6] F.E. Bustamante and K. Schwan, "Active I/O Streams for Heterogeneous High Performance Computing," *Proc. Parallel Computing (ParCo) '99*, Aug. 1999.
- [7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Trans. Computer Systems*, vol. 19, no. 3, pp. 332-383, Aug. 2001.
- [8] A. Chervenak, I. Foster, C. Kesselman, C. Salisburry, and S. 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).
- [9] E. Deelman, K. Blackburn, P. Ehrens, C. Kesselman, S. Koranda, and A. Lazzarin, "Grifphyn and Ligo, Building a Virtual Data Grid for Gravitational Wave Scientists," *Proc. 11th IEEE Int'l High Performance Distributed Computing (HPDC)*, Aug. 2002.
- [10] P.A. Dinda and D.R. O'Hallaron, "An Extensible Toolkit for Resource Prediction in Distributed Systems," technical report, Carnegie Mellon Univ., 1999.
- [11] G. Eisenhauer, F. Bustamante, and K. Schwan, "Event Services for High Performance Computing," *Proc. Ninth IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 2000.
- [12] R. Ferreira, T. Kurc, M. Beynon, C. Chang, and J. Saltz, "Object-Relational Queries into Multidimensional Databases with the Active Data Repository," *J. Supercomputer Applications and High Performance Computing (IJSA)*, 1999.
- [13] I. Foster, J. Insley, G. von Laszewski, C. Kesselman, and M. Thiebaux, "Distance Visualization: Data Exploration on the Grid," *Computer*, vol. 32, no. 12, pp. 36-43, Dec. 1999.
- [14] *The Grid: Blueprint for a New Computing Infrastructure*. I. Foster and C. Kesselman, eds. Morgan Kaufmann, 1999.
- [15] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l J. Supercomputer Applications*, 2001.
- [16] G. Fox and S. Pallickara, "An Event Service to Support Grid Computational Environments," *J. Concurrency and Computation: Practice and Experience—Special Issue on Grid Computing Environments*, 2002.
- [17] D. Gunter, B. Tierney, B. Crowley, K. Jackson, J. Lee, and M. Stouffer, "Dynamic Monitoring of High-Performance Distributed Applications," *Proc. 11th IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 2002.

- [18] M.J. Harrold and G. Rothermel, "Performing Dataflow Testing on Classes," *Proc. ACM Symp. Foundations of Software Eng.*, Dec. 1994.
- [19] W. Hibbard, "VisAD: Connecting People to Computations and People to People," *Computer Graphics*, vol. 32, no. 3, pp. 10-12, 1998.
- [20] C. Isert and K. Schwan, "ACDS: Adapting Computational Data Streams for High Performance," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2000.
- [21] C.E. Kilpatrick and K. Schwan, "Using Languages for Describing Capture, Analysis, and Display of Performance Information for Parallel and Distributed Applications," *Proc. IEEE Int'l Conf. Computer Languages*, Mar. 1990.
- [22] F. Kon, R. Campbell, M. Mickunas, K. Nahrstedt, and F. Ballesteros, "2K: A Distributed Operating System for Dynamic Heterogeneous Environments," *Proc. IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, 2000.
- [23] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Trans. Knowledge and Data Eng.*, Special issue on Web technologies, Jan. 1999.
- [24] S. Madden and M.J. Franklin, "Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data," *Proc. Int'l Conf. Data Eng. (ICDE)*, 2002.
- [25] D. Ogle, K. Schwan, and R. Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 7, pp. 762-778, July 1993.
- [26] G. Piatetsky-Shapiro and C. Connell, "Accurate Estimation of the Number of Tuples Satisfying a Condition," *Proc. ACM SIGMOD Conf.*, pp. 256-276, June 1984.
- [27] B. Plale, "Leveraging Run Time Knowledge about Event Rates to Improve Memory Utilization in Wide Area Data Stream Filtering," *Proc. 11th IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 2002.
- [28] B. Plale, V. Elling, G. Eisenhauer, K. Schwan, D. King, and V. Martin, "Realizing Distributed Computational Laboratories," *Int'l J. Parallel and Distributed Systems and Networks*, vol. 2, no. 3, pp. 180-190, 1999.
- [29] B. Plale and K. Schwan, "Run-Time Detection in Parallel and Distributed Systems: Application to Safety-Critical Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 163-170, June 1999.
- [30] B. Plale and K. Schwan, "dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries," *Proc. Ninth IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 2000.
- [31] R. Ribler, J. Vetter, H. Simitci, and D. Reed, "Autopilot: Adaptive Control of Distributed Applications," *Proc. IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 1999.
- [32] B. (Plale) Schroeder, S. Aggarwal, and K. Schwan, "Software Approach to Hazard Detection Using On-Line Analysis of Safety Constraints," *Proc. 16th Symp. Reliable and Distributed Systems (SRDS '97)*, pp. 80-87, Oct. 1997.
- [33] S. Smallen, H. Casanova, and F. Berman, "Applying Scheduling and Tuning to On-Line Parallel Tomography," *Proc. ACM/IEEE Supercomputing 2001*, 2001.
- [34] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *IEEE Trans. Computers*, vol. 37, no. 2, pp. 156-196, May 1988.
- [35] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous Queries over Append-Only Databases," *Proc. Int'l Conf. Management of Data (SIGMOD)*, 1992.
- [36] *Active Database Systems*. J. Widom and S. Ceri, eds. Morgan Kaufmann, 1996.
- [37] R. Wolski, "Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," *Proc. IEEE Int'l High Performance Distributed Computing Symp. (HPDC)*, Aug. 1997.



Beth Plale received the PhD degree in computer science from the State University of New York, Binghamton. She is an assistant professor in the Computer Science Department at Indiana University. Prior to joining Indiana University, she held a postdoctoral position in the Center for Experimental Research and Computer Systems at Georgia Tech. Her research interests include data streams, data driven applications, parallel and distributed computing, relational representation of grid resource information, and database query processing. She is a member of the IEEE and ACM.



Karsten Schwan received the MSc and PhD degrees from Carnegie-Mellon University in Pittsburgh, Pennsylvania. His PhD research in high performance computing concerned operating and programming systems support for the Cm* multiprocessor. At Georgia Tech, he now directs the university-wide CERCS Center for Experimental Research in Computer Systems, jointly with three codirectors from the College of Computing and the School of Electrical and Computer Engineering. Professor Schwan is an associate editor of the *IEEE Transactions on Computers*, and the journal *Concurrency: Practice and Experience and Cluster Computing*. He is a senior member of the IEEE, has held an IBM Faculty Fellowship, was a Fulbright scholar and a member of the Studienstiftung des Deutschen Volkes, received the 2000 Best Research Award at the Georgia Institute of Technology, and has received best paper awards at multiple international conferences.

► For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.