Dynamic Querying of Streaming Data with the dQUOB System

Beth Plale

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

Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
schwan@cc.gatech.edu

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 non-computer scientists to access data.

Leveraging the efficiency of query processing provided by relational queries, the dQUOB system provides a conceptual relational data model and SQL query access over distributed data streams. Queries can 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 assess to 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: wide-area computations, grid computing, data streams, publish-subscribe event channels, SQL, relational data model, database query processing.

1 Introduction

Background. Data-intensive parallel and distributed computations have seen dramatic increases in scale along multiple 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 [28], remote visualization [11], remote instrument control [27], and distributed data analysis [2, 23, 6]. 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 NSF Distributed Terascale Facility (DTF).

The impact on data-intensive computation has been dramatic. Scientists now acknowledge that remote location of a data set is no longer a barrier to its inclusion as a data source. The *Computational Grid (or Grid for short)* [12, 13] addresses the expanding computational base of distributed multi-user workstation clusters and supercomputers with a middleware infrastructure providing services such as communication, security, scheduling, and resource location.

Within the class of data-intensive applications is a subclass of applications characterized by the use of data streaming for distributed communication. Rich media and remote visualization are well known examples, but data streams could exist between any loosely-coupled, autonomous components that communicate asynchronously. In all such applications, streams of events flow from providers to consumers, where an event has no size restriction and contains timestamped data about the behavior or state of a computational entity, physical instrument, or user. Event streaming can be initiated by a data provider (push model) or by a consumer (pull model). Publish-subscribe event communication packages like ECho [8] implement event channels similar to those provided by CORBA [14] but focused on large-scale event flows. The publish-subscribe semantics allow any number of consumers to subscribe to an event channel; the provider need not have knowledge of the location or number of users. Data streams have been treated by others in [10], [4], and [19].

A problem in data streaming applications surfaces when the applications scale in 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 of a form needed by the user. For instance, scientific data generated by an atmospheric transport model may need to undergo a Fourier transform prior to rendering. Similarly, a user may not be interested in all 3D grid data generated, but in an aggregation of the data; a simple 2D plot of a one month trend, for instance.

Approach and Contributions. Our work addresses the needs mismatch problem in large-scale data flows with a novel approach to selectively extracting data from data streams. Earlier work done by our group has established the benefits of encapsulating needs mismatch-style computations into logical tasks that can be associated with data streams to maximize proximity or availability of resources [25, 5, 17, 21]. This paper extends these notions by providing the user with an intuitive relational model for thinking about needs-mismatch computations and a prototype system for creating these computations and embedding them into a data stream. Application of the work first to safety-critical systems[24] followed by scientific computing applications[25] has provided

early confirmation of the usefulness of the abstraction. Extensive experimental evaluation of the prototype system, dQUOB, presented in this paper shows the approach to be feasible.

Our approach is to view data streams as a relational database. A data stream is a flow of events from one or more data suppliers to a single consumer. Located between suppliers and consumer are tasks that execute needs-mismatch computations. A high performance application may contain numerous such data streams. Through the conceptualization of data streams as a relational database, we gain access to the SQL query language and an intuitive way of thinking about data streams. Specifically, we view data streams as data sources that can be operated on in the same manner as a relational database, that is, via a query language. The abstraction provides further value through the declarative nature of the query language, which enables automated stream optimizations not possible with procedural approaches.

Queries over data streams alone is a useful abstraction; their value is enhanced by coupling the query with a procedure that is triggered when the query evaluates to true. The resulting rule, based on triggers in active databases[31], provides users the benefit of queries for joining event streams, filtering, and creating (*i.e.*, materialized) event types, and application-specific algorithms for performing required mathematical computation to transform the data. Furthermore, by uniformly treating all data sources as relations, including monitoring information, the user can write a single query that is responsive to current conditions in the computing environment such as network conditions, or changing user needs or application behavior. Finally, through its formal foundation in relational calculus, SQL has evolved as an efficient query language with broadly accepted, widely used heuristics for query optimization. We leverage this foundation by adapting existing heuristics when appropriate to the data streaming domain.

In summary, the contribution of our work is three-fold:

- abstraction of data streams as a relational database,
- uniform, user-level control of adaptations in response to dynamic changes in data streams, and
- unique data stream optimizations enabled by the abstraction.

Key Challenges. The *data streams as database* abstraction requires us to address several interesting problems. The first issue is to define an appropriate query cost metric. The typical database query cost metric minimizes total cost (CPU and I/O cost). Thus, a traditional query optimizer will select from amongst multiple query plans the one that minimizes disk access times, while factoring in availability of indexes, etc. With disk latency in the millisecond range and CPU clock speeds in the nanosecond range, typical query optimization heuristics attempt to reduce the number of disk accesses. Queries over data streams, on the other hand, are evaluated on tuples (events) that are delivered to the query evaluation engine by the underlying communication infrastructure, thus eliminating disk access time as a significant cost factor. Hence, the query cost metric for data streams minimizes total CPU cost. It is in light of this new query cost metric that we reevaluate the traditional optimization heuristics.

The absence of traditional fixtures like tables of data introduces additional problems. How are joins supported when queries are evaluated on a moving stream of data (*i.e.*, no tables exist)? How are selectivities[1] computed when no data is available at compile time? Further, the streaming data of the domains we consider often contains application state which does not behave according to a normal distribution, so simple selectivity algorithms are not applicable. What is a suitable run-

[1] *Selectivity* is a probability assigned to a particular select operation.

time representation for a query that is amenable to runtime optimization while at the same time executing at rates in the microsecond range? Finally, we can expect event streams with widely differing arrival rates. What is the impact on query efficiency and join processing? These are the problem constraints of the data streams environment, and the ones addressed in this work.

Overview. The remainder of the paper is organized as follows: the relation of this work to similar works is discussed in Section 2. We motivate our work in Section 3 with a discussion of two data-intensive parallel and distributed applications: a distributed, collaborative scientific visualization and a virtual reality system for terrain navigation. In Section 4, we demonstrate the utility of SQL for data stream querying through a set of examples. In Section 5 we discuss the viability of the approach for high performance computing. Section 6 discusses the architecture of the prototype system. In Section 7 introduces a cost metric used in the measurements. Section 8 assesses the efficiency of query evaluation and its effectiveness in reducing end-to-end latency. We conclude with a summary of contributions and future work in Section 9.

2 Related Research

One familiar application of a valid-time database to parallel and distributed computing is the use of temporal SQL for performance analysis of distributed and parallel computations [30, 21, 18]. Queries are issued post-mortem against the performance data[30]. 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[21], one can reduce the amount of data that must be ultimately stored, and the period over which it must be stored.

The Meta Data Management System (MDMS) [7] addresses the difficulty non-computer scientists encounter in using parallel file systems, specifically High Performance Parallel File System (HPPFS). The authors propose a database layer between the scientist and HPPFS that provides a simpler, higher-level interface. User input is stored to the database and used as hints for effective HPPFS storage and retrieval. Like our work, MDMS is a database approach to wide area distributed data access, but unlike our work, focuses on optimal storage and retrieval strategies for persistent data where we focus on optimal retrieval strategies for streaming data. Eddies [3] 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 share the same goal as dQUOB of being responsive to changes in the environment.

Run-time detection in data streams has been addressed with fuzzy logic [27] 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. The Active Data Repository [10] 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 [20] 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.

3 Motivation

In an increasingly connected world, a growing number of data-intensive applications exist for which the data streaming model of communication is appropriate. We introduce two here: distributed scientific visualization and a virtual geographic information system.

3.1 Distributed Scientific Visualization

Atmospheric scientists and computer scientists at Georgia Tech recently coupled a parallel and distributed global atmospheric transport model and a parallel chemical model [23]. 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's interaction with short-lived chemical species (*e.g.*, N_2, NO_3, CH_3O_2). The primary client of the scientific models is a visAD [16] visualization executing on a high-end graphics workstation. The transport and chemical models stream ozone, temperature, winds, and the short-lived species data at each logical timestep, where a logical timestep is 2 hrs. of modeled physical time. A gridpoint is defined by (level, latitude, and longitude) where 'level' is a coarse partitioning of the earth's atmosphere into 37 levels.

Even in the relatively simple model of two data sources (*i.e.* transport and chemical models) that stream data to a single consumer running on a high-end visualization engine, the needs mismatch problem exists. For instance, the transport model computation assumes a spectral representation of the data whereas the visualization (and chemical model) assume a 3D grid representation. Further, the scales of the two models differ, forcing reconciliation prior to visualization. A more useful picture is of numerous clients with varying needs. A second client may desire a 2D graph plotting a trend over 1 month of data. A third, having limited compute resources, may want 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 client. The visualization application is the basis for the examples appearing throughout the paper.

3.2 Virtual Geographic Information System

The Virtual Geographic Information System (VGIS) [9] developed at Georgia Tech is a virtual reality world that allows one to navigate the earth's terrain at altitudes ranging from several thousand miles above the earth to one meter above the earth's surface. In the past, the primary research issues have been in rendering, tracking, and display management. However, the research focus has expanded recently to provide simultaneous support for numerous consumers, with vastly differing resources from high-end visualization engines to laptops.

Support for numerous distributed clients can be provided in a couple ways. Distributed users could query the centralized GIS data store for regional data at startup, then query upon demand as the trajectory through the regional data dictates. But there are drawbacks. Since the VGIS system must concurrently satisfy clients with a steady frames-per-second rate of display, a high query load might hinder performance. Further, the geographic information is maintained in a proprietary data structure so the API and query language would need to be created from scratch. A more viable alternative is to stream geographic information to the clients using a communication model such as publish-subscribe to support multiple clients. Intermediate computation can serve to tailor the data to a particular user's needs.

4 Query Language Through Examples

A strength of the relational view of data streams is that it makes available the SQL query language for extracting data from streams. The dQUOB language incorporates the create-if-then rule construct of the Starburst [31] active database query language. The *create*-clause creates a named rule, the *if*-clause contains an SQL query, and the *then*-clause is a set of actions to be executed when the query is satisfied. By means of a series of examples that follow, we explain the language features.

Example 1: Rule construct, Select. The following rule, named C:1, accepts two event types: data events from an atmospheric model as Data_Ev, and a user request for a particular region of data as Request_Ev. The query contains several select expressions, a select expression is considered as a single comparison. For illustration purposes, the user requested region is only one of two 3D points at selected latitudes. The query evaluates to true if the data event includes one of the requested latitudinal points. The data events satisfying the query are forwarded to the function, ppm2ppb, which converts the concentration values at the grid points of the 3D slice from parts-per-million to parts-per-billion.

```
CREATE RULE C:1 ON Data_Ev, Request_Ev
IF
   SELECT Data_Ev
   FROM Data_Ev as d, Request_Ev as r
   WHERE
      ((r.lat_point1 >= d.lat_min and
        r.lat_point1 <= d.lat_max) or
       (r.lat_point2 >= d.lat_min and
        r.lat_point2 <= d.lat_max)) and
       d.aid = r.aid
   THEN
       FUNC ppm2ppb
```

Example 2: Boolean Operators, Join. Rule C:1 uses the `and` and `or` boolean operators. The negation boolean operator is also supported. When a select consists of operands from two event types, as in `d.aid = r.aid`, an implicit join is performed on the two event streams. Join is a Cartesian product over two tables with an implicit time-based condition. A pair of events satisfies the join if the time-based condition evaluates to true. Specifically, for two events α and β, the Cartesian product, $\alpha \times \beta$, is taken if the timestamp of α equals the timestamp of β plus or minus some small ϵ.

Joins over data streams is a key challenge of the work. Because of the limits of on-chip and L2 cache memory, the serious performance delays incurred when disk files are used for buffer storage, and the large event sizes found in practice, a table will often not hold every event received up to a particular point in time. Consequently, queries must be evaluated over partial data. The set of events in a table constitute a *sliding window* over the table. As the size of the sliding window decreases, the risk of false negatives increases. A *false negative* is a match that should have been made that is missed [20]. A reasonable window size depends strongly on the temporal ordering of the two input streams used in the join and on their relative synchrony[30]. For two event streams, each with a guaranteed partial order and containing an event for every logical timestep, minimal storage would be required. But if ordering or relative synchrony are absent, the window size is potentially unlimited. The size of the sliding time window is determined at startup based on the amount of memory available for the in-memory buffers.

Example 3: Project. dQUOB supports *materialized views*, the stored results of a query. In the context of data streams, a materialized view can be viewed as an event stream originating at the query. In the following example, a user wishes to be alerted when a particular condition becomes true. The alert message, Notif_Ev, is a relation that originates at the query; that is, the query is the source for events of this type. The notification event is created by a project operation when a user event is received *and* the model has advanced to logical timestep 4270.

```
CREATE RULE C:3 ON Data_Ev, Request_Ev
IF
   SELECT Notify_Ev as d.timestep, d.timestamp,
                r.userName
   FROM Data_Ev as d, Request_Ev as r
   WHERE
       d.timestep = 4270 and d.aid = r.aid
```

Example 4: Temporal and Time-based Operators. dQUOB supports the 'meets' and 'precedes' temporal operators as defined in the temporal relational database query language, ATSQL [29]. The expression α MEETS β in the example evaluates to true if there exists two events α and β such that the start time of β is less than or equal to the end time of α but not less than the start time of α. The expression α PRECEDES β evaluates to true when for an event β there exists an event α in the sliding window such that the timestamp of α is earlier than the timestamp of β. As an example, consider a stream of frames, each marked by a start and end time. The user wishes to perform a difference over a pair of sequential frames. The query results in a pair of sequential frames that are passed to the difference function, diff(). The resulting difference frame is generated as Frame_Diff_Ev.

```
CREATE RULE C:4 ON Frame_Ev
IF
   SELECT Frame_Diff_Ev
   FROM Frame_Ev as f1, Frame_Ev as f2
   WHERE
       f1 meets f2
THEN
       FUNC diff
```

In dealing with time-dependent data, the database community distinguishes between a snapshot event and an interval event.[2] The *snapshot event* is a snapshot of application behavior and is tagged with a single timestep. The *interval event*, on the other hand, describes behavior existing over a duration, and is bounded by a start time and a stop time. dQUOB supports the event distinctions. In Section 8.2 we discuss an optimization made possible by the distinction.

Discussion. This section has described the supported SQL constructs and dQUOB rule syntax through examples, with the purpose of establishing the ease of use of the language for non-computer science end users. These constructs have evolved through their successive application to multiple domains over the past five years. In the domain of safety critical systems, dQUOB was used to detect hazard conditions occurring when autonomous robots cooperate to achieve a task. The data streams generated by the robots consist of frequent state updates and small event sizes (*i.e.*, several kilobytes). The temporal operators were introduced at this time to detect and respond to state transitions in the robots. In the succeeding domain of high performance computing (*i.e.*, online visualization and collaboration in scientific computing), the data streams consist of large

[2]The temporal database community refers to 'event' tuples and 'interval' tuples, but because of the awkwardness of the term 'event event', we use 'snapshot event' and 'interval event'.

events (up to 2.7Mb) generated frequently at a relatively uniform rate and small events generated infrequently and randomly. dQUOB queries are used to extract a tailored stream of data and adapt it to changes in user demand and network resources. This latter domain motivated the distinction between snapshot and interval events that enabled optimizations to join processing. Our goal is pragmatic: implement support for a language feature when its need is determined by the application. The language can easily be extended, to support union for instance, when it is determined through application use that the additional expressibility is needed.

5 Mapping Database Queries to Data Streams

dQUOB models event data transfer and provides a software system by which one can create computational entities to manipulate data streams. A data stream is a logical group of one or more event channels, one or more suppliers, a single consumer, intermediate query-driven computation called quoblets, and a relational schema. Data streams are built on top of ECho [8], an efficient publish-subscribe event channel implementation. Data suppliers and consumers are linked via logical event channels that permit suppliers to publish events unbeknownst of the types and numbers of consumers.

The relational data model consists of data organized into relations (or tables). A relation consists of a set of attributes; an attribute corresponds to one column of the table. Associated with an attribute is a domain indicating the set of values that the attribute can take. Each row of a table is called a tuple and describes one real-world entity. A schema defines the tables, attributes, and domains for a given application [33]. Data is retrieved from a relational database by means of SQL queries.

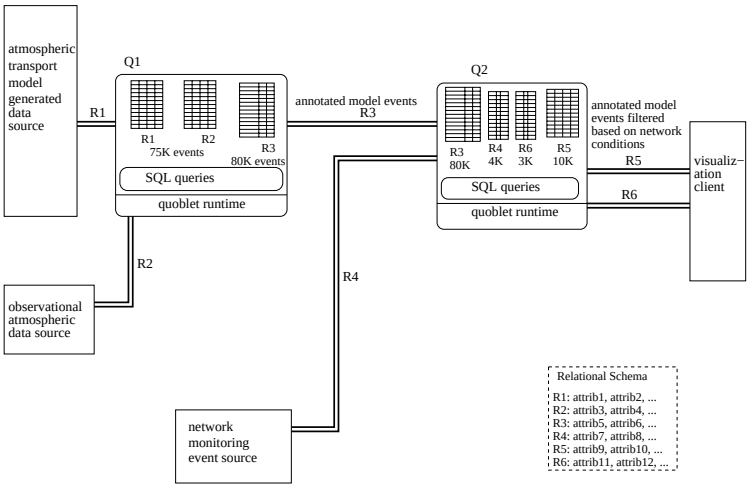

Figure 1. dQUOB data stream for visualization client.

The relational data model maps to the underlying event channel architecture. The domain of the relational data model consists of every event in every event channel existing between application startup and termination. Mapping begins at the lowest level: 'event' and 'tuple' are synonymous. That is, an event in an event channel is viewed as a tuple in a table. Given an event of type T and a tuple as a member of a relation R, there is a one-to-one relationship between the relation R and

event type T. The event channels are typed [8], meaning they carry events of a single type. As such, a relation can contain tuples (*i.e.* events) from multiple event channels.

An SQL query executes its operators (*e.g.*, select, project) in a pipeline, the viability of which is established in [30]. A query iterates continuously, triggered by the arrival of an event. When the query evaluates to true, the set of events that satisfy the query are passed to the action routine. The default behavior is to forward the results of the action routine downstream. The notion of a query "embedded into a data stream" is realized by a quoblet, a process, which evaluates one or more SQL queries.

Figure 1 depicts a single data stream. Three suppliers push events down the event channels R1, R2, and R4. Quoblet Q1 performs a join over model generated events and observational event data. The result is annotated model events that are pushed downstream on R3. Quoblet Q2 filters annotated model events based on infrequent network monitoring events received on R4 and on infrequent user requests for a particular atmospheric region of data on R6. The results are forwarded to the client on event channel R5. A single relational schema exists to describe the event types. It is conceivable to combine Q1 and Q2, but separating them provides the opportunity for another client to use the stream join provided by Q1.

A key challenge is the viability of continuous execution of queries over large streams of events that vary in size from a few hundred kilobytes to several hundred megabytes, and have arrival frequencies that vary from sporadic to continuous. The following measurements determine the viability of the approach.

Microbenchmarks. The microbenchmarks 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 an Sun UltraSPARC 1.

The results are as follows: *gate, 4.7ms; projection 8.6ms; selection 1.6ms; join 0.5ms*. The *gate* operator routes incoming events to query operators. It appears costly for its simple purpose, but reflected in the *4.7ms* is additional quoblet-wide overhead. The relatively high cost of the projection operator at *8.6ms* is attributed to projection's space allocation for a new event and copying of attribute values from existing events.

The microbenchmark results translate to a throughput of 62.5MB/sec on a newer Sun Ultra 30 for a "typical" query consisting of 13 operators, three of those join operators. All joins execute Cartesian product over one *snapshot event* table and one *interval event* table; the sliding window size is 120, the event size is 75K. We anticipate that additional optimizations to reduce memory management and data copying costs will drive throughput to rates commensurate with emerging wide area networks.

Query Optimization in Application Setting. To determine query processing costs as a percentage of total quoblet time, we use a quoblet with one E-A rule, that rule having a non-trivial query whose 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 algorithms like these, which perform a simple arithmetic operation on each grid point, to have mid-range computational needs.

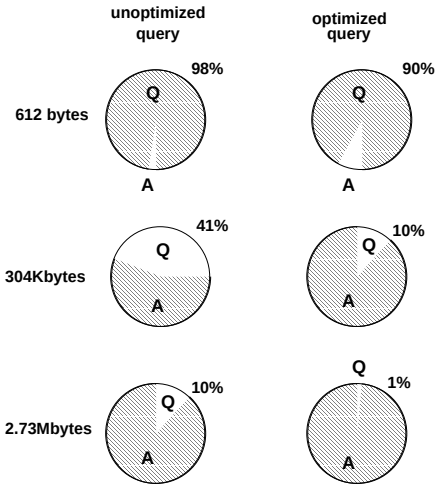

Figure 2. Breakdown of quoblet execution time by query (Q) and action (A).

Figure 2 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 total quoblet time varies widely, depending on computational cost of the user supplied action. At the 612 byte event size, query processing consumes 98% 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 2.73 Mbyte event size, the event contains 9 species, so all but 612 bytes of attribute information must be converted. In this case the query represents 10% of total execution time. The second column shows query execution time for the same query that has been optimized.

This section has demonstrated the viability of the concept of *data streams as database* for big-data streams. The results show that in an application setting with meaningful queries, realistic event sizes, and mid-range computational needs, optimized queries consume a small portion of total execution time. That is, when a moderately complex query is coupled with a mid-range computation (*i.e.*, touches every datapoint in a 3D grid event), the query consumes only 10% of the total computation time. The data streams as database concept yields additional utility and advantages, but before discussing these, we first introduce the dQUOB realization.

6 System Architecture

A key challenge in the system architecture is the need for a query representation and deployment strategy that satisfies the dual needs of portability and optimizability. That is, a wide area, long running application is by nature dynamic, in part because of dynamics in the underlying execution environment. Wide variations in end-to-end latency is but one example. For dQUOB this translates into a requirement to support adding and removing queries on-the-fly, and also reoptimizing queries in response to changes in the data. Regarding the latter, a data generating scientific model may undergo an major mode transition that would change the nature of the data generated. This type of change should trigger query optimization because it is likely that there now exists a more optimal version of the query than the one in current use.

The system architecture consists of a dQUOB client which resides at the user and is the vehicle through which the user creates event-action rules (*i.e.*, colloquially referred to as queries). The dQUOB server is a centralized, application-wide service and repository that accepts E-A rules from clients and deploys them at quoblets. Quoblets are the executable entities that are embedded into data streams.

Code Generation. The dQUOB client is a query compiler that optimizes a query through application of optimization heuristics to generate a portable, intermediate representation. Plan generation and selection, common steps in database query optimization, are not required largely because of the reduced importance on disk access time. To a lesser degree, we make the simplifying assumption that the heuristics yield a query sufficiently optimal for our needs. The client sends the intermediate representation to the dQUOB server via HTTP 1.1.

To meet the demands of continuous query evaluation at high data streaming rates, queries must execute as compiled code. Our approach centers around a class library at the quoblet used to construct a compiled query as a set of instantiated objects (*i.e.*, one for each select, project, etc. in the query) linked as a directed graph. It follows then that the generated code must be portable and capable of interfacing with the class library. A scripting language satisfies both needs, so we settled upon Tcl scripts. The script consists of a series of calls to the dQUOB library, with the order of calls determined by the relational calculus representation of the query. Scripts are by nature portable and small. A script for a query is roughly one-tenth the size of the compiled code representation [25].

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. 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 lacks support for on-the-fly query modification and require additional system calls for dynamic linking.

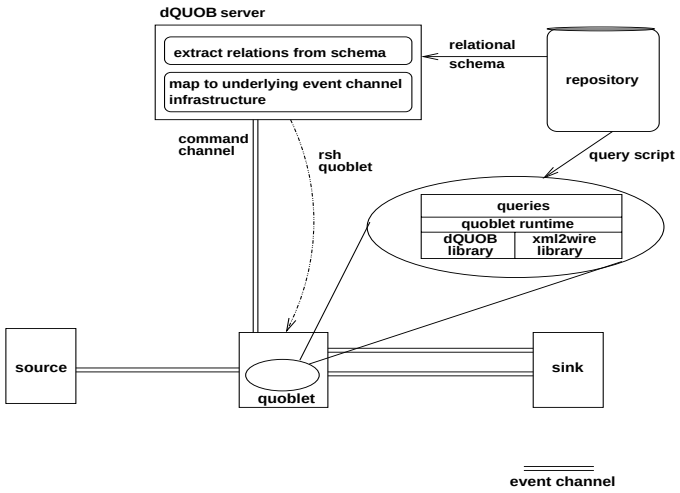

Figure 3. Code deployment.

Code Deployment. Code deployment, depicted in Figure 3, involves a query script, the class library, and the quoblet runtime environment. A quoblet is a task 'shell' for executing event-action rules. The quoblet runtime performs event handling, graph traversal, query instantiation and reoptimization. At startup a quoblet contains no queries and subscribes to a single event channel established between itself and the dQUOB server. The dQUOB server, shown at the top of Figure 3, passes the quoblet a Universal Resource Locator (URL) for the data repository. The quoblet issues an HTTP request to retrieve the query. It then passes the script to its resident Tcl interpreter, resulting in an instantiation of the query as compiled code. The query, once instantiated, registers itself with the quoblet runtime and begins receiving events. Query execution is performed as a depth-first traversal of the directed query graph. Multiple queries are linked as a directed graph that is also traversed depth-first.

7 Cost Metric

In scientific collaboration and remote visualization, it is important to present data to end users in quasi-realtime, therefore the metric to evaluate query/computational data stream performance is an end-to-end measure of user perceived performance. The metric, called *Effective Real Time* (ERT), defines the average rate at which a user can expect to see the data of interest. Since it is user perceived performance we address, the metric is expressed in the application-dependent measure of events/second. For example, an acceptable refresh rate for movement through geographical terrain is 32 frames (or events) per second, so this would be a desired ERT.

Effective Real Time. ERT of a computational data stream is the ratio of the number of events received to the total elapsed time of the application.

$$\mathsf{ERT} = \frac{number\_of\_events\_received}{total\_elapsed\_time}$$

Let the number of events received at the user be *m*. Let the total number of events generated by the source be *n*, and let *i* refer to the i^{th} event, where $m <= n$ and $0 < i <= n$. The total elapsed time is the time elapsed between the generation of n_1 and receipt of the mth event at the consumer. Assuming sequential transfer of events, and a communication model of one source, one client, and a single quoblet between, then the total elapsed time is the sum of the end-to-end latencies of *m* events, where the latency of event *i* is defined as follows:

$$\mathsf{latency_i} = t_{s\prime} + t_q + t_{q\prime}$$

$t_{s\prime}$ is the time to transfer an event from a supplier to a quoblet; including the cost of event gathering, buffering, and sending; t_q is the time consumed by the quoblet to perform a single query evaluation and action execution over one data event. It assumes no blocking on input or output. $t_{q\prime}$ is the time required to transfer the resulting event to the client. Event delivery is complete when the event arrives at the consumer. Then ERT can be expressed mathematically as:

$$\mathsf{ERT} = \frac{m}{\sum_{i<=n} latency_i}$$

The relationship between ERT and throughput is straightforward, where high ERT implies high throughput:

$$throughput = \mathsf{ERT} * event\_size$$

Since ERT is a user's *perception* of performance, we have flexibility in maintaining ERT at satisfactory levels. The most obvious is to decrease *m*, the number of events received, by placing stronger filtering in the data stream. Another might be to perform a compression or reduction computation on the data upstream such that less data travels the downstream path. The two can be coupled to provide a powerful tool for reducing downstream bandwidth. But bandwidth is not the only bottleneck. Computation time can also be reduced at points between data generation and consumption. As measurements in Section 5 reveal, total quoblet execution time is dominated by action computation. Thus an interesting performance benefit enabled by coupling query and action is that, taken over a stream of events, the execution cost for an event/action rule decreases as the probability that the query evaluates to true increases. Specifically, quoblet execution time is measured as the cost of the query t_{query}, fixed overhead $t_{overhead}$ cost, plus the cost of executing the action t_{action} subject to the probability that the query evaluates to true. This is expressed as:

$$t_q = t_{query} + (t_{action} * P(query)) + t_{overhead}$$

That is, the dominant computation cost can be reduced by introducing a stronger query, thus increasing ERT for the user. It should be noted that t_q is a worst case measure of quoblet processing time as it does not account for the opportunities for concurrency in an SMP that exist in the prototype implementation.

8 Experimental Evaluation

8.1 Query Optimization Heuristics

Recall that an advantage of dQUOB over a procedural language approach is the declarative query language that enables optimizations to transform the query from its original user-specified form. This feature is highly desirable since users are application scientists who are not likely to be interested in the most optimal way to state a request. This section considers four algebraic heuristics evaluated in light of the query cost metric for data streams. 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.

Pushing Selects. Pushing selects down the query tree is a common database optimization heuristic. Our experimental evaluation of the utility of this optimization technique for data streams 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 1's connected via a 100Mbps switched Ethernet. In the unoptimized query case, operator location in the parse tree is dictated by the order explicit in the procedural relational algebraic expression and as such might be the specification by a naive user. The optimized version of the parse tree results from a compiler optimization pass that pushes selects down the parse tree wherever possible.

Push Selects	*Optimized*	*Unoptimized*	*Factoring*	*Factoring*	*No Factoring*
execution time (secs)	0.14427	1.08329	execution time (secs)	0.22191	0.23224
number ops	2	51	number ops	1.97	4.28
% by op (ovhd/project/select/join)	.49/0/.51/0	.02/0/.11/.87	% by op (ovhd/project/select/join)	.33/.33/.33/0	.31/.07/.62/0

Table 1. Results of applying query optimization heuristics to query parse tree.

The left side of Table 1 shows the execution times for unoptimized and optimized queries. The execution time is for query execution over a single event. 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 execution down into percentage by operator type. For instance, in the unoptimized case 2% of the operations are overhead, 11% are selection, and 87% are join. Optimization yielded an 87% reduction in query execution time and a 96% 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% of the operations are attributed to join whereas that number is reduced to zero in the optimized version. That selects filter 100% of the events so none reach the join operation is best case.

Factorization. To determine the performance gains achievable through inter-node 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 is 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 1's connected via a 100Mbps switched Ethernet. The workload is the autonomous robotics stream described earlier.

The results, expressed on a per-event basis, appear on the right side of Table 1. 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. This reduction in number of operations is not matched by a proportional drop in execution time. Insight to the problem can be gained by the breakdown by operation type. In the no factoring case, project constitutes 7% of the total operations, whereas in the filter case it accounts for 33%. The disparity indicates that the gain achieved through reduced number of select operations is largely consumed by increased projection costs which we have determined to be costly. We conclude that factoring a common subexpression is not a beneficial heuristic for queries that share an address space. Distributed queries could benefit by pushing a common subexpression upstream, resulting in reduced compute cycles and bandwidth needs. We expect such a heuristic to further improve performance in the distributed case.

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 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 can 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 in total from a socket, and the relatively high cost of the projection operation, we rejected the heuristic of pushing projects down the parse tree.

Reordering Selects. The final query optimization heuristic is based on statistical metrics about the data. 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, particularly *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 under the assumption that the data behaves according to a normal distribution. 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 '80s by Shapiro [22]. 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 correctness cannot be compromised by an incorrect operator ordering. 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 [15].

Our experimental evaluations demonstrate that query optimization techniques common to databases also apply, though selectively, to data streams. Particularly, pushing selects down the parse tree yielded significant improvements in query evaluation time. Pushing projects, on the other hand, had a detrimental impact on performance. Factorization was 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 useful.

8.2 Overhead of Dynamic Run-time Query Representation

We quantify the overhead of using a graph representation for queries by comparing the performance of a dQUOB query against a baseline. 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 dQUOB; the second uses a C++ function compiled into the quoblet. The latter is referred to in Table 2 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.

dQUOB Query

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)
1	34.07	87.00	121.07
10	32.35	87.31	119.66
120	32.43	84.56	116.99

'Static' Query

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)
1	16.65	2.78	19.43

Table 2. dQUOB versus 'static' query cost per data model event.

The dQUOB query processing cost is given at the top of Table 2 for three sizes of join windows. The results are broken out by quoblet overhead and query evaluation time. 'Total' is the sum of the Overhead and Query columns. Total time to execute a dQUOB query over a single atmospheric data event is in the 120μs range. Other sample queries exectued independently confirm this number. Compared to the total execution time for a static query in the 20μs range, we see a dQUOB query processing overhead of roughly 6x.

It can be observed in the dQUOB query case that the smaller join window performed worse 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 lies in an optimization to join processing enabled by the distinction between *snapshot events* (*i.e.,* behavior of an application at an instant) generated by the atmospheric model and *interval events,* (*i.e.,* events describing behavior existing over a duration) generated by the client and performance monitor. Specifically, the join window of an interval event need be no larger than size one, regardless of the window size of the second stream. Cartesian Product over an event pair, then, requires at most one tuple evaluation. The decrease in execution time shown at the larger buffer sizes is not a decrease in join cost, because that cost is constant across buffer sizes, but the cost of storage reclamation. At size 120, the window is large enough to hold every data event generated, so no storage reclamation is performed.

8.3 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 the utility by testing the behavior of a set of cooperating queries over time. It demonstrates the performance advantage by quantifying the impact of quoblet location in a WAN environment.

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 [32], application data, and specific user requests. Responsiveness to changes in environment resources is based on our observation that resources (*e.g.*, end-to-end latency) can often be described by a small number of states. A user expresses data needs under different resource conditions by a set of queries, where each query is responsive to one resource state. Specifically, queries can be written as the conjunction of the application data portion of the query to a check on resource state. In this way, if the resource state check fails, the 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 execution environment. We employ a quoblet containing three queries: a 'no filtering' query called 'null query', query with moderate filtering capability, and one that applies strong filtering. The impact on end-to-end latency over time is captured in Figure 4. 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 null query is deactivated and 'moderate filtering' query activated through receipt of a management event. The newly active event 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 resource state change event is received from a resource monitor, notifying of a significant degradation in end-to-end latency. The resource event triggers the automatic turning off of the moderate filtering query and turning on of the 'strong filtering query', the latter performs strong filtering, resulting in 2K events being generated.

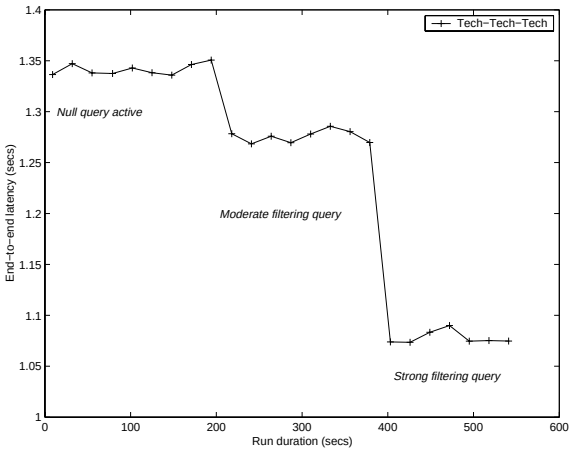

Figure 4. Quoblet potential for responsiveness in dynamic environment.

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 conclusions can be drawn. The test was run in a local area environment, thus the dominant variable in ERT is quoblet computation time, t_q. As we show, in a WAN evironment other variables

can 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.

Impact of Quoblet Location. The second experiment evaluates the impact of quoblet location in a WAN environment employing resources at Georgia Tech, the Albuquerque High Performance Computing Center (AHPCC), and NCSA at University of Illinois Urbana-Champagne. Using a model of single supplier, single consumer, and intervening quoblet, we undertook measurements with the quoblet at the source and at the client. These are shown respectively as Tech-Tech-AHPCC and Tech-AHPCC-AHPCC in Figure 5 and Tech-Tech-NCSA and Tech-NCSA-NCSA in Figure 6. The Tech-Tech-Tech plot is the baseline case from Figure 4. The link between Georgia Tech and the other sites is Internet 2. 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 NCSA machine is a SGI-CRAY Origin 2000 Array, 195MHz, R10000, 48 processors, with HIPPI interconnect between machines. The Georgia Tech machine is a SunUltra 30 with gigabit Ethernet connectivity to Internet 2.

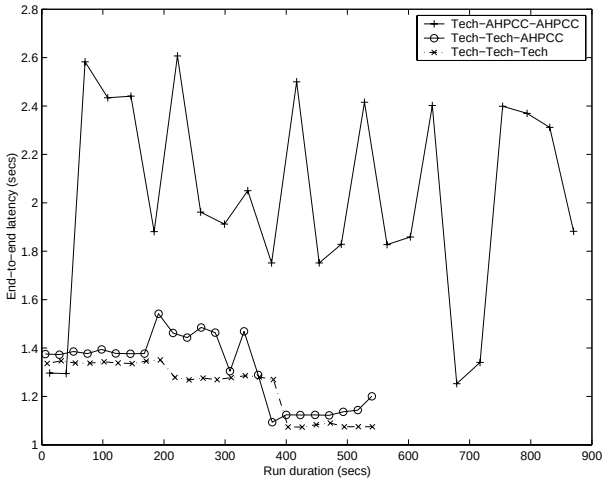

Figure 5. WAN measurement (Georgia Tech -> AHPCC)

In both cases, there is a benefit to pushing data manipulation closer to the source. In Figure 5, with the quoblet located at the client, ERT is 0.614 events/second. With the quoblet at the source, ERT increases to 0.998 events/second. This translates to 372 kbps when filtering at the client and 648 kbps when filtering at the source. It is interesting to note that pushing the quoblet toward the source results in performance that more closely patterns the LAN run over the duration of the application run. This we believe to be a functon of traffic flow control in TCP. This gives us encouragement that filtering early in the data stream can be effective not only in reductions in execution times that are under dQUOB control, but in ensuring more predictable overall WAN behavior. We are currently exploring this. The spiked behavior in Figure 6 observed when locating the quoblet at the source at the NCSA Origin machine we attribute to exponential-backoff style flow control at the Origin.

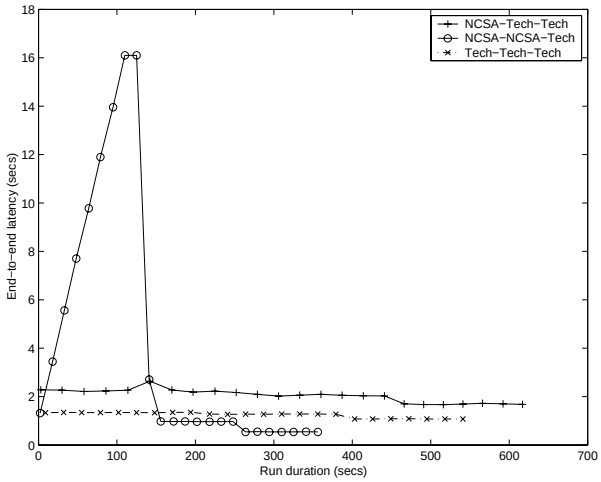

Figure 6. WAN measurement (Georgia Tech -> NCSA)

9 Conclusion

We have described dQUOB, a system and conceptualization for thinking about data in data streams. It addresses an open problem of how to extract manipulated data from a data stream. By conceptualizing the data streams in a distributed peer-to-peer application as a relational database, one can view extracting data as the specification of one or more queries. Through database joins, streams can be combined; through materialized views, new streams can be created.

We developed a prototype implementation, called dQUOB consisting of a compiler through which the user creates queries, a server that deploys queries, and quoblets containing a library and event handler that constitute the quoblet's query evaluation engine. We feel a strength of the conceptualization is its naturalness.

Measurements performed in LAN and WAN settings have confirmed the usefulness of the approach, particularly substantiating the need for query optimization, the benefit of integrating a query and a user specified action into a single E-A rule, and the viability of controlling user's ERT in response to changes in the underlying execution environment.

Efficient query processing is crucial in a data streams environment where queries are continuously iterated over incoming tuples. SQL has several strengths that enable efficient query processing. For one, SQL is declarative and relational operators associative so queries can be optimized (*i.e.*, operators reordered) from the original user specification. Contrast this to the procedural query languages for X.500 and LDAP. Since scientists are likely to not take care to specify the most optimal query, this feature is of key importance. Second, the relational algebra defines a single type; the relation. Most object-based query languages, while providing SQL-like syntax, have difficulties achieving efficient query evaluation, in part by the existence of nested references. We felt that efficient query evaluation outweighed the benefits of supporting nested references.

A limitation of the relational scheme is its lack of support for complex data types. Scientific data is frequently complex, consisting of arrays, complex numbers, etc., but relational attributes are limited to simple data types. We address this issue by representing array data as a single attribute that is opaque to the query engine but are exploring object-relational data model features

to extend query expressibility to complex data while not compromising efficiency.

In ongoing work we are considering alternate schemes for extremely large data events. At larger event sizes, quoblet performance suffers because the quoblet copies the full event into user space. Depending on the type of attributes accessed in the query and how much data is touched by the action function, the full copy may not be needed. At least, one could copy the minimal set of attributes needed to perform the query, then delay copying the remaining data until it is determined whether or not the event satisfies the query.

Future work also addresses interoperability of dQUOB with XML by showing how an XML schema can be transformed into a relational data model schema. We argue that the wealth of optimizations enabled by the relational model do not exist at present for XML's hierarchical model, thus a transformation from XML into the relational domain is an appropriate approach to interoperability[26].

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. http://plaza.ufl.edu/alallen/griphyn/index.html, 2001.

[3] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*, 2000.

[4] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[5] Fabián E. Bustamante and Karsten Schwan. Active I/O streams for heterogeneous high performance computing. In *Parallel Computing (ParCo) 99*, Delft, The Netherlands, August 1999.

[6] 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. *To appear in Journal of Network and Computer Applications, special issue on Network-Based Storage Services*.

[7] A. Choudhary, M. Kandemir, J. No, G. Memik, X. Shen, W. Liao, H. Nagesh, S. More, V. Taylor, R. Thakur, and R. Stevens. Data management for large-scale scientific computations in high performance distributed systems. *Cluster Computing: Journal of Networks, Software Tools and Applications*, 3(1):45–60, 2000.

[8] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. Event services for high performance computing. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.

[9] Nickolas Faust, William Ribarsky, T.Y. Jiang, and Tony Wasilewski. Real-time global data model for the digital earth. In *International Conference on Discrete Global Grids*, 2000.

[10] 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.

[11] Ian Foster, Joseph Insley, Gregor von Laszewski, Carl Kesselman, and Marcus Thiebaux. Distance visualization: Data exploration on the grid. *IEEE Computer*, 32(12), December 1999.

[12] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[13] Ian Foster, Carl Kesselman, and Steve Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.

[14] Object Management Group. Common object request broker architecture (CORBA) event service specification. http://www.omg.org, 2000.

[15] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *ACM Symposium on Foundations of Software Engineering*, December 1994.

[16] W. Hibbard. VisAD: connecting people to computations and people to people. *Computer Graphics*, 32(3):10–12, 1998.

[17] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[18] 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.

[19] Fabio Kon, Roy Campbell, Marshall Mickunas, Klara Nahrstedt, and Francisco Ballesteros. 2k: A distributed operating system for dynamic heterogeneous environments. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.

[20] 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.

[21] 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.

[22] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *ACM SIGMOD Conference*, pages 256–276, June 1984.

[23] Beth Plale, Volker Elling, Greg Eisenhauer, Karsten Schwan, Davis King, and Vernard Martin. Realizing distributed computational laboratories. *International Journal of Parallel and Distributed Systems and Networks*, 2(3), 1999.

[24] Beth Plale and Karsten Schwan. Run-time detection in parallel and distributed systems: Application to safety-critical systems. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 163–170, June 1999.

[25] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[26] Beth Plale, Patrick Widener, and Karsten Schwan. Taking the step from meta-information to communication middleware in computational data streams. In *IEEE Heterogeneous Computing Workshop (HCW)*, April 2001.

[27] Randy Ribler, Jeffrey Vetter, Huseyin Simitci, and Daniel Reed. Autopilot: Adaptive control of distributed applications. *IEEE International High Performance Distributed Computing (HPDC)*, August 1999.

[28] Shava Smallen, Henri Casanova, and Francine Berman. Applying scheduling and tuning to on-line parallel tomography. In *Supercomputing 2001*, 2001.

[29] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*. Springer-Verlag, 1998.

[30] Richard Snodgrass. A relational approach to monitoring complex systems. *IEEE Transactions on Computers*, 6(2):156–196, May 1988.

[31] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems*. Morgan Kaufmann, 1996.

[32] Rich Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *IEEE International High Performance Distributed Computing (HPDC)*, August 1997.

[33] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998.