

# dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries

Beth Plale and Karsten Schwan

College of Computing  
Georgia Institute of Technology

## Abstract

*The dQUOB system satisfies client need for specific information from high-volume data streams. The data streams we speak of are the flow of data existing during large-scale visualizations, video streaming to large numbers of distributed users, and high volume business transactions. We introduce the notion of conceptualizing a data stream as a set of relational database tables so that a scientist can request information with an SQL-like query. Transformation or computation that often needs to be performed on the data en-route can be conceptualized as computation performed on consecutive views of the data, with computation associated with each view. The dQUOB system moves the query code into the data stream as a quoblet; as compiled code. The relational database data model has the significant advantage of presenting opportunities for efficient reoptimizations of queries and sets of queries.*

*Using examples from global atmospheric modeling, we illustrate the usefulness of the dQUOB system. We carry the examples through the experiments to establish the viability of the approach for high performance computing with a baseline benchmark. We define a cost-metric of end-to-end latency that can be used to determine realistic cases where optimization should be applied. Finally, we show that end-to-end latency can be controlled through a probability assigned to a query that a query will evaluate to true.*

## 1 Introduction

**Motivation.** With ever-improving network bandwidths, end users increasingly expect rapid access to remote rich content, such as complex scientific and engineering data, image data, and large files, prompting the development of network-level solutions for efficient data transmission and routing, of operating system support

for communication scheduling, and of user-level support ranging from middleware for HPC applications to server systems for the web. Our group is creating middleware solutions for HPC applications; specifically for data flows created when clients request data from a few sources and/or by the delivery of large data sets to clients. Applications requiring such support include video-on-demand for numerous clients, “access grid” technology in support of teleconferencing for cooperative research, and distributed scientific laboratories in which remotely located scientists and instruments synchronously collaborate via meaningful displays of stored, captured, or generated data and computational models that use or produce these data [11, 14]. Further applications include the large data streams that result from digital library systems such as the National Information Library (NIL) library, a 6600 terabyte library that services 80,000 complex queries and 560,000 simple browser searches a day.

**The dQUOB System.** The *dynamic Query Object* (dQUOB) system addresses two key problems in large-scale data streams. First, unmanaged data streams often over- or under-utilize resources (*e.g.*, available bandwidth, CPU cycles). For instance, a data stream may consume unnecessary network bandwidth by transmitting high fidelity scientific data to a low fidelity visualization engine. Second, the potential benefits derived from managing streams (*i.e.*, through added computation to filter, transform, or compose stream data) can be difficult to attain. Specifically, it can be difficult to customize computations for certain streams and stream behaviors, and the potential exists for duplication of processing when multiple computations are applied to a single data stream.

The dQUOB system enables users to create queries for precisely the data they wish to use. With such queries may be associated user-defined computations, which can further filter data and/or transform it, thereby generating data in the form in which it is most useful to end users. Query execution is performed by

dQUOB runtime components termed *quoblets*, which may be dynamically embedded ‘into’ data streams at arbitrary points, including data providers, intermediate machines, and data consumers. The intent is to distribute filtering and processing actions as per resource availabilities and application needs.

**Key Concepts.** The goal of the dQUOB system is to reduce end-to-end latency by identifying and forwarding only *useful data*. Useful data is, broadly speaking, data of interest to a scientist. For example, atmospheric scientists investigating ozone depletion over the Antarctic may consider useful data to be the multi-month period of data at the tropics (not everywhere else on the earth), where ozone gradually rises from the troposphere to the stratosphere.

dQUOB conceptualizes useful data and its extraction from a data stream as a set of relational database tables<sup>1</sup>. Users request such data with SQL-like queries, and computations performed on the data are applied to consecutive data *views*; computation is associated with each such view.

Abstract data models for event-based computing are being explored by other groups, particularly in the context of XML. One advantage of using an abstract model is the ability of its implementation to span administrative domains, a key requirement in wide-area computing [6]. Furthermore, data models like the relational and the XML Query data model are supported by declarative query languages, which in turn present opportunities for optimizing data streams.

The strength of our work in addition to the general advantages of a common data model and declarative query language is three-fold. First, dQUOB presents a methodical and rigorous approach to formulating, implementing, and executing queries that permit users to focus on the data that is most ‘useful’ to them, thereby potentially eliminating large amounts of unnecessary data processing and transfers. Second, relational query languages, being a well established research area, form a solid basis from which to leverage further stream optimizations. Leveraging such work, however, does not imply simply adopting it, for the simple reason that dQUOB operates on different forms of data: on data flows rather than tables. Furthermore, the stream optimizations we target go beyond traditional database optimizations to include (1) stream filtering based on conditions defined at runtime, (2) fine-grain changes to optimize existing queries, and (3) larger-grain changes, such as the elimination of redundant stream processing actions, changes to the order in which actions are

---

<sup>1</sup>A relation can be thought of as a table, where attributes are column headers defining fields (SpeciesID, SpeciesName, SpeciesConcentration) and tuples are instances in the table (15, Ozone, 42).

executed, or reconfiguration of the query processing engines themselves. The third and final strength of our work is in the movement of the query code into the data stream in the form of an efficient, compiled-code *quoblet*.

**Contributions.** The specific contributions of this paper are fourfold. First, we demonstrate that by embedding queries into large data streams, it is possible to reduce the end-to-end latency and increase throughput between data providers and consumers for the data that is most useful to end users. Second, using examples from global atmospheric modeling, we establish the viability of the dQUOB approach for high performance applications. Third, we define a cost-metric of end-to-end latency that can be used to determine where and how optimization should be applied to data streams. Finally, we show that end-to-end latency can be controlled by dynamically determining and assigning to each query the probability that it will evaluate to ‘true’.

That dQUOB is lightweight is evidenced by its ability to sustain a generation rate of 10 Gbps for events of several hundred kilobytes in size to 90 Gbps for events of several megabytes. dQUOB’s ability to reduce end-to-end latency is demonstrated by a 99% reduction achieved by replacing a weak condition with a strong one, thereby improving the query’s filtering ability. Similar results were achieved across the Internet with an ad-hoc implementation of queries described in [8]. Finally, our results show that, using an application-realistic action, an unoptimized query can consume up to a startling 90% of quoblet execution time, thereby demonstrating the importance of runtime reoptimization. The opportunities presented by such optimization are demonstrated in earlier results [12], which show that reoptimization can reduce query evaluation time by an order of magnitude.

**Overview.** We motivate our research with examples drawn from global atmospheric modeling in Section 2, followed by an overview of the dQUOB system in Section 3. The cost metric used to evaluate performance is developed in Section 4. The experiments appear in Section 5. Related work is discussed briefly in Section 6, and concluding remarks appear in Section 7.

## 2 Motivating Example

Our work is motivated in general by the data streams created during visualization of large-scale data from engineering or scientific simulations[11, 1]. Our particular application is 3D atmospheric data generated by a parallel and distributed global atmospheric model [9], simulating the flow of a chemical species,

specifically ozone, through the stratosphere and interaction of ozone with short lived species (*e.g.*,  $CH_4$ ,  $CO$ ,  $HNO_3$ ). A logical timestep is 2 hrs. of modeled time. A gridpoint in the atmosphere is defined by the tuple (atmospheric pressure, latitude, and longitude) where atmospheric pressure roughly corresponds to an altitude and a gridpoint is roughly 5.625 degrees in the latitude and longitude directions.

```

CREATE RULE C:1 ON Data_Ev, Request_Ev
IF
  SELECT Data_Ev
  FROM Data_Ev as d, Request_Ev as r
  WHERE
    ((d.lat_min >= r.lat_min or d.lat_min <= r.lat_max) and
     (d.lon_min >= r.lon_min or d.lon_min <= r.lon_max)) or
    ((d.lat_max >= r.lat_min or d.lat_max <= r.lat_max) and
     (d.lon_max >= r.lon_min or d.lon_max <= r.lon_max)) or
    ((d.lat_min >= r.lat_min or d.lat_min <= r.lat_max) and
     (d.lon_max >= r.lon_min or d.lon_max <= r.lon_max)) or
    ((d.lat_max >= r.lat_min or d.lat_max <= r.lat_max) and
     (d.lon_min >= r.lon_min or d.lon_min <= r.lon_max)) and

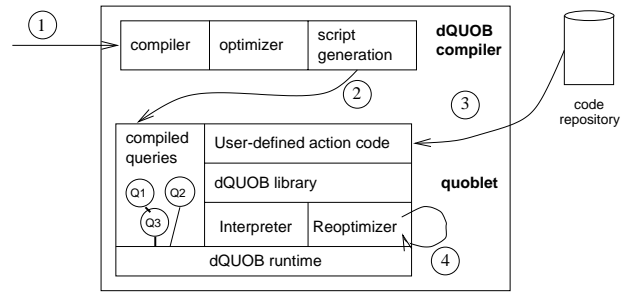
    d.level_min >= 30 and
    d.timestep % 12 = 0
THEN
  FUNC ppm2ppb

```

Figure 1. One record per ‘day’ for region of upper stratosphere defined by bounding box.

Scientists wish to view precisely the data most needed for their current investigations. For instance, since ozone changes are slow, a scientist may not be interested in data for each simulated timestep; one image per day may be perfectly adequate. Further, she may be investigating the transfer of ozone from the tropics to the south pole, so need only visualize the upper stratosphere of the southern hemisphere and the region below the antarctic circle. Figure 1 shows a suitable query specification.

Rule ‘C:1’ accepts input event types `Data_Ev` and `Request_Ev`. The `IF` clause delineates the query; `THEN` delineates the action. The `SELECT` statement defines the resulting event type, which can be either a new or an existing relation; in this case the latter. The `FROM` statement defines aliases used in the query body. The query condition is nested inside the `WHERE`. The user desires data from upper levels (above level 30) for the region defined by the bounding box generated automatically by an active user interface in response to user actions [8]. `d.timestep % 12 = 0` causes all but one event per day to be discarded. The `THEN` clause specifies that the function `ppm2ppb` be executed when the query evaluates to true. `Ppm2ppb` converts parts-per-million to parts-per-billion. Not shown is the function code itself which is written using a procedural language such as C or C++.



- ① SQL query and action defined by scientist
- ② query code moved into quoblet
- ③ action code dynamically linked into quoblet
- ④ reoptimization of compiled queries at runtime

Figure 2. dQUOB system and life of an E-A rule.

Several observations can be made about the example query in Figure 1. First, scientists specify data by referencing well-defined and meaningful data attributes like longitude and latitude. The data must obviously be structured to expose the attributes. Second, the query checks for the intersection, but currently relies on the action computation to create a new event composed only of the intersected gridpoints. Third, as can be seen, query specification can be tedious and error prone, making automatically generating queries from user actions a profitable research direction. Finally, not shown is language support for temporal operators (*e.g.*, “meets”, “precedes”) and for time related policies (*e.g.*, “a decrease in ozone at lower pressures at the equator should be followed by an increase at upper pressures within three months.”)

### 3 dQUOB System Overview

The dQUOB system is a tool for creating queries with associated computation, and dynamically embedding these query/action rules into a data stream. The software architecture, shown in Figure 2 consists of a dQUOB compiler and run-time environment. The dQUOB compiler accepts a variant of SQL as the query specification language (Step 1), compiles the query into an intermediate form, optimizes the query, then generates a script version of the code that is moved into the quoblet (Step 2).

A quoblet consists of an interpreter to decode the script, and the dQUOB library to dynamically create efficient representations of embedded queries at run-time. The resulting user E-A rules are stored as com-

piled queries. The script contains sufficient information for the quoblet to retrieve and dynamically link-in the action code (Step 3). During run-time, a reoptimizer gathers statistical information at runtime, periodically triggering reoptimization (Step 4). The dispatcher executes and manages rules.

### 3.1 dQUOB Compiler

The dQUOB compiler accepts an event-action rule consisting of an SQL-style query and associated application-specific action (*e.g.*, converting a 3D grid slice from parts-per-million to parts-per-billion.) The compiler converts the declarative-language based query into a query tree, during which time a procedural order is imposed on the non-procedural SQL specification. Query trees undergo optimization; optimized trees are used to generate code. Resulting object code is in the form of a Tcl script.

**Declarative Query Language** As is common in database query languages, our query language is declarative. The obvious strength of a declarative query language is that it shifts the burden of specifying the order of evaluation from the scientist to the compiler. Scientists specify the “what” but not the “how”. The dQUOB compiler selects the execution order. The power of declarative languages cannot be underestimated in high performance data streaming. Efficient query evaluation in any setting depends upon knowledge of the underlying representation; scientists, not knowing or caring to know that representation, cannot be assumed to specify it optimally. More importantly, because queries are executed over streaming data, certain query optimization decisions must be deferred to runtime. It is clear that a strategy for subsequent reoptimizations cannot involve continuous user involvement.

**Query Optimization for Partial Evaluation.** A key contribution to attaining high performance is efficient *partial query evaluation*. Partial query evaluation is the ability to decide the outcome of a query without evaluating the entire query; its semantics are not unlike partial evaluation of C language conditions. That is, when evaluating the condition of an IF statement, the falsehood of the condition can be determined from the failure of the first expression to evaluate to true. Partial evaluation is essential in for data streams because queries are continuously executed. An optimized query is one with the most efficient partial query evaluation. As we show in Section 5, unoptimal queries can result in increased end-to-end latency to a client.

### 3.2 dQUOB Runtime

Two key features of dQUOB’s runtime are its efficient representation of queries as compiled code and its ability to perform runtime query reoptimization.

**Queries as Compiled Code.** As stated earlier, the compiler back-end generates a Tcl script of calls to the dQUOB library. The dQUOB library, embedded in the quoblet, is a set of routines for creating compiled code for a query. That is, it is a set of routines for creating objects for operators (*e.g.*, temporal select, join), links between operators, E-A rules, and rule links. To put it another way, the dQUOB library API is akin to a set of C++ style templates. Templates, when invoked with a set of parameters, create an instantiation of the object that is customized with the parameters. Hence, the dQUOB library creates customized objects representing the query and the rule to which it belongs.

A script representation of queries has the advantage of being compact and portable. A script for a moderately complex query is roughly 10% the size of the compiled code and one can swap out a rule at run-time simply by sending a new script to a quoblet.

**Dynamic and Continuous Query Reoptimization.** Reoptimization is undertaken to generate a more optimal version of a query. Though database research has determined that an optimal version is difficult to obtain for all but a few well-defined cases, we can strive for a more optimal version where optimality is expressed in terms of total query execution time as defined in Section 4. A more optimal version will likely result if the stream data behavior has significantly changed or if a newly added query has been optimized based only on a available historical trace data. Reoptimization is accomplished through equi-depth histograms to collect statistics about data values in the stream, an optimizer to reoptimize a query from an internal representation, and a reoptimization algorithm to control statistics gathering and trigger reoptimization. A detailed description of dQUOB’s reoptimization algorithm will appear in a companion paper.

## 4 Cost Metric

A cost metric must consider the implementation of dQUOB queries and quantify the notion of ‘useful’ data transport and processing.

**Model.** Data streams are comprised of data sources, transformers, and clients, all of which are depicted in

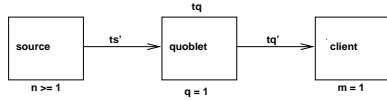


Figure 3. End-to-End Data Flow Model

Figure 3 as nodes; the arcs depict directed data flows between nodes. The depicted logical data flow obviously does not imply a physical distribution across machines. In any case, this particular data flow is modeled as emanating from multiple sources ( $n \geq 1$ ), such as coupled atmospheric transport and chemical simulations, for example, to a single client ( $m = 1$ ) such as a visualization client. The data is processed by filters before being presented to the client. The communication model,  $n \geq 1$  and  $m = 1$ , is used elsewhere for wide-area computing [3].

**Effective Real Time.** We define *Effective Real Time (ERT)* as the average end-to-end latency taken over some large number of events traveling from sources through transformer/filter to the client, where ‘large’ is dependent upon data behavior. dQUOB achieves success if embedding dQUOB queries into the data stream reduces a client’s Effective Real Time. That is, if by adding one or more queries to the data stream the client can reduce arriving data to just *useful* data without paying a penalty of unwieldy overheads, then dQUOB is a success.

The key cost metric parameters, defined in Table 1, are  $t_{s'}$ , the time to transfer a data event from a data source to one or more transformers. Since we assume an event streaming model of communication,  $t_{s'}$  also includes any instrumentation overhead (*i.e.*, event gathering, buffering, sending) at the data generator and, in the case where a data record is partitioned across multiple hosts, the total time to move the entire record to the quoblet.

$t_q$ , explained in detail below, is the time for the fastest CPU to execute the query and action over a single data event. It assumes no blocking on input.  $t_{q'}$  is the time required to transfer transformed data from a single transformer to a client. Event delivery is complete when the event arrives at the visualization client.

The model gives us a rather straightforward calculation for end-to-end latency for a single event as:

$$ERT_1 = t_{s'} * n + t_q + t_{q'}$$

reflecting possibly multiple sources, the transformation time, and transfer time to the client. As mentioned,

Parameter	Meaning
$n$	number of source hosts
$t_{s'}$	source to quoblet data transfer time
$q$	number of quoblet hosts ( $q = 1$ )
$t_q$	quoblet processing time
$t_{q'}$	quoblet to client data transfer time
$m$	number of client hosts ( $m = 1$ )

Table 1. Performance modeling parameters.

$ERT_1$  is the end-to-end time for a single event. Direct generalization to a stream of events cannot be done since  $t_q$  and  $t_{q'}$  are influenced by data stream behavior or query efficiency. Thus end-to-end latency is measured as the average over  $n$  events:

$$ERT = \sum_{i < n} (ERT_i) / n$$

**Quoblet Time.** The time spent in the quoblet (*i.e.*, transformer) is called quoblet processing time ( $t_q$ ). Processing time is dependent upon three factors: query time, action time, and some fixed overhead. Specifically, quoblet time is the sum of the query evaluation time  $t_{query}$ , action execution time  $t_{action}$ , and some fixed overhead time  $t_{overhead}$  as follows:

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

This is a worst case measure as it implies the sequential execution of quoblets, whereas dQUOB’s implementation of quoblets is multi-threaded and capable of executing in parallel on SMP machines. When multiple E-A rules are present, the cost of interaction between rules is reflected in  $t_{overhead}$ .

Improvements in quoblet processing time can be achieved by reductions in query execution time ( $t_{query}$ ), reductions in action computation time ( $t_{action}$ ), or changes in query probability ( $P(query)$ ), the probability that the query evaluation will result in a ‘true’ outcome. Reductions to action computation time is outside the scope of our work. Query probability is discussed in more detail in the full version of this paper [13].

## 5 Experimental Evaluation

The following experiments demonstrate the performance benefits of dQUOB. Specifically, that:

- dQUOB is lightweight;
- query reoptimization can bring about significant performance gains; and

- dQUOB is effective in reducing overall end-to-end latency of a data stream.

**Experiment Parameters.** Experiments are based on the data flow from Section 2 and communication model from Figure 3. Particularly, a source generates successive 3D slices of global atmospheric data for a visualization client. The quoblet physically resides on a separate workstation. The experiments are run on a cluster of single processor Sun Ultra 30 247MHz workstations running Solaris 7 and connected via 100 Mbps switched FastEthernet. Data streams are implemented using ECho [4], a publish-subscribe event-based middleware. All event data, queries, and actions in our work are realistic and obtained from experience and involvement with atmospheric scientists.

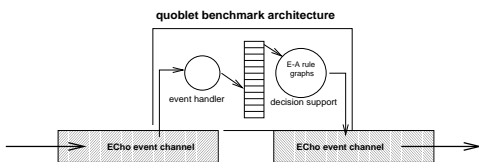


Figure 4. dQUOB architecture; benchmark case

**Microbenchmark.** To test the first claim that dQUOB is lightweight, we benchmark the execution time of a quoblet having no event-action (E-A) rules. The software architecture of the benchmark quoblet, shown in Figure 4, consists of an event handling thread, a queue, and a decision support thread.

The results shown in Figure 5 measures the minimum time a quoblet requires to process an event. That is, the time required to execute the event handler upon handler invocation by ECho, the time to copy the event to the queue, dequeue time by the decision code, then invoking ECho to send the event. Note that the measurements reflect no concurrency between event handler and decision support.

The *non-optimal copy* numbers of Figure 5 show how overwhelmingly copy cost dominates total execution time. This is evidenced by the large increases as event size grows. Partly in response to these numbers, our group is working on a version of ECho that removes the restriction that events needing retention must be fully copied out of ECho buffers. The *optimal copy* numbers are thus theoretical, and reflect our design decision to copy attribute information to quoblet space but not the actual 3D data. Thus 612 bytes are copied for each event size. In the absence of the large copy overhead, total quoblet time can be seen to be a small fraction

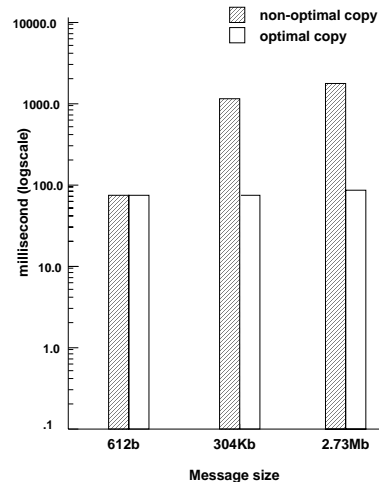


Figure 5. Benchmark quoblet time

of the data copy cost; .004 percent of the cost to copy a 2.73 Mbyte event for instance. Further, as shown in Table 2, at larger event sizes a quoblet's sustained event generation rate is in the Gbps range<sup>2</sup>.

<i>event size</i>	<i>quoblet overhead as percentage of copy</i>	<i>sustained event generation rate</i>
612 bytes	.91	20Mbps
304K	.04	10Gpbs
2.73 M	.004	90Gpbs

Table 2. Quoblet overhead explained in terms of copy cost and event generation rate.

**Justification of Query Reoptimization.** The second measurements determine whether or not in an application-realistic setting, the percentage of time spent executing a query is significant enough to justify the cost of inter-rule reoptimization (*i.e.*, reordering the operations that make up a single query).

For this experiment, we use a quoblet having one E-A rule. The query merely performs a few selects so is representative of simpler queries. The action converts a 3D grid slice from parts per million (ppm) to parts per billion (ppb); a representative algorithm of a class of operations having mid-range computational needs. To minimize the interference of the copy cost, we assume an optimized copy.

Figure 6 shows a breakout of quoblet time by query

<sup>2</sup>This sustained event generation rate assumes a quoblet does not block on socket select operations (waiting for event arrival).

and action for the three event sizes. As can be seen by looking at the ‘unoptimized query’ numbers, a query can consume a substantial amount of total quoblet time, particularly for mid-sized events (304K). The ‘optimized query’ numbers show the kind of gains that can be expected from optimizing a query of moderate complexity. Earlier studies have shown gains of up to an order of magnitude [12]. We conclude from this experiment and other observations that particularly at larger event sizes, query optimization should be undertaken because (1) query computation time is a non-trivial ‘slice of the pie’ and (2) reoptimization can successfully ‘shrink the pie’.

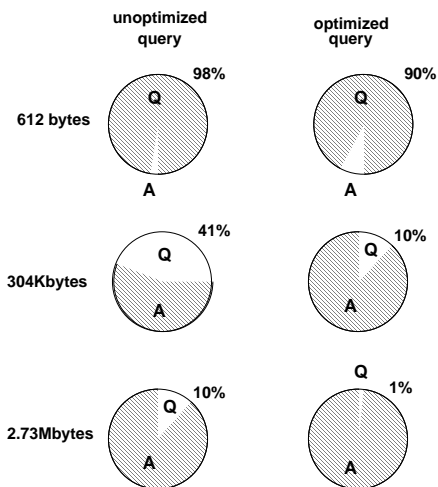


Figure 6. Breakdown of quoblet time into query (Q) and action (A) for application realistic rule.

**Reducing end-to-end latency.** Our final experiment substantiates the claim that the dQUOB system is effective in reducing the end-to-end latency of a data stream. These measurements expand on our group’s previous work which establishes the utility of conditional filtering [8] by showing the measured impact of query probability on end-to-end latency. Intuitively, a *strong rule* contains a query with a high probability of discarding events where a *weak rule* is the opposite: it passes on the majority of events received. Query probability is defined in the full version of the paper [13].

Our measurement compares queries at both extremes by comparing a weak query that discards no events,  $P(\text{query}) = 1.0$  to a strong one that discards most,  $P(\text{query}) = 0.01$ . That  $P(\text{query}) = 0.0$  was not used is the obvious case that a user requiring no data is uninteresting.

The results, as expected, favorably support our initial hypotheses by showing an end-to-end latency reduction of by as much as 99% when a weak query is replaced with a strong one. More substantial results require longer model runs with a data stream that varies its behavior over time. This is an ongoing effort.

<i>event size</i>	$P(q)=1.0$ (ms)	$P(q)=0.01$ (ms)	<i>reduction</i> (pct)ERT
612 bytes	113.37	62.19	45%
304K	922.44	63.99	93%
2.73M	11627.08	65.56	99%

Table 3. Extremes of effect of condition strength on ERT.

## 6 Related Research

Run-time detection in data streams to satisfy client data needs, the focus of our work, has been addressed with fuzzy logic [14] and rule-based control [2]. 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. ACDS [8] focuses on dynamically splitting and merging stream components; these are relatively heavyweight optimizations that might be added to our work. The Active Data Repository [5] is similar to our work in that it evaluates SQL-like queries to satisfy client needs. However, queries are evaluated over a database. dQUOB has the freedom to embed queries anywhere in a data stream so is better able to manage streams from input sources of diverse origins. Finally, the Continual Queries system [10] is optimized to return the difference between current query results and prior results. It then returns to the client the delta ( $\delta$ ) of the two queries. This approach complements our work, which is optimized to return full results of a query in a highly efficient manner.

HDF5 allows a user to selectively extract data from HDF files. Selective extraction can be thought of as a database ‘views’; HDF5 allows computation on the ‘view’. Our work complements HDF5. Franke’s [7] model of data flow gives the client explicit control over the data generator and the intermediate transformation is applied to every data event. The approach is directed at a single visualization client and single event type.

## 7 Conclusions and Future Work

In this paper we have introduced the dQUOB system as an approach to managing large data streams.<sup>2</sup> The idea behind dQUOB is that by embedding small queries with associated computation into a data stream, one can reduce the data flow to a client to only the data that is useful to the client. By providing a data model for specifying queries, a user can express precise data needs and resource constraints in a single request that crosses domain boundaries, making it particularly well suited for grid-based computing. A query is specified declaratively, which removes the burden of implementing requests from users to the dQUOB system, and also enables the latter to optimize such requests.

Future work targets wide-area computing with scaled-up numbers of users and data sources with a scheme that applies optimizations across queries, and leverages the data model to map a query ‘pushed-back’ (*i.e.* pushed upstream) by a client to a data stream architecture that is efficient and can adjust to sources under control of other administrative domains. Ongoing work looks at moving additional computation capability into the query where it can be subject to automatic optimization. We are also exploring controlling level of service to a client using the end-to-end latency metric (ERT) and query probability defined in this paper.

## References

- [1] Earthquake ground motion modeling on parallel computers. In *Proceedings Supercomputing '96*, November 1996.
- [2] 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.
- [3] Peter A. Dinda, Bruce Lowekamp, Loukas F. Kallivokas, and David R. O’Hallaron. The case for prediction-based best-effort real-time systems. In *Proceedings of Workshop on Parallel and Distributed Real Time Systems (WPDRTS)*, April 1999.
- [4] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. Event services for high performance computing. 2000.
- [5] 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.
- [6] Ian Foster and eds. Carl Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1999.
- [7] Ernest Franke and Michael Magee. Reducing data distribution bottlenecks by employing data visualization filters. In *Proc. of High Performance Distributed Computing (HPDC8)*, 1999.
- [8] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.
- [9] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.
- [10] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. Technical Report TR95-17, Department of Computer Science, University of Alberta, 1996.
- [11] 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.
- [12] Beth Plale and Karsten Schwan. Run-time detection in parallel and distributed systems: Application to safety-critical systems. In *Proceedings of Int’l Conference on Distributed Computing Systems (ICDCS’99)*, pages 163–170, June 1999.
- [13] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. Technical Report GIT-CC-00-07, Georgia Institute of Technology, Atlanta, Georgia, August 2000.
- [14] Randy Ribler, Jeffrey Vetter, Huseyin Simitci, and Daniel Reed. Autopilot: Adaptive control of distributed applications. *Proceedings of High Performance Distributed Computing*, August 1999.

---

<sup>2</sup>Prior experience with dQUOB are to safety-critical applications.