Full version of article appearing in Proceedings of High Performance Distributed Computing (HPDC2000)

dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries

Beth Plale and Karsten Schwan

GIT-CC-00-07

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
{beth,schwan}@cc.gatech.edu

Abstract

The dQUOB system satisfies clients in need of specific information from high-volume data streams. The data streams we speak of are the flows of data that exist in large-scale visualizations, video streaming to a large number of distributed users, and high volume business transactions. dQUOB introduces the idea of conceptualizing a data stream as a set of relational database tables. Within this model, a scientist can request information in an SQL-like query. Transformation or computation that often needs to be performed on the data before it arrives at a client can be conceptualized as computation performed on consecutive views *of the data; computation is associated with each view. Additionally, the dQUOB system moves the query code into the data stream as a* quoblet*; an efficient 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 to the experiments and 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[18], 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 [26, 29, 3]. 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 [22].

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 bene-

fits 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 multimonth 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 [1]. 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 [4]. One advantage of using an abstract model is the ability of its implementation to span administrative domains, a key requirement in wide-area computing [13]. Furthermore, data models like the relational and the XML Query data model [8] 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 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 [16]. 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 [27], 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 Sec-

[1] 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) [32].

tion 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 by the data streams created by the visualizations of large-scale data produced by engineering or scientific simulations[26, 1]. The data stream used in this paper concerns the visualization of 3D atmospheric data generated by a parallel and distributed global atmospheric model [17], simulating the flow of a chemical species, specifically ozone, through the stratosphere. Species transport is dependent upon horizontal winds, vertical winds, and temperature. Transport is coupled to a chemical model that models the interaction of the ozone with short lived species (*e.g.*, $CH4$, CO, HNO_3) at each logical timestep. A logical timestep is 2 hrs. of modeled time. A gridpoint in the atmosphere is defined by the tuple (atmospheric pressure, latitude, and longitude). 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.latitude_min <= -27.7 and
          d.latitude_max <= -63.68 and
          d.longitude_min = 0.0 and
          d.longitude_max = 360.0 and
          d.level_min >= 30 and
          d.level_max <= 36) and

      (d.latitude_min >= r.latitude_min and
       d.latitude_max <= r.latitude_max and
       d.longitude_min = r.longitude_min and
       d.longitude_max = r.longitude_max and
       d.level_min = r.level_min and
       d.level_max = r.level_max) and

      d.timestep % 12 = 0
THEN
   FUNC ppm2ppb
```

Figure 1. Rule C:1: query for one record per simulated day for southern hemisphere and south pole. Action is to transform requested data from PPM to PPB before sending.

For large data, scientists wish to view precisely the data they most need for their current investigations. For instance, since ozone changes are slow, a scientist may not be interested in data for each 2 hours of simulated time; one 3D slice per day is perfectly adequate. Further, she may be investigating the transfer of ozone from the tropics to the south pole, so need only receive the upper stratosphere in the southern hemisphere and the south pole. A sample rule to satisfy the request is given in Figure 1.

The query demonstrates how scientists focus on the data most useful to them, by reference to well-defined and meaningful data attributes like longitude and latitude. In this example, filtering is performed using both (1) an explicitly defined region of interest (i.e., the arctic circle), and (2) a bounding box generated by an active user interface in response to user actions [16].

The rule, named 'C:1', accepts two input event types: Data_Ev, Request_Ev. The `IF` clause delineates the query portion of the rule; the action portion is delineated by the `THEN` clause. The `SELECT` statement within the query defines the resulting event type. The result can be either a new or an existing relation. In this rule, it is the latter. The `FROM` statement establishes aliases appearing in the query body. Nested inside the `WHERE` is the query itself. The user desires data from upper levels (levels 30 through 36) of the southern hemisphere, starting at the equator and ending at the antarctic circle. She further desires data from the south pole, where the exact coordinates are defined by the bounding box event. The final line of the query limits records to one per day by discarding others. The `THEN` clause specifies that a single function, converting parts-per-million to parts-per-billion, is to be executed when the query evaluates to true. Not shown in the figure is the ppm2ppb function itself which is written using a procedural language such as C or C++.

Not evident from the example is support for additional logical connectives (*e.g.*,"or", "not"), for temporal operators (*e.g.,* "meets", "precedes"), and for time related policies. The latter, based on Chomicki [7] allow queries of the nature (*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 of a variant of SQL as the query specification language (step 1), compiles the query into an intermediate form, optimizes the query,

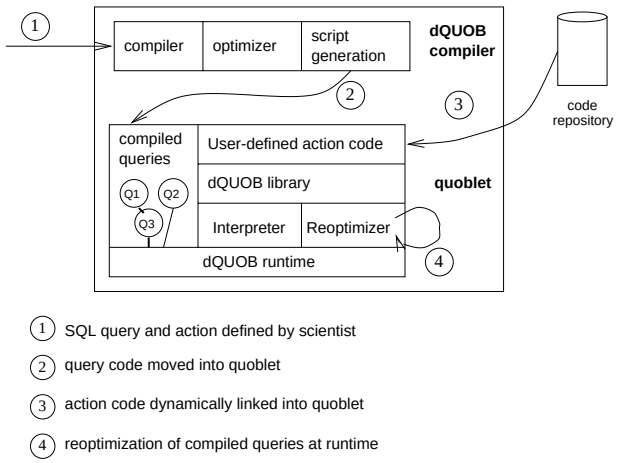

Figure 2. Architecture of dQUOB System showing the life of a query action rule.

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 runtime. The resulting user E-A rules are stored as compiled 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 event-action rules of the kind used in active databases [31]. The rules consist 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, a process which imparts a procedural order to the non-procedural SQL specification. Queries are optimized prior to code generation, the latter which parses the query tree to generate a Tcl [23] 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. In high performance data streaming, the power of declarative languages cannot be underestimated. 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. dQUOB, on the other hand, has knowledge of the representation. More importantly, because queries are executed over streaming data, some query optimization decisions must be deferred to runtime. It is clear that subsequent reoptimization cannot rely on continuous user involvement. For instance, it is desirable to push certain select operations lower in the query tree depending on the probability that the select operator evaluates to true. The select operation `d.timestep % 12 = 0` evaluates to false eleven out of twelve times, so it is a candidate for relocating lower in the tree. However, the probability cannot be determined without access to the data.

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 a streaming data flow environment because streaming data changes continuously so queries are continuously evaluated. As we show in Section 5, inefficient query evaluation techniques can result in poor end-to-end latency to the client. In a sense, partial query evaluation can be likened to code motion in optimizing compilers where the intent is to apply computation only and exactly to the data to which it must be applied.

Meta language constructs for rule manipulation. The meta-language constructs offered by dQUOB's query language resemble those of active database query languages, particularly Starburst [31]. The constructs exist for rule manipulation (*i.e.*, adding, replacing, activating, or deactivating a rule). The E-A rule construct binds an event (*i.e.*, an SQL-like query) to an action. Rule manipulation allows rules to be accessible to clients, and responsive to changes in the data stream.

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 a compiled code of 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. Though total quoblet-level portability has not been addressed (a quoblet runtime and library must exist for a host architecture), rule portability is achieved. One can swap out a rule at run-time simply by sending a new script to a quoblet.

Dynamic and Continuous Query Reoptimization. Dynamic query reoptimization is a strength of the dQUOB system. Its need can be illustrated by simple example. Key database optimization techniques rely on information gleaned from the data itself: the MIN and MAX values of an attribute or its location (*e.g.,* cached or on disk), for example. In dQUOB's streaming environment, however, the best data values available are historical trace data so it is highly likely that additional optimizations beyond the initial ordering will be required to more closely approximate a optimal query.

The underlying data structure (*i.e.* no tables) introduces other challenges that are compounded by the nature of the queries written over streamed data. For instance, a simple statistic, such as a running average, is often not accurate because it is based on the assumption of normal distribution of data, an assumption that does not hold for the type applications we work with. The reason is clear; aberrant behavior by an application is often characterized by values out of range. The sporadic nature of error conditions alone violates the assumption of normal distribution.

Detection of changes in data stream behavior are accomplished by a statistical sampling algorithm that gathers statistical information about data behavior into an equi-depth histogram [21, 25] through periodically sampling the data stream. A detailed description of dQUOB's reoptimization algorithm will appear in a companion paper.

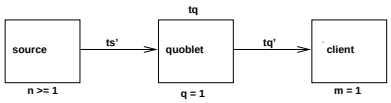

Figure 3. End-to-End Data Flow Model

4 Cost Metric

To formulate a meaningful cost metric, we must both consider the implementation of dQUOB queries and capture the notion of 'useful' data transport and processing more abstractly.

Query Implementation. Data streams in general are comprised of data sources, transformers, and clients, all of which are depicted in Figure 3 as nodes; the arcs depict directed data flows between nodes. The logical data flow depicted in this figure obviously does not imply a physical distribution across machines. In any case, this particular data flow is modeled as emanating from multiple sources ($n >= 1$), 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 [9, 6].

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, dQUOB is a success 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.

The key cost metric parameters, defined in Table 1, are $t_{s\prime}$, 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, $t_{s\prime}$ 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. $t_{q\prime}$ 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.

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

Table 1. Performance modeling parameters.

The model gives us a rather straightforward calculation for end-to-end *Real Time* or RT as:

$$\mathsf{RT} = t_{s\prime} * n + t_q + t_{q\prime}$$

reflecting possibly multiple sources, the query and transformation time, and event transfer to the client. As mentioned, RT is the end-to-end time for a single event. Direct generalization to a stream of events cannot be done, primarily because changes occuring in data behavior or in client queries can affect t_q and $t_{q\prime}$. Thus end-to-end latency is measured as the average Real Time over n events:

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

To quantify the impact of variable transformer time, we next decompose transformer time (i.e., t_q).

Quoblet Time. The time expended by a quoblet to process a single event in the data stream is called quoblet processing time. Processing time is dependent upon three factors: query time, action time, and some fixed overhead. Specifically, quoblet time is defined as 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 reducing the query execution time (t_{query}) or action computation time (t_{action}), or by changing $P(query)$, the probability that the query evaluation will result in a 'true' outcome. Optimizing user-defined computation is outside the scope of our work. In the prototype, the client passes a function name to the quoblet at the time an E-A rule is created that we assume has been optimized for the host machine. The function can be dynamically linked into the quoblet executable. Our efforts instead are focused on query optimization and query probability. The former has been discussed earlier while the latter is discussed next.

Query Probability. Query probability, assigned to an E-A rule, gives an indication of the query's strength. Specifically, it is a number from 0.0 to 1.0 that indicates the probability that a query when executed, will yield a 'true' outcome. A 'true' outcome results in the corresponding action being triggered. A query probability of 1.0 suggests that the query eliminates or filters all received events, while a probability of 0.0 eliminates none. Query probability has a significant impact on quoblet time because, as we show in Section 5, action computation consumes a larger portion of quoblet time, so a strong query can be very effective in reducing effective real time to a user (ERT).

Query probability can be approximated by an *indirect measure*:

$$P(query) = 1.0 - t_{q\prime}/t_{s\prime} \quad (1)$$

To clarify, in the case of 'null' actions, query probability is the inverse of the ratio of quoblet output bandwidth to input bandwidth. For example, if input bandwidth is 5 GB/sec and output bandwidth .1 GB/sec, then $P(query)$ is 0.98. This indirect measure is only approximate because it reflects hidden costs such as communication overhead and latencies between sender and receiver. A prediction of query probability would be desirable.

Limited query probability *prediction* can be undertaken, leveraging off established work on database selectivities [32]. A *selectivity* is the probability that a select operation will return a boolean value of 'true' for its expression. In ongoing work we are expanding the types of queries for which query probability can be predicted to include more complex queries. For queries consisting of project and most types of select operations, query probability is simply the product of the selectivities as shown in equation (1) where $selectivity_j$ is the selectivity of the j^{th} operation and m is the total number of operations in a query. The selectivity of selects is assumed to be determined through historical trace data collected from prior runs.

$$P(query) = \prod_{j<m} selectivity_j \qquad (2)$$

Section Summary. The cost metric, ERT, developed in this section provides a mechanism for quantifying the benefits of embedding dQUOB quoblets into a data stream. It also highlights two points where performance improvements can occur: either by optimizing a query so that it performs partial evaluation earlier, or by strengthening the query to increase the probability that a query will discard an event. The next section provides results of measurements that focussed on query optimization, query probability, end-to-end latency ERT, and benchmark performance of an embedded quoblet.

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.

The results in this section are further supported by previous work done by our group that has shown the utility of conditional filtering in distributed computing environments [5, 16],

Experiment Parameters. Experiments use the the data flow described in Section 2 and the communication model of Figure 3. That is, a source generates successive 3D slices of global atmospheric data for a visualization client. A quoblet is embedded into the data stream between source and sink and 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 as event channels using ECho [10], a publish-subscribe event-based middleware and using binary I/O (PBIO [11]) for fast heterogeneous transport. Data events are of three sizes (612 bytes, 304Kbytes, and 2.73Mbytes); the size of the event is dependent upon the number of chemical species contained therein. Specifically, a 612byte event contains no species, a 304K event contains the transported species, ozone, and the 2.73Mbyte events contains all nine modeled chemical species.

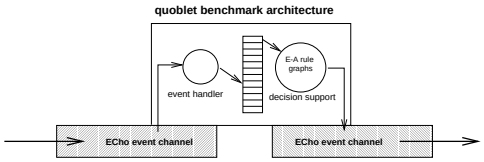

Figure 4. dQUOB architecture; benchmark case

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 are shown in Figure 5 and Table 2. 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 terminating when the decision code invokes ECho to send the event. Note that the measurements reflect no concurrency.

The *non-optimal copy* numbers of Figure 5 show how overwhelmingly the 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 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[2].

event size	*quoblet overhead as percentage of copy*	*sustained event generation rate*
612 bytes	.91	$20Mbps$
304K	.04	$10Gpbs$
2.73 M	.004	$90Gbps$

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

[2]This sustained event generation rate assumes a quoblet does not block on socket select operations(waiting for event arrival).

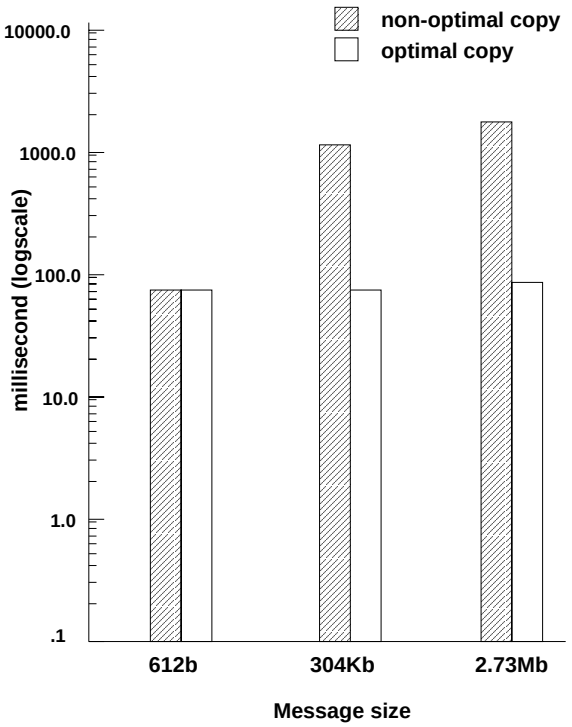

Figure 5. Benchmark quoblet time

Justification of Query Reoptimization. The second measurements are undertaken to 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. That is, reordering the operations that make up a single query based on information acquired about the data at runtime that a more optimal ordering possible.

For this experiment, we use a quoblet containing a single E-A rule. The query merely performs a few selects so it is representative of simple 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 that apply a floating point operation to each gridpoint. We categorize the action as one of 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 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 what kinds of gains can be expected from the optimization of a query of moderate complexity. Earlier studies have shown this to be true by up to an order of magnitude [27]. 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 non-trivial with respect to total quoblet time and (2) reoptimization can yield significant reductions in overall quoblet processing.

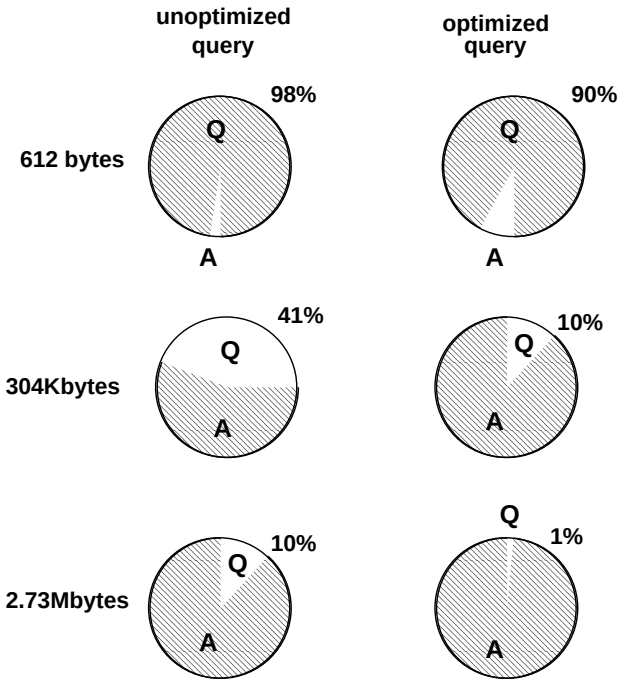

Figure 6. Breakdown of quoblet time into query (Q) and action (A) for application realistic rule.

Reducing end-to-end latency. Our final experiment is to substantiate 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 [5, 16] by showing the measured impact of query probability on end-to-end latency. As described earlier, 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.

Our measurement compares queries at both extremes. Specifically, it compares a weak query that filters no events, P(query) = 1.0, and a strong one that filters all events but one, P(query) = 0.01. That the case of 'all events filtered' was not used in latter case is an anomaly of the test infrastructure that requires a single event to terminate measurement at the client.

The probability of a query is computed equation (2) appearing in Section 4.

The results 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. Deeper understanding of condition strength requires additional testing with longer model runs, in order to obtain a data stream that varies its behavior over time. This is an ongoing effort.

event size	*P(q)=1.0* (ms)	*P(q)=0.01* (ms)	*reduction* (*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 event-based environments has a long history. Pablo [28], Paradyn [20], SCIRun [24], and Falcon [15] established run-time performance evaluation of parallel and distributed systems. Run-time detection to satisfy client needs for data, the focus of our work, has used fuzzy logic [29] and rule-based control [2]. We argue that, because of the more static nature of these approaches, the dQUOB system is more adaptive to changes in data behavior, thus achieving better performance when stream behavior is variable. ACDS [16] focuses on dynamically splitting and merging stream components; these are relatively heavyweight optimizations that might be added to our work. Research on the Active Data Repository [12] is similar to our work in its evaluation of SQL-like queries to satisfy client needs. However, in this work, queries are evaluated over a database before the resulting information items are streamed to clients. In comparison, dQUOB can embed queries anywhere within a data stream, thereby able to manage streams consisting of multiple input sources and at a location wherever appropriate in the distributed system source and client stream elements. Finally, the Continual Queries system [19] is optimized to return the difference between current query results and prior results. It then returns to the client the delta (δ) 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 provides extractor capability by allowing a user to selectively extract data from HDF files. An extractor can be thought of as supporting the database notion of 'views' and allows computation on a 'view'. Our work complements extractor capabilities; in fact we are exploring joint efforts. Franke presents a model of transformers [14] where a consumer explicitly controls the data generator and does a remote shell command to start up the generator. The transformer handles data events uniformly; that is, it performs the same transformation to every data event. The work is directed at satisfying the needs of a single visualization client.

7 Conclusions and Future Work

In this paper we have introduced the dQUOB system as an approach to managing large data streams.[2] 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.

To establish the utility of the dQUOB system, microbenchmarks are presented for dQUOB's runtime infrastructure. Quoblets are found it to be sufficiently efficient for the large data streams of interest to high performance applications. We also substantiate the need for query optimization, by measuring an unoptimized query and an optimized one using real application 'action' computations. The need for these measurements is clear: if a 'typical' user computation overwhelms total quoblet time, then it makes little sense to devote resources to optimizing the query. Our results substantiate the need for optimization. Finally, we demonstrate the significance of query 'strength' on data stream efficiency (measured in terms of end-to-end latency.) As expected, the benefits are strong. Having the ability to predict the probability of a new E-A rule will let the dQUOB system be more responsive to changes it detects in the resource base of the overall data stream. This final item is one topic of ongoing work.

[2] Prior experience with dQUOB was in application to safety-critical systems [30].

8 Acknowledgments

The first author would like to express thanks to Patrick Widener of Georgia Tech for his help in measuring dQUOB's performance and to Richard Snodgrass for advice in formulating dQUOB's query language. '

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] Henri E. Bal, Aske Plaat, Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, and Ronald Veldema. Parallel computing on wide-area clusters: the Albatross Project. In *Proceedings of Extreme Linux Workshop*, pages 20–24, 1999.

[4] Tim Bray, Jean Paoli, and eds. C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, World Wide Web Consortium, 1998.

[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] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical Report CS-95-313, University of Tennessee, 1995.

[7] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2), June 1995.

[8] W3C consortium. XML Query Requirements. http://www.w3.org, 2000.

[9] 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.

[10] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. Event services for high performance computing. 2000.

[11] Greg Eisenhauer and Lynn K. Daley. Fast heterogeneous binary data interchange. In *Proceedings of Ninth Heterogeneous Computing Workshop (HCW2000)*, 2000.

[12] 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.

[13] Ian Foster and eds. Carl Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1999.

[14] Ernest Franke and Michael Magee. Reducing data distribution bottlenecks by employing data visualization filters. In *Proc. of High Performance Distributed Computing (HPDC8)*, 1999.

[15] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[16] 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.

[17] 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.

[18] Argonne National Laboratories. Access grid. http://www-fp.mcs.anl.gov/fl/accessgrid, 2000.

[19] 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.

[20] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. Paradyn parallel performance measurement tools. *IEEE Computer*, 28, November 1995.

[21] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings ACM SIGMOD Conference*, pages 28–36, June 1988.

[22] National Information Library (NIL). Invited talk on digital libraries. Supercomputing (SC98), 1998.

[23] John Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1995.

[24] S.G. Parker and C.R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proc. Supercomputing 95*, pages 1–1, 1995.

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

[26] 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.

[27] 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.

[28] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment.* Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, November 1992.

[29] Randy Ribler, Jeffrey Vetter, Huseyin Simitci, and Daniel Reed. Autopilot: Adaptive control of distributed applications. *Proceedings of High Performance Distributed Computing*, August 1999.

[30] Beth (Plale) Schroeder, Sudhir Aggarwal, and Karsten Schwan. Software approach to hazard detection using on-line analysis of safety constraints. In *Proceedings 16th Symposium on Reliable and Distributed Systems SRDS97*, pages 80–87. IEEE Computer Society, October 1997.

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

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