IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2001

Optimizations Enabled by a Relational Data Model View to Querying Data Streams

Beth Plale and Karsten Schwan

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

Abstract

We postulate that the popularity and efficiency of SQL for querying relational databases makes the language a viable solution to retrieving data from data streams. In response, we have developed a system, dQUOB, that uses SQL queries to extract data from streaming data in real time. The high performance needs of applications such as scientific visualization motivates our search for optimizations to improve query evaluation efficiency. The purpose of this paper is to discuss the unique optimizations we have realized by a database point of view to streaming data and to show that the enhanced conceptual model of viewing data streams as relations has reasonable overhead.

1 Introduction

Passage of time and widespread adoption have made the benefits of queries as a means of retrieving data from databases widely known to computer science specialists and non-specialists alike. The wide popularity of SQL as the standard query language for relational database management systems attests to that particular language's ability to satisfy a user's need for data of interest. That is, its widespread use for a wide range of applications is informal testament to its expressiveness.

From the recent explosion of the Internet and the ubiquity of computers has emerged a new data source, however, the computational data stream. Data streams are generally regarded as event data in transit from some source to some consumer. And the streams are prevalent: the Aware Home Project at Georgia Tech has data continuously flowing from the myriad of sensors in the house to large compute engines on campus. Scientific investigation using high end graphics machines to visualize results requires large volumes of complex scientific data be transported. Delta airlines pushes in excess of 12 million events per day between the ticket counter, check-in counter, passenger check-in, airport update monitors in concourses, reservations desks, and the mainframes [10].

We hold a commonly held view of data streams as streams of events, where an event is timestamped data about a component. Our group has developed the notion and established the viability of *computational data streams*, streams with computation inserted at the source, destination, or at intermediate points between. The computations serve to transform, aggregate, or filter the data. For example, aggregation might be employed to sum values over neighbor points in a 3D space to reduce downstream bandwidth needs. Transformation might perform units conversion or partially prepare the data for visualization. Computational data streams are one of the underlying mechanisms of the Infosphere project [13]. Their viability has been established in [6]. Data streams have been treated by otherse in [4], [2], and [7].

Our work with dQUOB is in adapting database queries to operate over streaming data instead of database tables. Viewing data streams as data sources over which relational queries can be specified has been explored in the past in the context of performance monitoring [15], but it suffered limitations in the ability to keep up. Our contribution is to replace all traces of a database with temporary buffers to improve performance. The work further contributes adaptivity to query processing, under the hypothesis that more efficient, optimal queries can be achieved if run-time data can be fed back into the optimization cycle.

Earlier results [12] have shown that optimized queries can significantly reduce query computation time. Further, our work with a global atmospheric transport model and earlier with a autonomous robotics application has shown that relevant and meaningful queries can be stated with the SQL query language. Whereas earlier work by our group has justified the benefits of stream computation [6], our work introduces a conceptual model for thinking about computational data streams and demonstrates the utility of coupling computation with queries to achieve greater gains in total stream processing.

The contributions of this paper are two-fold. Conceptualizing streaming data with a relational data model creates an opportunity for new optimizations on data streams. We present the optimizations we have realized from this new way of thinking. Our choice of deployment strategy and internal representation for queries maximizes query portability and adaptability. The second contribution is to quantify the overhead incurred by our general query representation.

In the following section we give a brief overview of dQUOB. The optimizations made possible by our implementation and the relational data model are the topic of Section 3; measurements appear in Section 4. We conclude with related work in Section 5 and future direction in Section 6.

2 dQUOB Overview

The dQUOB (dynamic QUery OBjects) system enables users to create queries for precisely the data they wish to use. With the queries are 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.

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 1 consists of a dQUOB query compiler and run-time environment. The compiler accepts an SQL query (Step 1), compiles the query into an intermediate form as a parse tree, performs query optimizations over the parse tree, then generates a script. The query is deployed at the quoblet by passing it a script (Step 2). A quoblet consists of an interpreter to execute the script, and the dQUOB library to dynamically create compiled code representations of the queries at runtime. The script also contains information used by the quoblet to retrieve and dynamically link the user defined action code (Step 3). During run-time, the reoptimizer gathers statistical information about the data stream, periodically triggering reoptimization (Step 4). The quoblet has three instantiated queries, Q1-Q3, and Q1 is dependent upon the output of Q3.

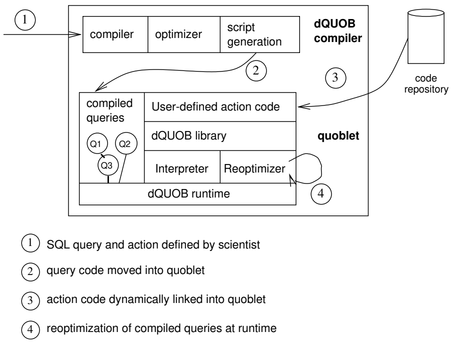

Figure 1. Life of a Query/Action Rule.

3 Optimizations

Our query-based approach to making decisions over streaming data enables optimizations that may not be possible with other approaches, or may not be done as naturally.

The SQL query language provides optimization potential largely because SQL queries are declarative, that is, the query specifies *what* data is wanted, not *how* the data is to be retrieved. The optimization potential is obvious: we can create a more efficient version of a query than the one given to us by the user. Procedural queries, on the other hand, embed an explicit order of execution. The latter type query is more common than one might think. A request for data written as a condition statement in a procedural programming language such as C, C++, or Java is obviously procedural, but so are the query languages associated with today's directory servers such as LDAP and DNS. The choice of SQL as a query language allows us to optimize the queries before enacting them and also reoptimize them on-the-fly as the environment changes.

The relational data model provides efficient query processing in large part because its simplicity has allowed for its well-defined mathematical foundation in relational algebra. And efficiency is a leading reason for the popularity of relational databases. Contrast this to most object-based query languages that, while providing a SQL-like syntax, have difficulty achieving efficient query evaluation, in part by the very features that give the object model its enhanced conceptualization; specifically complex objects and class hierarchy. *Complex objects* are objects having nested references to other objects. The resulting path expressions are a key feature in object queries, and a key difficulty in creating efficient queries. A *class hierarchy* is a set of objects related by a parent-child relationship. A query must access and return objects in some or all classes in a class hierarchy.

From our experience, events are independent in the sense that they frequently do not contain nested relationships to other events, nor are they inextricably tied to a class hierarchy. Thus the additional conceptual expressiveness of complex objects and class hierarchies is not worth the additional complexity and loss of efficiency that accompany it.

Algebraic optimizations are heuristics which directly manipulate the query tree, the latter being the internal representation of the query in the query compiler. In our work, we have considered four algebraic heuristics, all of which have been applied in traditional database query evaluation:

- push select operations down the query tree,
- push project operations down the query tree,
- factor out a subexpression common to two or more queries resident in the same quoblet into a separate query resident at the quoblet, and
- reorder select operators based on statistical metrics about the data.

Through experimental evaluation, we determined that while pushing selects down the parse tree yielded significant improvements in query evaluation time (by reducing the number of events participating in Cartesian product), pushing projects had a detrimental effect on performance. Projects traditionally serve to minimize the amount of data retrieved from disk. But events in data streams have already arrived and reside at the quoblet so there is little value to projecting away a subset of the fields. Further, since projection as implemented in dQUOB requires the creation of a new event, projects impose the burden of a copy cost.

On the basis of additional experimental evaluation, we rejected the third heuristic. Factoring out a common subexpression requires creating a new query that has a project operator. The cost of executing the additional project operator overshadowed the benefit of a reduced number of select operations.

The fourth heuristic takes advantage of the associativity property exhibited by relational operators. That is, the property ensures operators can be reordered without a compromise to correctness. Whereas the first three heuristics are derived from the relational model and declarative query language and adapted to the unique circumstances of streaming data, the fourth depends upon the query deployment strategy and run-time representation. Hence we introduce these before returning to the heuristic.

3.1 Code Deployment and Internal Representation

Code generation and deployment is accomplished through scripts, a class library, and a runtime environment. A query compiler located at a client outputs scripts consisting of calls to the dQUOB library. The dQUOB library is a set of classes capable of instantiating a query as a set of operators (for select, project, join) linked as a directed graph. Code deployment occurs by passing a script to a quoblet. The quoblet, upon receipt of the script, invokes the interpreter to execute the script, resulting in instantiation of the query as compiled code.

Alternative deployment processes exist. The query compiler could instead generate procedural language source code that is shipped to the target host machine. A host resident 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. The advantage of either alternative is it removes the need for the dQUOB library at the quoblet. However, both lack support for modifying a query on-the-fly and require additional system calls for dynamic linking. Alternatively, the compiler could generate a script of the query that is then interpreted at the quoblet. This approach is employed by earlier versions of Java JDK. A next generation on-the-fly compiler, such as Java's Hot Spot compiler, mitigates the performance disadvantages of the interpreted approach.

Our script approach to code deployment has two major strengths: first, we can conveniently deploy scripts because scripts are small and portable. A script is roughly one-tenth the size of compiled code it represents and is more portable because script languages in general are host architecture independent. Second, the combination of a script to deploy code and a directed graph for a query's internal representation simplifies the problem of efficient on-the-fly query reoptimization.

As previously mentioned, certain query optimization heuristics depend upon statistical metrics about the data. Except in the case where historical trace data is available, statistical information cannot be computed by the compiler because the data itself has not yet been generated. Thus these types of query optimizations must be deferred to runtime. The particular statistical metric we are interested in is called a selectivity. A *selectivity* is a probability assigned to a particular select operation. Intuitively, if a select tests for atmospheric level equal to 5 and there are 37 levels, then assuming the data follows a normal distribution, the probability that the select will evaluate to true is 1/37. Select operators having smaller values are pushed lower in the query tree. dQUOB computes selectivities at runtime by sampling the data streams to build depth-first histograms [9]. Details of the query reoptimization process are outside the scope of this paper.

Reoptimization requires on-the-fly reordering of the operators making up a query. A key strength of the work is dQUOB's ability to accomplish the reordering efficiently. The efficiency rests on two facts: first, each node in the graph is an independent, side-effect free function with an input queue and list of output queues to which to direct events. Second, because relational calculus operators are associative, correctness cannot be compromised by an incorrect operator ordering. Thus operator reordering can be accomplished without the overhead of guaranteeing correctness. At worst case, an ordering less optimal than its predecessor will be selected. 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 [5]. The dQUOB runtime guarantees atomicity in query update.

Deploying scripts as queries and representing them internally as directed graphs permits on-the-fly query reoptimization. Though current runtime optimizations are limited to detecting and responding to changes in selectivities of the data, we are pursuing other potential optimizations.

4 Measurements

We have shown the opportunity for optimization that exists when one views data streams as tables over which SQL queries can be stated. We have also shown the gains in portability and in on-line optimization made possible by deploying queries as scripts and representing queries as directed graphs. The purpose of this section is to quantify the cost of so general an approach. Using a moderately complex query taken from the atmospheric transport application we compare a dQUOB query against the same query implemented as a hard-coded function.

The sample SQL query, shown below, requests data for specific atmospheric levels, 30 and above, where the data meets one of two user specified latitudinal criteria and when an end-to-end performance value (*i.e.* p.latency), does not exceed a constant value (*MAX_ERT*). The directed graph consists of ten operators: seven selection, two join, and one projection.

```
CREATE RULE C:2 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.level_min >= 30 and
       r.aid == 1001 and
       p.latency == 1001 and
       p.latency <= MAX_ERT and
       d.aid = r.aid
```

The experiment compares a dQUOB SQL query to the same query implemented as a procedural language function compiled into the quoblet, the latter referred to as a 'static query'. dQUOB query cost is the cost of traversing the query graph plus overhead of storage reclamation. Since quoblets can support multiple queries simultaneously, there is overhead associated with event handling as well. Query processing costs are averaged over the total stream of events to obtain a per arriving event processing cost. Since our measure is focussed on query processing, no action (*e.g.* units conversion) is performed. The performance evaluation environment consists of a cluster of single processor Sun Ultra 30 247MHz workstations running Solaris 7 and connected via 100 Mbps switched FastEthernet. Data streams are implemented with the ECho [3], a publish-subscribe, event-based middleware library. The quoblet, provider, and consumer reside on separate workstations.

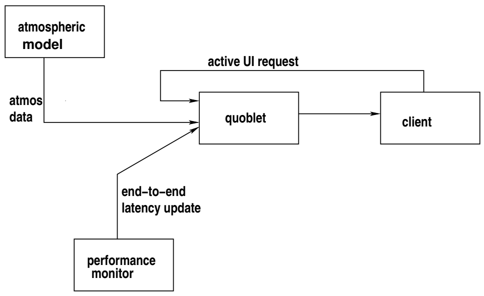

Figure 2. Data stream model for Case 1.

As shown in Figure 2, Case 1 consists of a quoblet receiving events on three event event channels and writing events to one. 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 sporadically and randomly.

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)	*Total bytes*
1	34.07	87.00	121.07	36.48M
10	32.35	87.31	119.66	36.48M
120	32.43	84.56	116.99	36.48M

Table 1. Case 1: dQUOB query cost per data model event.

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)	*Total bytes*
1	16.65	2.78	19.43	41.95M

Table 2. Case 1: Static query cost per data model event.

The dQUOB query processing cost is given in Table 1 for different sized join buffers. The results are broken out into quoblet overhead and time spent evaluating the query, that is, traversing the query graph. 'Total cost' is the sum of the overhead and query columns. 'Total bytes sent' is the total number of bytes received at the client. In comparing query cost for the various buffer sizes, we see that contrary to intuition, smaller join buffers result in increased processing time. This is counter-intuitive because, as we discussed earlier, the join operator is a Cartesian product (with a condition) over two buffers so larger buffers should always yield increased per-event processing time.

The explanation lies in the types of events that are participating. In dealing with time-dependent data, the database community has introduced the distinction between a snapshot event and the interval event.[1] The *snapshot* event is a snapshot the of behavior of an application, 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. Interval events are useful for describing persistent state of an application component. For instance, a performance monitor can issue an end-to-end latency event that dQUOB interprets as being valid until the next latency event is received.

dQUOB interprets an arriving interval event that has no stop time as being valid either until a stop event is received or the next interval event arrives. In the latter case, the end time of an interval event with no stop time is assumed to be the start time of the next received event. dQUOB uses its knowledge of interval events for more efficient query processing: by limiting the join buffer size for interval events. Independent of the size of the join buffer for a snapshot event, the Cartesian product between an interval event and a snapshot event results in at most one tuple being generated. Our example query has two join operations, both of which involve an interval event, so join cost is constant irrespective of buffer size. Thus, what we see at buffer sizes 1 and 10 is storage reclamation time. As long as an event exists in a join table, it must be kept around. Once it is no longer needed, its storage is reclaimed. At buffer size 120, the buffer is large enough to hold every data event generated, so no storage reclamation is performed.

The second table, Table 2, gives the cost per event of processing the static query. There are a couple points to note. First, for our purposes of comparison, a join buffer of size one is sufficient as we are interested in worst case. Second, the split between overhead and condition is not clean because nuances in coding make it difficult to capture task processing costs more precisely.

A comparison between Table 1 and Table 2 reveals the cost of dQUOB query processing. Particularly revealing is the 5x overhead in total cost of using dQUOB queries. These costs, while admittedly not trivial, must be balanced against three observations. First, the static query is fast in part because we coded it with an understanding of interval events gleaned from our work. It is less likely that a query hand-coded by a domain specialist would be as efficient. Second, the good results for the static query cost must be tempered by comparing the 'total bytes sent' column of both tables. This number, which reflects the total number of bytes streamed to the client, is 15% higher in the static case. The additional data is attributed to duplicate data events that are generated whenever a new interval event is received. Though duplicate removal processing could have been added to the static case, we chose instead to highlight an important issue: that issues of this type must be dealt with, if not by the query evaluator, then downstream, or worse yet, at the client. Finally, the dynamic adaptability of queries over time give long term performance gains.

[1] 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'.

Case 2 consists of two snapshot events (as opposed to one). This model could exist when a scientist wishes to compare model generated data and observational data obtained from satellite feeds. The publishing rate of the sources differ considerably, one is 1/6th the rate of the other, requiring the query to buffer the faster stream. The results, shown in Table 3, show the cost (in microseconds per event arriving from the atmospheric model) to execute the dQUOB query.

Comparing the two rows of Table 3, the additional cost at buffer size 120 can be attributed to graph traversal and additional Cartesian product processing. For the static query, Table 4, the decreased overhead at buffer size 120 can be explained by less frequent storage reclamation, as explained earlier. Increased query processing time at the larger buffer size can be attributed to the cost of processing Cartesian product pairs. Comparing Tables 3 and 4, we see that the dQUOB query cost ranges from 2.4x to 5.7x. It is worthwhile to point out that queries with high join costs still execute in the hundreds of microsecond range.

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)
3	19.33	65.89	85.22
120	37.28	147.79	185.07

Table 3. Case 2: dQUOB query cost per event.

Join buffer size	*Overhead* (μs)	*Query* (μs)	*Total* (μs)
3	25.41	10.75	36.16
120	12.94	19.47	32.41

Table 4. Case 2: Static query cost per event.

We have, in this section, quantified the cost of our relational data model approach to extracting data from data streams. The sample queries give us a good feel for the upper-end cost at roughly 6x the cost of a static implementation. Additionally, we have highlighted issues such as buffer size, duplicate suppression, and events versus intervals that are a necessary part of data stream management. Results showing the benefit of specific query optimizations appear elsewhere [11]. In general, we are seeing roughly an order of magnitude reduction in cost between an unoptimized query and its more optimal cousin. The examples used here are optimized.

5 Related Research

Run-time detection in data streams has been addressed with fuzzy logic [14] 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. ACDS [6] focuses on dynamically splitting and merging stream components; these are relatively heavyweight optimizations that might be added to our work. The Active Data Repository [4] 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 [8] 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.

6 Conclusion

dQUOB is a way conceptualize data streams and a system for extracting data from data streams at runtime. By conceptualizing a data stream as a set of relations, one can view the process of extracting data from these streams as specifying an SQL query that continually evaluates over the stream, returning data that matches the query. The dQUOB system is a prototype implementation of this conceptualization.

This paper shows that SQL, the relational data model, and our code deployment and internal representation techniques enable optimizations that other approaches to querying data streams do not have at their disposal. The measurements show that this enhanced conceptual model has reasonable overhead. In fact, much of the overhead, such as duplicate elimination and joins, must be addressed anyway, and better done automatically than by forcing the user to deal with it in an ad-hoc and likely inefficient way.

Future research. Our most pressing task is to determine reoptimization overhead both in terms of load on the quoblet and in response time. We are also addressing 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 a correct approach to interoperability.

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] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2000.

[3] G. Eisenhauer, F. Bustamente, and K. Schwan. Event services for high performance computing. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.

[4] R. Ferreira, T. Kurc, M. Beynon, C. Chang, and J. Saltz. Object-relational queries into multidimensional databases with the Active Data Repository. *Journal of Supercomputer Applications and High Performance Computing (IJSA)*, 1999.

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

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

[7] F. Kon, R. Campbell, M. Mickunas, K. Nahrstedt, and F. Ballesteros. 2k: A distributed operating system for dynamic heterogeneous environments. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.

[8] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. Technical Report TR95-17, Department of Computer Science, University of Alberta, 1996.

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

[10] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. Operational information systems - an example from the airline industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, 2000.

[11] B. Plale and K. 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.

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

[13] C. Pu, J. Walpole, K. Schwan, L. Liu, and G. Abowd. Infosphere. http://www.cc.gatech.edu/projects/infosphere/, 2000.

[14] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive control of distributed applications. *High Performance Distributed Computing*, August 1999.

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