Continuous Query Processing and Dissemination

Yan Chen, Kelvin Lwin and Sam Williams

Abstract

Traditional database information monitoring techniques, such as database triggers, active DBMS and continuous queries have various limitations, can not provide large scale dynamic data push and have no efficient data dissemination support. Meanwhile, current dissemination systems, which are separated from database, can not handle complicated query requests, such as aggregate operation. Also, as a middleware, it adds more overhead to the data querying. We designed, implemented and analyzed the first efficient active lightweight DBMS which also has full dissemination features built-in. We proposed a novel technique: record-based triggering, for efficient monitoring of clients' interested records known *a priori*. Also we use message passing for communication between clients and database servers. Unlike table-based triggering in current commercial database systems, record-based triggering gives much better scalability as well as speed. Meanwhile, compared with traditional communication mechanism as remote procedural call, message passing give full control to the active DBMS system, thus it can optimize the data processing and dissemination. The system, called Continuous Query Processing and Dissemination (CPQD), has been implemented in Java and is portable, extensible and scalable. Compared with traditional periodic pull techniques, the experiments results are very promising.

1. Introduction

With the ongoing advance of the Internet and intranets, everyone can publish information on the web independently at any time. There also may be millions of users of the network. So there is a tremendous scalability problem here. Meanwhile, it is equally daunting to navigate, collect, process, and track data in this dynamic and open information space. The problem is aggravated when source information changes constantly and unpredictably. As a result, users have to frequently poll the web sites of interest and fuse the newly updated information manually to keep track of changes of interest, which is a great pain.

For the problems above, it is desired to have a efficient Database Management System (DBMS) which can efficiently process the data and deliver the data. There are quite a few benefits by using *push* technology instead of *pull*. The clients can receive more up-to-date data with less network traffic. High-degree of overlap of user data may be utilized for shared-caching and multicast. In addition, it is desired if the repetitive nature of some aggregate query operation can be avoided to save the server computing time, which leads to better scalability of server. With all the features, we designed, implemented and analyzed the first active lightweight DBMS with full dissemination functionality, called Continuous Query Processing and Dissemination (CQPD) system.

The paper is organized as follows: the next section will give a comprehensive survey and evaluation to the relevant previous work. Section 3 gives the system overview. Section 4 details the communication through queued message passing and we study the contSQL server in details in Section 5. Section 6 discusses the performance experiments and result analysis. Conclusions are given in Section 7 and future work, Section 8.

2. Previous Work

There has been considerable research done in the monitoring of information changes and efficient re-computation in databases and web-based search systems. Powerful database techniques such as database triggers, rule-based active database, continuous queries and incremental computing have been developed.

Database Triggers: Like constraints, conceptually a database trigger is an event-condition-action (ECA) rule. Commercial DBMSs have been introducing support for database triggers at various

levels mainly for integrity constraints. Compared with constraints, triggers allow us to maintain database integrity in more flexible ways. It also can generate a log of events and to support auditing and security checks. In the SQL standard, checking of conditions, such as price > 0, or existence of a referential integrity constraint is triggered by the DBMS.

However, support for triggers in SQL standard is limited. The trigger events can only be built-in SQL operations (update, insert, delete) on a single base table. Triggers over views are not allowed. Database triggers can only be part of the triggering transactions. For example, unlike our Continuous Query Processing system, Sybase allows only one trigger to be associated with an operation on a table. The action part of a trigger is limited to a sequence of SQL statements. Further, triggering is limited to one level, where the triggered actions themselves do not cause triggers to be fired. Current commercial DBMS have support for dynamic data flow to the users. However, the triggers are table based, which means any change to any record of the table will fire the triggers even though the change may have nothing to do with the triggering queries. For example, if a client wants to monitor the stock for Sybase, but he can only put the trigger on the update of the STOCK table, i.e., any update to the table will cause the execution of trigger. Thus the scalability is really poor [KL95, MS99, RG98]. Our CQPD solves this problem with hashed, record-based triggering. Also in current commercial system, it is the update of data first, then fire the triggers, while in our CQPD, we pushed the new results to the clients first before update the data on the disk. Consequently the users can get the most up-to-date data.

Rule-based Active DBMS: There has been a lot of work on Active DBMS [DBB+88, SPAM91, HLM88, MD89, WC96]. The old model, Passive Database management systems (DBMS) are *program-driven* – user query the current state of database and retrieve the information currently available in the database. Active DBMS, on the other hand, are *data-driven* – users specify to the DBMS the information they need. If the information of interest is currently available, the DBMS immediately provides it to the relevant users; otherwise the DBMS actively monitors the arrival of the desired information and provides it to the interested users as it becomes available. In other words, the scope of the a query in a passive DBMS is limited to the past and present data, whereas the scope of a query in an active DBMS additionally include the future data.

Alert system of IBM [SPAM91] is one of the most important early work on active DBMS. Alert is an extension architecture designed for transforming a passive DBMS into an active one. So basically it reuses old technology and has minimal changes to the language and implementation. Our CQPD also adopts this approach, in addition, we have minimal changes to the SQL interfaces and API. The limitation of Alert system is that it only work on *active tables*, which are append-only tables in which the tuples are never updated in-place and new tuples are added at the end. The only data-driven *active query* is to have a blocking read on its cursor. The cursor opened for an active query gets an EOF in response to a fetch, but then gets new tuples in response to future fetches if new tuples are added to the underlying active tables. Thus, they can only return append-only results.

Continuous Queries: Similarly to active DBMS, Tapestry system of Xerox [TGNO92] implemented continuous queries over append-only databases. They proposed the concept of continuous queries, which are similar to conventional database queries, except that they are issued once and henceforth run "continually" over the database. Unlike Alert, they did not depend on triggers and thus can be implemented on any commercial database that supports SQL.

Incremental Computation: There are some other works about incremental computing [BM95, SLR94], which essentially stores the result of old queries in persistent caches and to reuse them later. But the earlier work only focused on the static query optimization.

None of the database systems above have client profiling or any efficient dissemination features built in. Given the big overlap of clients' requests, it will lead to repetitive computation and bad scalability for both the server and network because clients have to periodically or aperiodically (adaptively) pull the DBMS for latest results.

Dissemination-Based Information Systems [FZ97, FZ98, AAB+98, AAB+99] set up the framework of adaptable middleware for large scale data delivery. They identified three main characteristic for describing data delivery mechanisms: (1) push vs. pull; (2) periodic vs. aperiodic; (3) unicast vs. multicast. These characteristics are orthogonal to each other as shown in Figure 1. We adopt this classification in our implementation. The two bold combinations are the baseline and our CQPD which are used in performance comparison later. The DBIS toolkit serves as an adaptable middleware layer that incorporates several different data delivery mechanisms and provides an architecture for deploying them in a networked environment. But this approach separates data processing (DBMS) from data dissemination (DBIS) so that it may work well with legacy systems, however it results in high overhead unavoidably. For example, the client requests are field twice, once from clients to DBIS, the other from DBIS to DBMS. For the same reason it can not support complicated queries, such as aggregate operations. Also some implementations details are highly unclear in their papers, e.g., the synchronization problem of multiple data sources writing to the Information Broker simultaneously is never discussed.

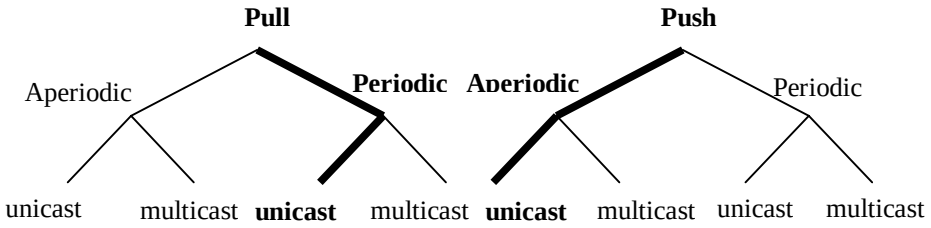


Figure 1. Classification of data delivery methods

Another recent work [LPT99] proposed a distributed event-driven continual query system called OpenCQ. OpenCQ plans to provides push-based, event-driven and context-sensitve information capabilities. By event-driven, they mean that the update events of interest to be monitored are specified by users or applications. By context-sensitive, they mean that the evaluation of the trigger condition happens only when a potentially interesting change occurs. Their DBMS is more general as they allow deletions and modifications to the database updates, not just append-only. However, it is not implemented and just a paper system. Thus many real details, such as how to continuously push the data to the user is not touched in the paper. Also, there is no user-profiling and efficient dissemination techniques (e.g., multicast) in their system, which means they can not make use of the big overlap of user interested data and will have bad scalability.

In conclusion, previous work on data-driven processing and dissemination has the following problems:

- Active/dynamic queries use ad hoc table-based triggering to support dynamic data flow to the clients, which has bad scalability and performance;
- Data processing are separated from data dissemination, which lead to high overhead and lack of support for complicated query requests.

So we designed a new system which combines data processing and dissemination. Imagine clients first send the queries to the server, then whenever there is unpredictable, dynamic data flow in, the system will push the new results to the interested clients immediately and meanwhile, store the data for durability. Thus there is smooth data flow from the data sources to the clients. The users can get latest data with minimum intervention.

3. System Overview

To implement our system, one of the fundamental problems is the communication between the clients and the server. Traditional passive DBMS and even those append-only active DBMS, such as Alert system. It is the client who first set up jdbc connection to the database as below:

```
Connection con = DriverManager.getConnection("jdbc:contSQL", "user_name", "passwd");
```

Then the results are passed back by procedure call, such as:

```
ResultSet rs = stmt.executeQuery ("SELECT company.name FROM stock, company
WHERE stock.change = 10% AND stock.symbol = company.symbol ")
```

In this way, server is just a passive callee and has no control of stopping or resuming query processing. In traditional active DBMS such as Alert, it still takes the same model, but new results of active queries can always be appended at the end of cursor by introducing a new SQL primitive - *fetch-wait*. The fetch-wait corresponds to a *blocking read*, i.e., if the current answer set is exhausted, the process doing a fetch-wait is blocked until one becomes available. However, it can not update the old results as it is append-only. Here is the query above as example, if originally the results are: Sun and Sybase, later an update occurs that makes Sun.change less than 10%. Then the new result should be Sybase. Alert can not do it because it can not handle the update case while our proposed CQPD will be able to.

Our CQPD system uses message passing for communication between clients and server. Client set up queue connection and send out queries as messages:

```
sendmsg.setString("query", "SELECT company.name FROM stock, company
        WHERE stock.change = 10% AND stock.symbol = company.symbol ") ;

queueSender.send(sendmsg);
```

Client receive the results also in the form of messages and the result sets are encapsulated in it:

```
result_msg = (MapMessage) queueReceiver.receive();
```

By using the queued message, server has complete control of query processing and delivery. It can send the whole set of results (i.e.: Sybase) on Sun data update for the query problem before because the trigger event is DELETE or UPDATE. Meanwhile, our CQPD is smart enough to only send the appended result with different tag if the trigger event is INSERT.

We built a stock server as application to test our CQPD system. The system was designed from the ground up to be able to handle both normal and continuous queries simultaneously. To that end, a general design was implemented, with the ability to individually type cast messages as initialization, normal query, continuous query, termination, data, etc… This also makes message decode very simple and efficient. As shown in Figure 2, there are three main sections to the system: data sources, in our example the NASDAQ stock server (a daemon), the continuous SQL server (another daemon), and clients, which in our example are Java applets.

For our example the data sources have been condensed into a single NASDAQ server, although it would work the same with any number of servers. All it does is send messages which are essentially trades. They contain symbol, change, and the number of shares. It can be configured to mimic real world trading practices or deliver a simple progression which can be tracked visually. It can also be configured for bursty or uniform update patterns.

The continuous SQL server has an input queue for initialization, queries, and termination, and another one for updates from the data sources. It has direct write access to the database disk for durability of updates, and a connection to the message passing software so it can send updates to the clients. Since there is a single queue feeding the server updates, one can be ensured they will be processed in order assuming they arrive in order. As updates arrive, they are decoded, sent to waiting clients, and an update query is formed to write the data into the disk file. We will discuss more about continuous CQL server in the next section.

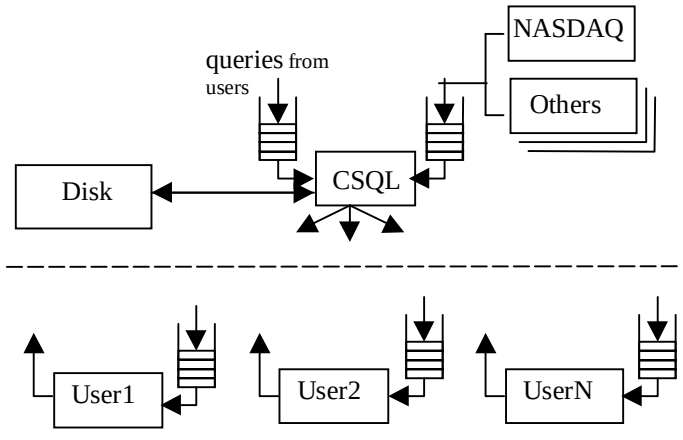


Figure 2. - System Overview

Each client has a connection to the server's request queue, and its own queue to receive replies. In order to ensure correct routing of messages back and forth, upon initialization, it sends a message

containing a unique id for its receiving queue to the SQL server. For normal query processing, the client will send a message to the server, which gets placed in the server's request queue, and then wait for data to appear in its queue. For continuous queries, a single query is made (tagged appropriately), and the client then continually polls its queue for messages. Since the queue is local, all the processing required for polling is done on the client's side with no additional bandwidth or server processing required. When wishing to terminate a continuous query, the client sends a stop message. We have both unicast and multicast support for message delivery.

4. Communication through Message Queues

As the system overview Figure 2 depicts, all the communication between different components happen through queues. Queue implementation is the most crucial to the operation of the project as a whole. The software used to support queue functionality is Java Message Queue™ (JMQ) [Sun99] from Sun Microsystems. It is a messaging middleware that solves complex inter-process communication issues in a cost-effective and industry-standard fashion. JMQ implements the industry standard Java Message Service (JMS) 1.0.1 specification, which provides a standardized API and common messaging strategies for developers. It supports traditional, address-based queues for message delivery as well as asynchronous Publish and Subscribe message delivery model. In our CQPD system, we use the former for unicast and the latter for multicast.

For servers and clients on different machines to have the ability to connect to each other, it's necessary to have the routers running on the machine. The nice thing about having this interface protocol is as long as the routers know of each other, the programs on each machine have to only talk to the resident router on it's own machine. The programs can connect to other machines directly by specifying the machine name instead of going through a local router. However that requires advance knowledge of the machine names before the connection and it's not useful for a general application in the Internet. Since the goal of this project is to provide a general application in the context of the Internet, the ability to connect from anywhere is very important. The system provides any network connection topology, which bodes well for an internet based system. The messages are able to go through arbitrary number of routers and so the network doesn't have to be fully connected. As long as one connection to the network exists, the messages can travel to the destination.

The first thing a new client has to do is connect to the CSQL server. This has to be done through a well-defined and fixed input queue that all connections must know of. To start a connection, the client must send a well-defined initiation message to the CSQL server using the input queue of the CSQL server. Once the server received the message, it will add the client to the table of clients and open up a queue to send messages back to the client. The first such message is the acknowledge message telling the client the session has been successfully opened and operational.

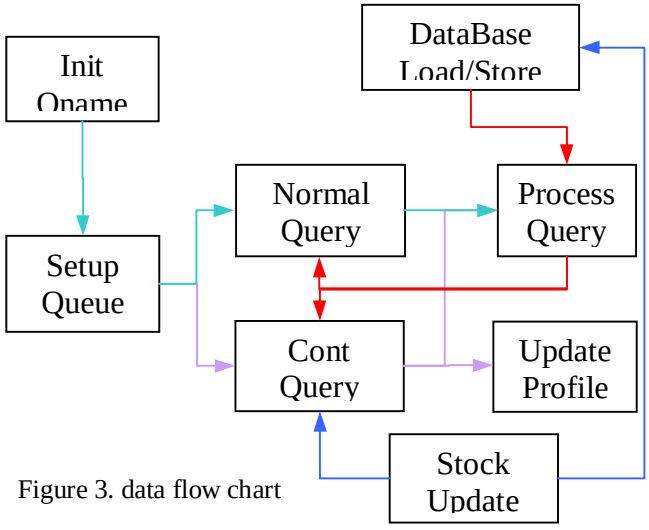


Figure 3. data flow chart

All the client's request must come through the input queue of CSQL and now the client can choose either normal query or continuous query to send to the contSQL server for processing. The data flow is as depicted in Figure 3.

5. Continuous SQL Server

We implemented an extreme lightweight active SQL engine in Java. It is encapsulated in a package called edu.ucb.contSQL. There are two reasons for choosing Java as developing language:

- Java is a new and prospective language. There have been a lot of query processing (SQL engine) on C/C++, but very few on Java
- Java is "write once, run anywhere", it is easy to ported to UNIX, LINUX or Windows platform.

Our contSQL server is generic, extensible and portable. It consists of three parts: contSQL JDBC driver, parser and SQL statement executor as in Figure 3.

JDBC is to provide a standard API and conformance level for programming SQL databases in Java. The JDBC driver manager is available from: http://splash.javasoft.com. Our contSQL driver implements only a subset of the functionality required by a JDBC Compliant driver. The most notable exception to a fully compliant driver are the PreparedStatement, the CallableStatement and Database MetaData classes. Neverless, enough of the JDBC functionality is implemented to allow you to preform queries and updates and also to make discoveries about the result sets.

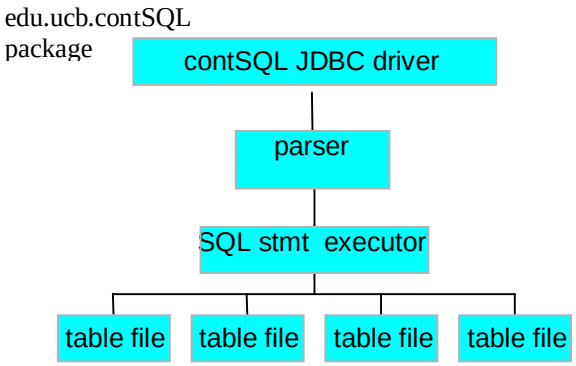


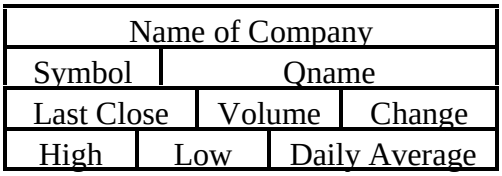
Figure 4. architecture of contSQL server

Both of the normal and continuous queries have the same syntax as SQL standard. The parser is generated through JavaLex [BA99] and JavaCup [HFA99]. JavaLex is a lexical analyzer generator for Java. JavaCup is Java based Constructor of Useful Parsers (CUP for short). CUP is a system for generating LALR parsers from simple specifications. Thus JavaLex and JavaCup perform roughly the same function as the standard Unix tools *lex* and *yacc*. We used them to develop the parser, which analyzed the SQL commands and map them to actions that must occur. First *scanner* is produced by javaLex and takes care of breaking the input into tokens, while the *parser* is produced by JavaCup and determines whether the incoming tokens make some sense in the context of the grammar.

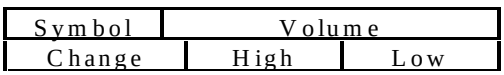
The core part of contSQL server is the SQL statement executor. We implemented six SQL commands: CREATE TABLE, DROP TABLE, INSERT, DELETE, UPDATE and SELECT. Simple join algorithms are implemented for SELECT. But as it is just an experimental system, it can not handle range query or nested SELECT. Our CQPD can proceed both static and dynamic SQL, both interactive and embedded SQL.

The normal query processing is processed just like any commercial DBMS. For continuous query, there are two profile hash tables. Our system is based on the assumptions that there is a big overlap for users' queries and users specify their interested records by the keys in their queries. We designed a record-based triggering technique for improving the efficiency and scalability. It is accomplished through the two profile tables, one is clients table which holds the current users along with their queues, indexed by the key of their interested record, in our example, the specific stock symbol. We use linked list to solve the hash table collision problem. Client who is interested in multiple records will have multiple entries in the clients table. The other, records table, holds the records that are interested by the continuous queries.

When there is a continuous query flow in, it first updates the clients table, registers the records that it is interested in. Then it uses hash table lookup to check the records table to see if their interested records are there. If hit, it can just return the results instantly. Otherwise, it has to go to the disk to get the data, just as normal query does. However, after getting the data, it will insert them into the records table for future use.



Normal Update Message



Continuous Update Message

Figure 5. Normal and Continuous Message Format

When a stock update is received, contSQL used the record table to figure out all the required information based on previously cached values. For example, the new highest, new lowest bid price and the trading volume can be updated. Then it consults the clients table to see which users are interested in a particular stock and multicasts a message encapsulating the data. Figure 3 depicts the whole flow of the system from the messages' point of view. Stock update will also do the necessary update function of the database with the new data.

The contSQL server also supports the aggregate operation, such as sum. As in demo Figure 6, the total volume sum of all trading stocks is calculated by incremental computing. Whenever there is an stock update, it will re-compute the total volume based on previous cached results. Thus it saves a lot of disk lookup time and calculation.

When a user issues a query, all the information about a particular stock must be sent back since the history of the user isn't available. However in a continuous query, the server knows exactly what information the user received last. So it can go ahead and send minimal identification field (symbol), and only the values that have changed for that particular stock. Figure 5 shows the relative size difference between two messages for two query types. Thus it's clear that network traffic will be substantially less due to continuous query result messages if they can be used instead of normal result message. For the particular example of a stock server, high and low values will only be included in the message if they happened to have changed from the previously sent message. For multicast to be supported, the output queues will be changed to Topics where multiple subscribers would be supported. However it's still necessary to have Qname for each client because profiling needs some way of distinguishing between the users and sending the correct update information

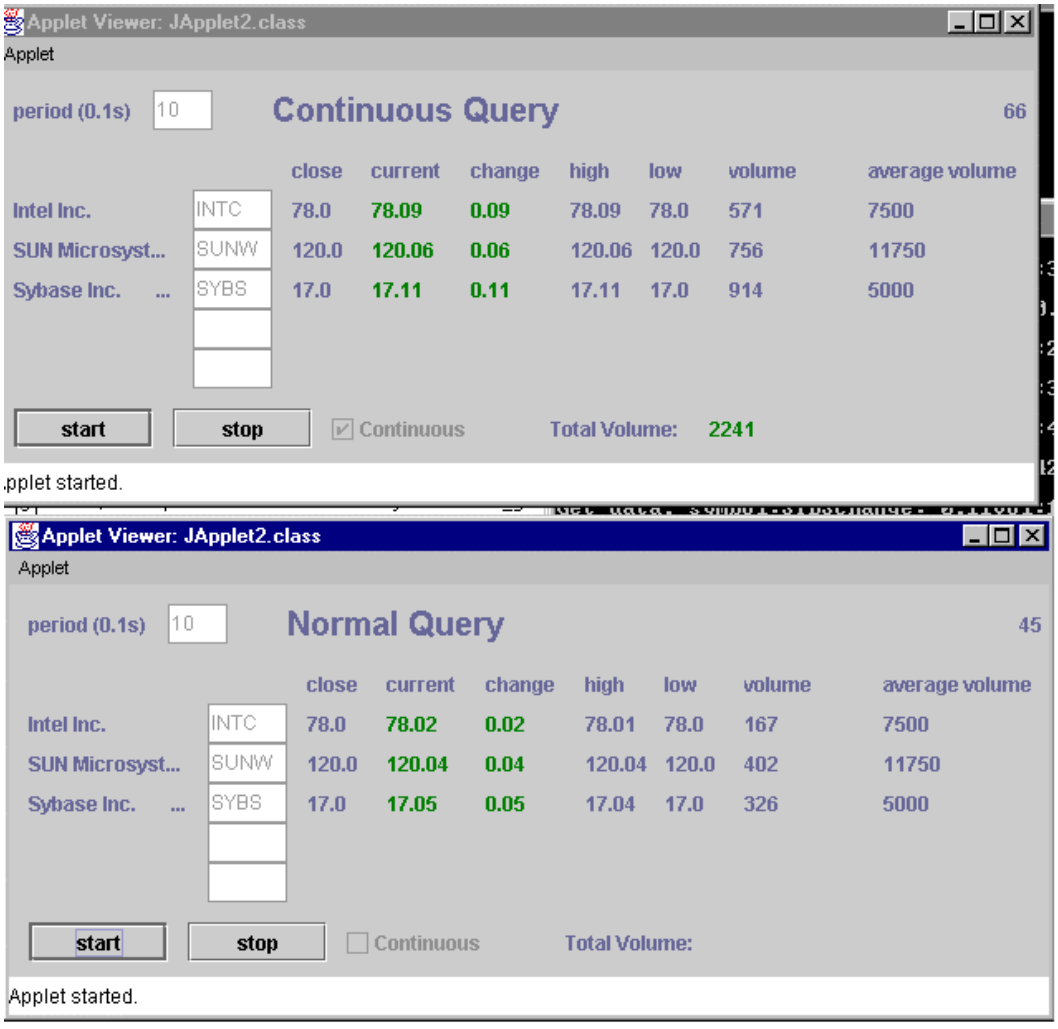


Figure 6. Demo snapshot of the CQP system

to each of the users.

6. Performance Analysis

Our two main metrics for performance in this design, was the total latency of the stock quotes received (as measured by the time difference in when the stock changes on the server, and when it is seen by the client), and second, the number of messages sent, and therefore bandwidth required. Additionally, another metric, the rate at which updates can be received, is also important. Our test environment was a Pentium II, 300MHz, with 128MB RAM running Windows NT workstation SP5. Our simulated server was set to deliver only three stocks, and our applets were configured only to look for those stocks. The demo figure is displayed as Figure 6. The normal query, as the baseline, is the periodic unicast pull. The continuous query, is the aperiodic unicast push. Both are marked as bold in Figure 1. As we need to use timestamps to compare the performance, both of them have to run on a single machine. So we did not use multicast for continuous queries. But obviously with large scale of clients in real world, multicast will work better with our CQPD.

We set the NASDAQ simulated server to send data with incremental positive changes in the step of 0.01. From the figure we can tell that the continuous query getting the data far ahead of the normal one even it get additional messages for the aggregate operation - the total sum of all stocks. However, this still is the worst case, although scaled down, where the user is interested in every stock. So by doing better in the worst case, we know we will do significantly better in the general case.

Measurement of the total latency, loosely equivalent to the age of a quote, was done by creating a log file on a simulated NASDAQ server, and another generated by the client. The queries generated by the applet were simple selects, grabbing all data for a single stock. The time is the time required for the NASDAQ server to send a message, the SQL to receive and decode it, and store it somewhere, plus the time required to process a clients normal select or determine which clients want this update, send it to them, and finally the time required for the client to pull it from its message queue. The time was just the difference in times in the logs. In order to ensure time synchronization, this test was run on a single machine with all clients, the continuous SQL processor, and the simulated server running simultaneously. We generated roughly 200 updates, and varied the number of clients from 1 to 4. The polling period was set at 0.1s, and the updates period averaged about 1s. Results are plotted as Figure 7 and displayed in Table 1. These periods put the normal select in the same region as continuous. A low normal query period coupled with a low update period (say 60s average each), would result in data being at least 30s out of date for the normal query, however the continuous query would remain constant, since updates are sent as soon as they arrive, but this result is obvious. For the 0.1s / 1s case, they data received showed that the continuous query consistently delivered data which was only half as old as the continuous case, and scaled with the number of users. For normal query, we ignored all but the first select during and update period. This helps the normal query, but as seen it still wasn't able to overcome the continuous query's performance. Network travel time would add twice as much for the normal queries as it would for the continuous case. This is just because normal queries use a request/reply model, as opposed to a one way push model. An estimate of at least 150ms for continuous and 300ms for normal queries. These amount to at most a 33% increase. The minimums represent the case where data has been updated just before the query arrives, and thus is the time required to send it to the contSQL, process it and return it to the client, plus any waiting time in the queues. With multiple clients

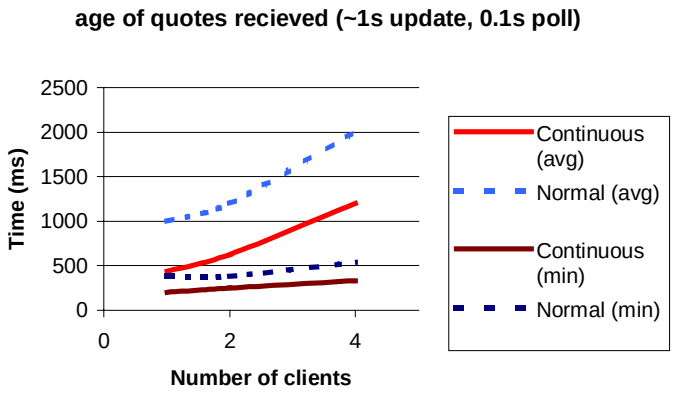


Figure 7. Performance comparison for the normal and continuous queries.

feeding queries into the server queue, it is possible for it to get back logged and require significantly more time to process. This partially explains the exponential growth of the curves. Additionally there is the added processing required to run multiple clients, and both servers on a single, somewhat underpowered node.

	Base latency				with additional network latency			
number of clients	C_{avg}	N_{avg}	C_{min}	N_{min}	C_{avg}	N_{avg}	C_{min}	N_{min}
1	437	1000	200	380	737	1300	500	680
2	625	1200	250	380	925	1500	350	680
4	1200	2000	330	540	1500	2300	630	840

Table 1. Latency comparison for normal and continuous queries under different # of clients

he second metric, the number of messages sent, is highly dependent on typical interests of users, and minute by minute changes in market. Thus we make some estimates on them. For short periods of time, we can express the number of messages sent as:

$$Nmsg_{n\ (per\ user)} = 2 * N_{stocks} * R_{poll}$$
$$Nmsg_{c\ (per\ user)} = N_{stocks} * R_{update} * f$$

Where R_{poll} is the polling rate, R_{update} is the typical rate of incoming updates, N_{stocks} is the average number of stocks a user is interested in, and f is a weighting function which represents the percentage of updates that are wanted by that user. The 2 comes from the fact that in the normal case, queries must be sent, and replies must be read. In continuous queries, aside from the initial query, no others are required to receive updates. The questions is: is continuous query always better? If not, what solution is there?
Dividing by R_{update} we can plot these as a function of the ratio of R_{poll} / R_{update}.

$$Nmsg_n = 2 * N_{stocks} * R_{poll} / R_{update}$$
$$Nmsg_c = N_{stocks} * f$$

Therefore the trade off point is at: $R_{poll} / R_{update} = f/2$. We assume there is a relatively low probability that the user in question wants the latest update of some random stock, say 1%. This puts the break even point at 0.005, as seen in Figure 8. The total number of messages is just the number per user times the number of users (for unicast). The left side of the graph represents high update rate, and the right represents high polling. We can see that normal quickly becomes unbearable when highly up-to-date data is required. The bandwidth is further compound of this. Since a naïve normal query will return all data, whether or not it has changed, the typical message size could be 4 times as large. A polling rate 30 times smaller than the update period still requires more than fifty times the bandwidth.

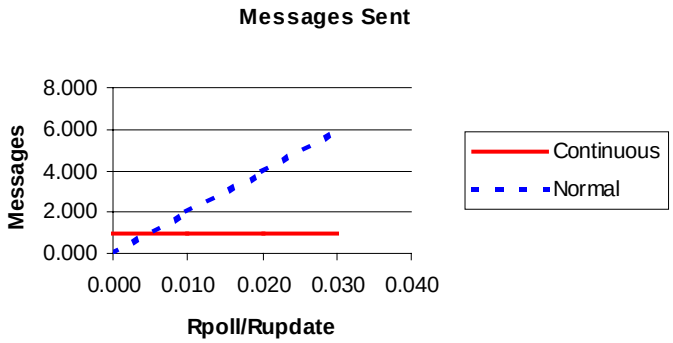


Figure 8. Number of messages requirement per user under different polling vs. update rate.

For continuous queries, the applet will retrieve any waiting messages, this allows all updates to be received in batches. Obviously a normal query will request, and get a response after some time. Clearly it is unlikely in the high update case that it could get every update. This is not a big deal for a stock server being used by a casual user, however there are more demanding applications which might require every update be seen.

A little queuing theory [Don98] can show us exactly how many queries we can service. First I'll assume that the system can be represented by a single queue, as shown in Figure 9. Second, the minimum times for n=1 have a utilization near zero, which allows me to say that theses are approximately $T_{server(mode)}$.

$T_{server(N)}$ = 380ms $T_{server(C)}$ = 200ms
T_N = 1000ms T_C = 437ms

$Utilization_{mode} = (T_{mode} - T_{server(mode)}) / T_{mode} = \{0.62, 0.54\}$
$\lambda^{-1}_{mode} = \{612ms, 370ms\}$
$\lambda_{mode} = \{1.63Hz, 2.7Hz\}$

These numbers represent the rate at which updates are issued from the NASDAQ server for continuous queries, and the issue rate for normal queries. So we can increase updates by more than a factor of 2, but can't really increase the query rate. If utilization were close to 0, we still could only issue normal queries at a rate of about 3Hz, however this limitation is clearly not present in a continuous query. Since testing was done a single machine, the high polling rate multiplied by the number of clients skews the results.

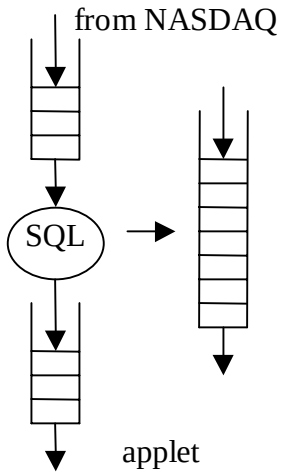


Figure 9. Queue Simplification

For our incremental operation, total NASDAQ volume for today, which could be calculated by first storing data in an append only fashion, then selecting all stocks from the database which have an update from today, and summing their individual volumes, was accomplished by only adding the volume of the incoming trades. This significantly cut down on the computation required, but more importantly reduces the disk accesses to zero, aside from the initial access to get a starting value, although if the server had been started before the NASDAQ, it could just be set to zero.

7. Conclusions

We can clearly see the merits of continuous query, not only in delivering up-to-date data, but also in the ability to significantly reduce the network bandwidth required, and further it is more adept at delivering all updates as opposed to the most recent. This fast update is due to the following saves:

- polling query travel time;
- query running time (including the disk lookup)
- repetitive computation for aggregate operations (e.g. sum)

In order to achieve this performance using record-based triggering, some generality was sacrificed. However, for applications such as stock, weather, sports or map information servers, or even more complex systems, where queries can be condensed into select statements with interested records known *a priori*, this is ideal. Finally our basic approach to incremental computing avoids necessary expensive disk accesses for potentially the entire database. Our prototype programs verified all of our initial assumptions. It is our belief that next generation of DBMS should have full efficient dissemination functionality for the ubiquitous, large scale and dynamic information flow in the Internet era.

8. Future Work

One of the most important future work is to have our CQPD to handle more general continuous queries, that is, try to get rid of the limit of pre interested records in the query. Another adaptation to reduce the number of messages sent is to use a periodic push, which wouldn't push out every update, but just the one in the latest time interval. This is especially useful in the low R_{poll} / R_{update} region where currently normal queries send out fewer messages.

Currently we use a data driven model where any change forces an update. Additionally we will implement a more general event driven model (e.g. time intervals, and more complex data changes like hitting a minimum or maximum value)

One last are is in the scalability of the continuous SQL processing engine. We need to ensure that it can efficiently handle large numbers of users all requesting different stocks. This can be achieved in the queue initialization phase. If the server detects too many continuous queries running on the server, it will deny some continuous query initialization requests depending on their priority. It will enforce them to use normal query.

References:

1. [AAB+98] D. Aksoy, M. Altinel, R. Bose, U. Cetintemel, M. Franklin, J. Wang, S. Zdonik, "Research in Data Broadcast and Dissemination" (Invited Paper), Proc. 1st International Conference on Advanced Multimedia Content Processing, November, 1998.

2. [AAB+99] M. Altinel, D. Aksoy, T. Baby, M. Franklin, W. Shapiro, S. Zdonik, "DBIS Toolkit - Adaptible Middleware for Large-Scale Data Delivery", Demo Description, ACM SIGMOD Intl. Conference on Management of Data 1999.

3. [BA99] E. J. Berk and C. S. Ananian, "JLex: A Lexical Analyzer Generator for Java", http://www.cs.princeton.edu/~appel/modern/java/JLex/, 1999

4. [BM95] L. Baekgaard, L. Mark, "Incremental Computation of Nested Relational Query Expressions", ACM SIGMOD International Conference on Management of Data 1995, pages 111 -148

5. [Don98] G. Donald, "Fundamentals of queueing theory", 3rd ed. New York : Wiley publisher, 1998

6. [DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. MCCarthy, A. Rosenthal, S. Sarin, M. Carey, M.Livny, R. Jauhari, "The HiPAC Project: Combining Active Database and Timing Constraints", ACM-SIGMOD Record, 17(1):51-70, March 1998

7. [FZ98] M. Franklin, S. Zdonik, ‘‘Data in Your Face”: Push Technology in Perspective (Invited Paper), ACM SIGMOD Intl. Conference on Management of Data 1998, pages 516-519

8. [FZ97] M. Franklin, S. Zdonik, "A Framework for Scalable Dissemination-Based Systems" (Invited Paper) International Conference Object Oriented Programming Languages Systems (OOPSLA 97)

9. [HFA99] Scott Hudson, Frank Flannery, C. Scott Ananian, "CUP Parser Generator for Java", http://www.cs.princeton.edu/~appel/modern/java/CUP/, 1999

10. [HLM88] M. Hsu, R. Ladin, D. McCarthy, "An Execution Model for Active Data Base Management Systems", Proc. 3rd International Conference on Data and Knowledge Bases - Improving Usability and Responsiveness. (ICD 88), 1988

11. [KL95] G. Koch, K. Loney, "Oracle: the Complete Reference", third edition, McGraw-Hill, 1995

12. [LPT99] L. Liu, C. Pu, W. Tang. "Continual Queries for Internet Scale Event-Driven Information Delivery", Special issue on Web Technologies, IEEE Transactions on Knowledge and Data Engineering, Jan. 1999

13. [MD89] D. MaCarhty, U. Dayal, "The Architecture of an Active Database Management System", ACM SIGMOD Intl. Conference on Management of Data 1989, pages 215 - 224

14. [MS99] Microsoft Inc. "Personal Online Support for SQL Server 7 -- Triggers (T-SQL)", http://support.microsoft.com/support/SQL/Content/inprodhlp/_create_trigger.asp, 1999

15. [RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, J. Blakely, "Situation Monitoring for Active Databases", Proc. 15th International Conference on Very Large Data Bases, 1989. pages 455-464

16. [RG98] R. Ramakrishnan, J. Gehrke, "Database Management Systems", second edition, McGraw-Hill, 1998

17. [SLR94] P. Seshadri, M. Livny, R. Ramakrishnan, "Sequence Query Processing", ACM SIGMOD International Conference on Management of Data 1994, pages 430 - 441

18. [SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, C.Mohan, “Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS”, ACM SIGMOD International Conference on Management of Data 1991, pages 469 – 478

19. [Sun99] Sun Microsystems Inc., Java Message Queue, http://www.sun.com/workshop/jmq, 1999

20. [TGNO92] D. Terry, D. Goldberg, D. Nichols, B. Oki, "Continuous Queries over Append-Only Databases", ACM SIGMOD International Conference on Management of Data 1992, pages 321 – 330

21. [WC96] J.Widom and S. Ceri, "Active Database Systems", Morgan Kaufman, 1996.