Update Monitoring: The CQ Project

Calton Pu* and Ling Liu

Department of Computer Science & Engineering
Oregon Graduate Institute
P.O.Box 91000 Portland, Oregon 97291-1000 USA
{calton,lingliu}@cse.ogi.edu

Abstract. In rapidly growing open environments such as the Internet, users experience information starvation in the midst of data overload, due to difficulties similar to finding the needle in a haystack. Update monitoring is a promising area of research where we bring the right information to the user at the right time, instead of forcing the user through manual browsing or repeated submission of queries. As an example of update monitoring research, we outline our work in the Continual Queries (CQ) project, including its basic concepts, software architecture, and current implementation status.

1 Introduction and Problem Statement

The World Wide Web (usually referred to as *the Web*) has made an enormous amount of data freely accessible over the Internet. However, finding the right information in the midst of this mountain of data has been likened to finding the proverbial needle in a haystack. This phenomenon has been called "information starvation due to data overload."[2] Commonly used search engines, including web robots (e.g., AltaVista) and indexers (e.g., Yahoo!), have ameliorated the situation somewhat, but the current exponential growth of the Web is quickly aggravating the fundamental problem.

We divide the problem of finding relevant information into two parts. The first part is the search for historical data in the Web. Given its static nature, historical data is best suited for search engines, and generally speaking, data warehousing tools. The second part of the problem is *update monitoring*, which deals with the new information arriving into the Web and the databases. There are many applications in both parts. Let us consider a simple example in decision support systems. On the one hand, historical data is used in long term projections and planning, for example, by Wal-Mart in the selection of inventory. On the other hand, update monitoring is used in (near) real-time decisions, for example, by investment bankers in the buying and selling of stock.

---

* The CQ project is partially funded by DARPA grant MDA972-97-1-0016, and grants from Intel and Boeing.

[2] Gio Wiederhold of Stanford University seems to have been among the first to use this phrase.

While both historical data and update monitoring have interesting applications and research challenges, we focus on update monitoring in this paper. There are three reasons for this bias. First, managing read-only historical data is a more mature area, with many commercial data warehouse systems available. Consequently, many of the most obvious research questions have been answered. Second, update monitoring is a problem that requires solutions that combine writes, which are usually handled by a transaction processing systems, and reads, which are usually handled by database management systems. Third, update monitoring introduces special difficulties when heterogeneous data sources (e.g., from Web pages and relational databases) are being monitored together. As a result, update monitoring presents some interesting new research challenges.

In the Continual Queries (CQ) project at Oregon Graduate Institute we are developing and techniques and a software toolkit for update monitoring and event-driven information delivery on the Internet. The practical result of the CQ project is a "personalized update monitoring" toolkit based on *continual queries* [12]. In contrast to conventional database queries, continual queries are standing queries that are issued once and run "continually" over the source data. As updates to the data sources reach a specific threshold or timed event, the standing query is (conceptually) re-evaluated and new results returned to the user or the application that issued the query. We say that the query is conceptually re-evaluated because of the variety of algorithms and approaches to the standing query re-evaluation.

For each continual query, an update monitoring program (CQ robot for short) creates distributed programs that act together as an intelligent assistant, keep track of information sources that are available (on the Web and elsewhere), how to access them, and the changes that happened. Whenever updates at the data sources result reach a specific update threshold or a timed event, the CQ robot computes and integrates the new results and presents them to the user. Compared with the pure pull (such as DBMSs, various web search engines) and pure push (such as Pointcast, Marimba, Broadcast disks [1]) technology, the CQ project can be seen as a hybrid approach that combines the pull and push models by supporting personalized update monitoring through an integrated client-pull and server-push paradigm.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of continual queries and the CQ project. Section 3 describes the architecture of CQ software and implementation. Section 4 outlines a client-server design of the CQ architecture. Section 5 describes how continual queries may be executed efficiently. Section 6 describes in some detail the opportunities and problems presented by the push technology. Section 7 summarizes the status of current implementation in the CQ project. Section 8 concludes the paper.

2 Basic CQ Concepts

The goal of the CQ project is to develop techniques and a toolkit for update monitoring with event-driven delivery in an open and dynamic evolving environment such as the Internet and intranets. We pursue this goal along two dimensions: The first dimension is to develop a set of methods and techniques that can incorporate distributed event-driven triggers into the query evaluation and search process to enhance information density and improve system scalability and query responsiveness. The first dimension is summarized in this and the following sections (2 and 3). The second dimension is to build a working system that demonstrates our ideas, concepts, and techniques developed for continual queries using real-world application scenarios. The second dimension is summarized in Sect. 7.

2.1 Concepts and Definitions

A continual query is defined as a triple (Q_{cq}, $Trig_{cq}$, $Term_{cq}$), consisting of a normal query Q_{cq} (e.g., written in SQL), a trigger condition $Trig_{cq}$, and a termination condition $Term_{cq}$. The initial execution of a continual query is performed as soon as Q_{cq} is issued and the whole result is returned to the user.

The subsequent executions of Q_{cq} happen when the trigger condition $Trig_{cq}$ becomes true. Currently, CQ supports two types of trigger conditions: *time-based event triggering* and *content-based event-triggering*. For time-based event triggering, three modes are supported:

1. immediate, whenever a change to the source data occurs;
2. at a specific time point (e.g., execute Q_{cq} every Monday or every first day of the month); and
3. at regular time intervals (e.g., execute Q_{cq} every two weeks).

For content-based event triggering, we support a variety of content-based conditions. Examples include:

1. a simple condition on the database state (e.g., execute Q_{cq} whenever a deposit of $5,000 is made);
2. an aggregate condition on the database state (e.g., execute Q_{cq} when the total deposits reach $100,000), and
3. a relationship between a previous query result and the current database state (e.g., execute Q_{cq} when a total of $100,000 dollars in deposits have been made since the previous execution of Q_{cq}).

The termination condition $Term_{cq}$ specifies the event that determines the end of a continual query. Both the trigger condition $Trig_{cq}$ and the termination condition $Term_{cq}$ will be evaluated prior to each subsequent execution of Q_{cq}.

2.2 Continual Query Examples

Several examples that illustrate the uses of continual queries. First is "*notify me in the next week each time Microsoft stock price rises by 10%*". This request is codified by the following three components:

- Query: Result(`SC,SP`) = `SELECT Stock.company, Stock.price FROM Stock WHERE Stock.company = ''Microsoft'';`
- $Trig_{cq}$: `Stock.company = ''Microsoft''` .and. `Stock.price > SP*1.1`
- $Term_{cq}$: `one week from query creation`

Second example is *"tell me the flight number of the plane whenever it has remained in this sector for more than 5 minutes"*.

- Query: Result(`FN`) = `SELECT AirControl.FlightNumber FROM AirControl WHERE AirControl.SectorTime > ''5min'';`
- $Trig_{cq}$: `AirControl.SectorTime > 5min`
- $Term_{cq}$: `nil`

Third example is *"for the next month, report which manufacturers can supply 1000 units per warehouse whenever the average storage level of canned soup is below 200 units"*.

- Query: Result(`MANUF`) = `SELECT Manufacturer.Name FROM Manufacturer WHERE Manufacturer.Supply > ''1000''` .and. `Manufacturer.Item = ''Canned Soup''`
- $Trig_{cq}$: `Inventory.Item = ''Canned Soup'' .and. Inventory.StockLevel < ''200''`
- $Term_{cq}$: `one month from query execution`

Fourth example is *"At 5pm every day, notify me the itemized amount and classification of materials coming into or going out from these ports and their origin or destination"*.

- Query: Result(`AMOUNT, CLASS, ORIGIN, DESTINY, PrevLoc`) = `SELECT Materials.Value, Materials.Type, Materials.StartPoint, Materials.EndPoint, Materials.Location FROM Materials WHERE Materials.Location != PrevLoc;`
- $Trig_{cq}$: `DayTimer = ''17:00''`
- $Term_{cq}$: `nil`

2.3 Continual Semantics

Let us denote the result of running continual query Q_{cq} on database state S_i as $R_{cq}(S_i)$. The result of running a continual query Q_{cq} is a sequence of answers $\{R_{cq}(S_1), R_{cq}(S_2), \ldots, R_{cq}(S_n)\}$ obtained by running query Q_{cq} on the sequence of database states $S_i, 1 \leq i \leq n$, each time triggered by $Trig_{cq}$, i.e., $\forall S_i,\ Trig_{cq} \wedge \neg Term_{cq}$.

If the termination condition $Term_{cq}$ is nil, Q_{cq} will produce results from $R_{cq}(S_1)$ to $R_{cq}(S_\infty)$. Otherwise, Q_{cq} will produce results from a starting time t_1 to a final time t_n, when $Term_{cq}$ becomes true. In other words, Q_{cq} (the sequence) ends when the termination condition becomes true.

Each time $Trig_{cq}$ is triggered, conceptually Q_{cq} is evaluated against the current state of the database S_i, and the result $R_{cq}(S_i)$ is sent to the user who issued Q_{cq}. In general it is expensive to re-evaluate the whole query over the entire source data for each execution of a continual query, although in some circumstances (e.g., legacy databases and some file systems) it may be unavoidable to reprocess the query from scratch. Therefore, it is important to find optimization steps that can bypass the complete re-evaluation and thus avoid the duplicate computation and unnecessary data transmission. In paper [12] we describe a strategy to generate $R_{cq}(S_i)$ from $R_{cq}(S_{i-1})$ incrementally, thus reducing both processing time and network transmission bandwidth.

3 System Architecture

As mentioned earlier, the ultimate goal of the CQ project is two-fold: On one hand, we intend to develop an adaptive system architecture and a set of techniques for update monitoring in open environments. On the other hand, we provide effective support for enhanced data transparency, data quality, and system scalability and responsiveness. The method and key techniques of the CQ system development include:

- using the notion of continual queries to support customized (or personalized) update monitoring based on users' preference and requirement (user pull followed by server push),
- incorporating broadcast-based server push sources with the pure pull based data sources in the continual queries service provision,
- integrating distributed query processing and dynamic optimization techniques into the continual query evaluation process for achieving effectiveness and responsiveness of the system.

The first generation of the CQ system has a three-tier architecture: client, server, and wrapper/adapter. The client tier is primarily responsible for receiving users' request and expressing such request in the form of CQ query Q_{cq}, CQ trigger $Trig_{cq}$, and CQ termination condition $Term_{cq}$. The client manager is also in charge of user registration and providing CQ users with system utilities such as browsing or editing installed continual queries. The client manager currently has four main components as shown in Fig. 1:

1. The form manager that provides the CQ clients with fill-in forms to register and install their continual queries;
2. The registration manager which allows clients to register the CQ system with valid user id and password, and return the clients a confirmation on their registration;
3. The client and system administration services which provide utilities for browsing or updating installed continual queries, for testing time-based triggers and content-based triggers, and for tracing the performance of update monitoring of source data;
4. The Client manager which coordinates different client requests and invokes different external devices.

For instance, once a continual query request is issued, the client manager will parse the form request and construct the three key components of a continual query (Q_{cq}, $Trig_{cq}$, $Term_{cq}$), before storing it in the CQ system repository. Although not a direct part of the CQ project, one could imagine value-added update monitoring services based on CQ, where a continual query request can be posted in natural language through either voice or hand-writing or both. Recall the example given earlier: "*notify me whenever Microsoft stock price rises by 10%*". By hooking up the CQ client with a natural language text recognizer, or hand-writing recognizer, or voice recognizer, we can parse this request and automatically generate the query, the CQ trigger, and the termination condition for this request. The results can be returned to the user either by email, by fax, by phone, or through user-specific bulletin posting.

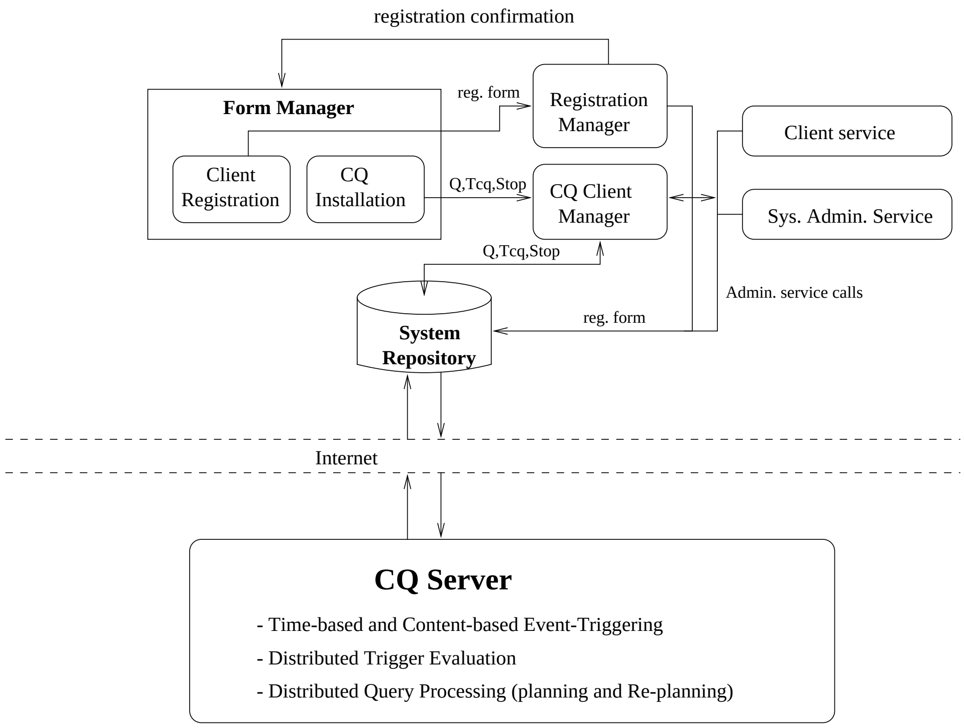

Fig. 1. The CQ Client Tier Architecture

The second tier is the CQ server which is responsible for evaluating continual queries based on the specified update threshold of interest and providing capability for handling distributed event-based triggers. The CQ server manager consists of three key components:

The event-driven update monitor. It coordinates with the CQ wrappers and adapters to track the new updates to the source data. Its main tasks include: (1) the selection of relevant data sources for evaluating CQ triggers, (2) the generation of distributed triggers that can be executed at the selected data sources, and (3) the evaluation of CQ trigger by combining the sub-results of distributed triggers. In short, it decides when to issue an execution of the installed CQ queries.

The CQ-trigger-firing daemon. It is in charge of calling the event-driven update monitor to evaluate the CQ trigger condition for each installed continual query. Two kinds of events are supported in the first generation of the CQ system: (1) a clock daemon checks specified date and time events, and (2) a content-based trigger-firing daemon checks specific update thresholds. In short, the CQ trigger-firing daemon deals with the timing for firing the distributed trigger evaluator, i.e., when a CQ trigger needs to be evaluated.

The continual query evaluator. It is responsible for processing the query Q_{cq} when the trigger condition $Trig_{cq}$ is true. It also provides a guard for the $Term_{cq}$ condition of a query to guarantee the semantic consistency of the continual query $(Q_{cq}, Trig_{cq}, Term_{cq})$. The key components of CQ evaluator include: the query router [11, 10], the query planner [9, 12], the dynamic query replanning manager, and the query result assembler. The query router is a key technology that enables the CQ system to scale in order to handle thousands of different information sources. When the user poses a query, the router examines the query and determines which sites contain information that is relevant to the user's request. Consequently, instead of contacting all the available data sources, the CQ evaluator only contacts the selected sites that can actually contribute to the query.

The CQ server tier is also in charge of removing the installed continual queries whenever their termination conditions become true.

The third tier is the CQ wrappers/adapters tier. The CQ query evaluator and the event-driven update monitor talk to each information sources using an information wrapper. A wrapper is needed for each site because each one has a different way of requesting data and a different format for representing its results. Each wrapper is a specialized data converter that translates the query into the format understood by the remote site. As the result comes back, the wrapper packages (translates) the response from the site into the relational database format used by the CQ system. Figure 2 shows a sketch of the coordination between CQ server and CQ wrappers/adapters, the key components of CQ server and the critical interconnections among them.

4 Client-Server Design

Depending on the need of the application, the CQ client manager, the CQ trigger daemon, the event-driven update monitor, the query router, query planner, and

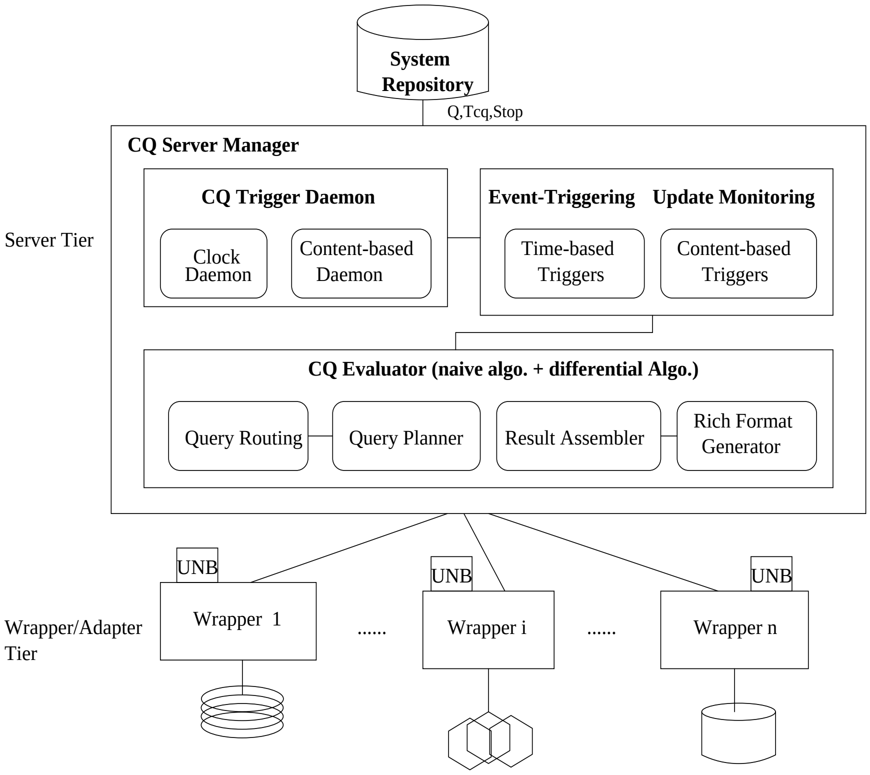

Fig. 2. The CQ Server and Wrappers/Adapters Architecture

query result assembler could be located on a single machine, or distributed among several computers connected through local or wide area networks. CQ uses the most flexible client-server arrangement which is customizable with respect to the particular system requirement of the applications. In this demo, for example, we plan to have the query router running on a powerful server machine, where we also maintain a library of all the current information wrappers including the source capability profiles. Figure 3 shows three different scenarios for multi-layer client/server coordination among CQ components.

5 A Model for Efficient Execution of CQ

Continual queries are standing queries that run continually until the termination condition becomes true. Whenever an relevant update is performed, the CQ system will trigger the execution of the corresponding continual queries. It is obvious that the subsequent executions of a given continual query is only interested

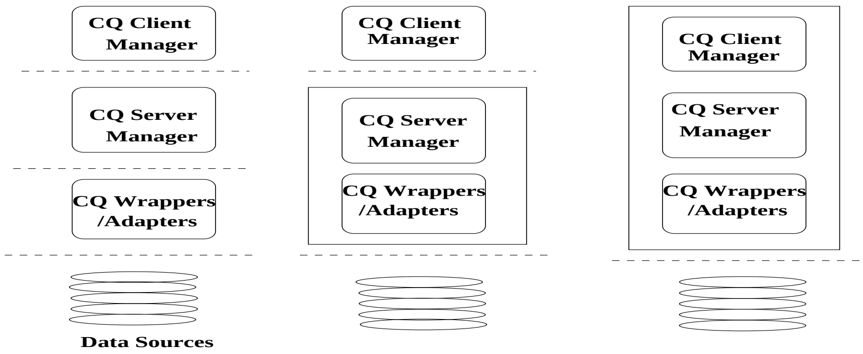

Fig. 3. Example Scenarios of the Client/Server coordination among CQ components

in those data that have been updated since the previous execution. In the situation where the amount of updates is small, one way to optimize the subsequent executions of a given CQ query is to use differential evaluation method such that queries that can be answered using delta information (i.e., the updated data) rather than the full set of base data. Similar techniques have been widely used in incremental view materialization [3, 5, 7, 6, 13].

More concretely, recall Sect. 2, we have defined a CQ query as a sequence of query results, modeled by $Q_{cq}(S_1)$, ..., $Q_{cq}(S_n)$, where $Q_{cq}(S_i), (i = 1, ..., n)$ is the result of running Q_{cq} on the database state S_i. Quite often there are situations where users are more interested in the difference between $Q_{cq}(S_i)$ and $Q_{cq}(S_{i+1})$. This can be accomplished by naively executing the entire query and then filtering out the part of the query result that is the same as the previous result. This simple and straightforward approach can be quite expensive, especially in the Internet environment where query results need to be gathered from multiple source data repositories. An obviously more attractive approach is the differential query evaluation method, which is particularly powerful when $Q_{cq}(S_i)$ is relatively large, and only a small percentage of the result changes from state S_i to state S_{i+1}.

We have proposed a differential re-evaluation algorithm [12]. The key idea behind this algorithm is the following: We produce $Q(S_{i+1})$ by incrementally updating $Q(S_i)$. More concretely, in contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be performed by using the differential form of the query. This way, we avoid reprocessing the entire query from scratch. When the changes are substantially smaller compared with the latest query execution result, this differential update will be more efficient than reprocessing the entire query.

The differential re-evaluation algorithm (DRA) is invoked by the CQ manager based on the epsilon specification associated with the given CQ. We assume that the information available when the DRA is invoked includes:

- the CQ specification (Q_{cq}, $Trig_{cq}$, $Term_{cq}$);
- the contents of each base relation after the last execution of the CQ;
- the differential relations for each of those operand relations that have been changed since the last execution of the CQ;
- the timestamp of the last execution of the CQ;
- the complete set of the result of the CQ produced by the last execution.

In short, the Differential Re-evaluation Algorithm (DRA) is developed for incrementally computing the new query result from processing updates on top of the previous result. We prove that our differential re-evaluation algorithm to continual queries is *functionally equivalent* to the "recompute the query from scratch" solution, and, in many situations is more efficient. For a formal description of differential (delta) relations and the DRA, refer to [8], where a number of implementation issues is also discussed, including asynchronous evaluation of CQ conditions and strategies for garbage collection of differential relations.

6 Incorporation of Push Data Sources

6.1 Overview of Data Delivery Modes Protocols

Data delivery is defined as the process of delivering information from a set of information sources (servers) to a set of information consumers (clients). There are several possible ways that servers and clients communicate for delivering information to clients, such as clients request and servers respond, servers publish what are available and clients subscribe to only the information of interest, or servers disseminate information by broadcast. Each way can be considered as a protocol between servers and clients, and has pros and cons for delivering data in an open and dynamic information universe.

Client Request and Server Response The *Request/Response* protocol follows the data delivery mechanism that clients send their request to servers to ask the information of their interest, servers respond to the requests of clients by delivering the information requested.

Current database servers and object repositories deliver data only to clients who explicitly request information from them. When a request is received at a server, the server locates or computes the information of interest and returns it to the client. The advantage of the *Request/Response* protocol is the high quality of data delivery since only the information that is explicitly requested by clients is delivered. In a system with a small number of servers and a very large number of clients, the *Request/Response* mechanism may be inadequate, because the server communication and data processing capacity must be divided among all of the clients. As the number of clients continuous to grow, servers may become overwhelmed and may respond with slow delivery or unexpected delay, or even refuse to accept additional connections.

6.2 Servers Publish and Clients Subscribe

The *Publish/Subscribe* protocol delivers information based on the principle that servers publish information online, and clients subscribe to the information of interest. Information delivery is primarily based on the selective subscription of clients to what is available at servers and the subsequent publishing from servers according to what is subscribed.

As the scale and rate of changes for online information continues to grow, the *Publish/Subscribe* mechanism attracts increasing popularity as a promising way of disseminating information over networks. Triggers and change notifications in active database systems bear some resemblance to the *Publish/Subscribe* protocol based on point-to-point communication [2]. The *Publish/Subscribe* mechanisms may not be beneficial when the interest of clients changes irregularly because in such situations clients may be continually interrupted to filter data that is not of interest to them. A typical example is the various online news groups. Another drawback is that publish/subscribe is mostly useful for delivering new or modified data to clients, but it cannot be used to efficiently deliver previously existing data to clients, which the clients later realize they need. Such data are most easily obtained through the request/respond protocol.

6.3 Servers Broadcast

The *Broadcast* mechanism delivers information to clients periodically. Clients who require access to a data item need to wait until the item appears. There are two typical types of broadcasting: *selective broadcast* (or so called *multicast*) and *random broadcast* [4]. Selective broadcast delivers data to a list of known clients and is typically implemented through a router that maintains the list of recipients. Random broadcast, on the other hand, sends information over a medium on which the set of clients who can listen is not known *a priori*. Note that the difference between selective broadcast and *Publish/Subscribe* is that the list of recipients in selective broadcast may change dynamically without explicit subscription from clients.

The *Broadcast* protocol allows multiple clients to receive the data sent by a data source. It is obvious that using broadcast is beneficial when multiple clients are interested in the same items. The tradeoffs of broadcast mechanisms depend upon the number of clients who have the commonality of interest and the volume of information that are of interest to a large number of clients [4, 2].

6.4 Summary of Data Delivery Modes

With the rapid growth of the volume and variety of information available online, combined with the constant increase of information consumers, it is no longer efficient to use a single mode of data delivery. A large-scale modern information system must provide adequate support for different modes of data delivery in order to effectively cope with the various types of communications between clients and servers to improve query responsiveness. Another benefit of providing different modes of data delivery is to allow the system to be optimized for various criteria according to different requirements of data delivery. In this section we identify three potentially popular modes of data delivery and compare them with the types of delivery protocols that can be used. They are client pull-only option, server push-only option, and client pull with server push combined option.

Pull-only Mode In the *Pull-only* mode of data delivery, the transfer of data from servers to clients is initiated by a client pull. When a client request is received at a server, the server responds to it by locating the requested information. The *Request/Respond* style of client and server communication is *pull-based.*

The main characteristic of pull-based delivery is that the arrival of new data items or updates to existing data items are carried out at a server without notification to clients unless clients explicitly poll the server. Also, in pull-based mode, servers must be interrupted continuously to deal with requests from clients. Furthermore, the information that clients can obtain from a server is limited to when and what clients know to ask for. Conventional database systems (including. relational and object-oriented database servers) and most of the web search engines offer primarily pull-based data delivery.

Push-only Mode In *Push-only* mode of data delivery, the transfer of data from servers to clients is initiated by a server push in the absence of specific request from clients. The main difficulty of push-based approach is to decide which data would be of common interest, and when to send them to clients (periodically, irregularly, or conditionally). Thus, the usefulness of server push depends heavily on the accuracy of a server to predict the needs of clients. *Broadcast* style of client and server communication is a typical *push-only* type.

In push-based mode, servers disseminate information to either an unbounded set of clients (random broadcast) who can listen to a medium or a selective set of clients (multicast) who belong to some categories of recipients that may receive the data. It is obvious that the push-based data delivery avoids the disadvantages identified for client-pull approaches such as unnoticed changes. A serious problem with push-only style, however, is the fact that in the absence of a client request the servers may not deliver the data of interest in a timely fashion. A practical solution to this problem is to allow the clients to provide a profile of their interests to the servers. The *Publish/Subscribe* protocol is one of the popular mechanisms for providing such profiles. Using publish/subscribe, clients (information consumers) subscribe to a subset of a given class of information by providing a set of expressions that describe the data of interest. These subscriptions form a profile. When new data items are created or existing ones are updated, the servers (information providers) publish the updated information to the subscribers whose profiles match the items.

Hybrid Mode The hybrid mode of data delivery combines the client-pull and server-push mechanisms. The continual query approach [12] presents one possible way of combining the pull and push modes, namely, the transfer of information from servers to clients is first initiated by a client pull and the subsequent transfer of updated information to clients is initiated by a server push.

The hybrid mode represented by continual queries approach can be seen as a specialization of push-only mode. The main difference between hybrid mode and push-only mode is the initiation of the first data delivery. More concretely, in a hybrid mode, clients receive the information that matches their profiles from servers continuously. In addition to new data items and updates, previously existing data that match the profile of a client who initially pull the server are delivered to the client immediately after the initial pull. However, in push-only mode, although new data and updates are delivered to clients with matching profiles, the delivery of previously existing data to clients that subsequently realize that they need it is much more difficult than through a client pull.

6.5 Pure Push versus Continual Queries

In a pure bush environment such as broadcast services, the server broadcast the update periodically and the clients may tune the channels to listen to those broadcast information that is of particular interest to them. Thus, the data is pushed from source to the broadcast server and then pushed from the server to the client. Figure 4 shows the typical data delivery flow in a pure push environment.

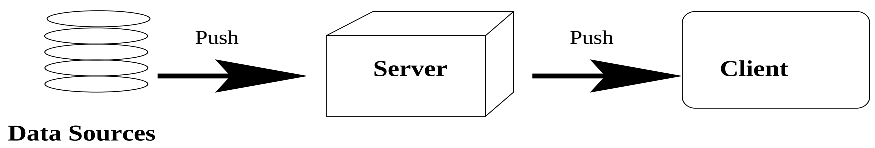

Fig. 4. The data delivery flow in a broadcast-based push service

Continual Queries server is not a pure-push based server since the request is initiated by a client pull. Once the client pulled the CQ server at the time of installing a CQ, the CQ server starts pushing the subsequent updates that satisfy the update threshold specified in the CQ to the client continually until the termination condition is met. The data delivery flow is shown in Fig. 5.

6.6 Incorporation of Broadcast-based Push in the CQ system

In the Continual Queries system architecture, we consider both pull-based data sources such as databases and web search engines and push-based data sources such as Pointcast, Marimba, BackWeb, AirMedia, Intermind. The idea for incorporating push sources into the CQ architecture is to provide a broadcast-based

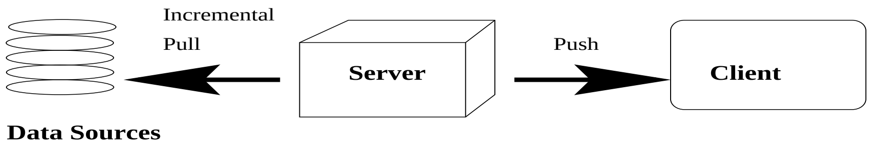

Fig. 5. The data delivery flow in a broadcast-based push service

push client agent on the CQ server for each push source. This push client agent listens to the broadcast lines on behave of a group of CQ clients, grabs the broadcast information of interest, and then responds to the CQ clients' queries by filtering out the irrelevant information. Figure 6 shows a sketch of the data delivery modes in a CQ system that incorporates the push sources in answering queries.

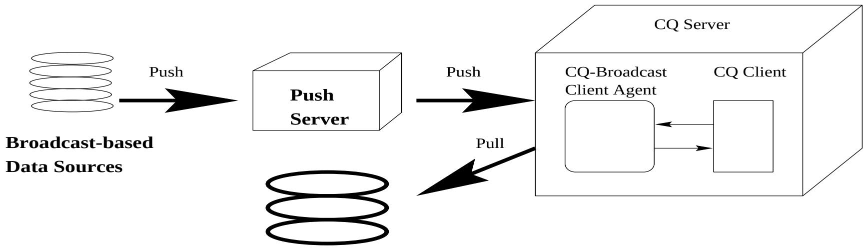

Fig. 6. The data delivery flow in a broadcast-based push service

7 Description of Demo

We demonstrate the latest version of our CQ robot, as described in the previous sections. Specifically we show how to use our CQ robots for monitoring updates in the following four different types of sources containing bibliographic data in the heterogeneous formats:

- A Oracle database which is remotely accessible through SQL, OraPERL, and SQLNet.
- A DB2 database which is remotely accessible through JDBC and SQL.
- A collection of UNIX files which are accessible through a Perl script or a Java Applet.
- A World Wide Web source which is accessible through our semi-structured information adapter and filter utility.

Although all four sources support different access methods, the wrappers hide all source specific details from the application/end-users. CQ users may pose a query on the fly, and install the query as a continual query by specifying the interested update threshold using the CQ trigger and specifying the continual duration using the CQ termination condition. We will demonstrate our query router technology and show how the multi-level progressive pruning improves the overall responsive time of queries as well as trigger evaluation. We also provide a testbed which consists of a user-friendly interface to allow users to experiment the updates at the data sources and watch the CQ system to compute the update threshold and evaluate the trigger, and alert or notify the user by email the new updates that match the query. We will also demonstrate the client and system administration services such as browsing and updating the installed continual queries, canceling some running continual queries upon request, and tracing the CQ trigger evaluation status and the update monitor status.

8 Conclusion

In this paper, we have described the problem of update monitoring in open environments such as the Internet. By update monitoring we mean the timely delivery of new information (updates) to users. This is a challenge research problem because of several factors. First, it combines update processing (detection) and query processing (new information filtering). Second, the detection and synchronization of updates in several heterogeneous data sources presents fresh problems of its own.

We also outline the Continual Queries (CQ) project as an example of research work being done in the update monitoring area. In the CQ project, a continual query is a combination of a normal query and a trigger condition. The query is evaluated when the trigger condition becomes true. The trigger condition may be time-based, e.g., every Monday at 8am, or content-based, i.e., a predicate on the database state such as "Microsoft stock going up by 10%". The query would be in SQL for relational databases, and keyword search for Web pages. A termination condition stops the query execution cycle.

We outline an architecture for the concrete implementation of continual queries in the CQ project. The architecture divides the problem into several components. On the client side, we have the GUIs for the specification of continual queries. These queries are translated into executable sub-queries and triggers by the CQ server. The CQ server consists of a system configuration repository (storing the names and capabilities of data sources), query translator and processing, and distributed trigger execution. The sub-queries are passed to data source wrappers, which translate and execute the local queries. The results are passed back to the CQ server, which assembles them for the user.

We are implementing the continual query capability for DARPA's Advanced Logistics Program. Although a relatively small step towards the lofty goal of generic update monitoring in open environments, the CQ project is making concrete progress. To try out our current demo, please point your Web browser to the following URL: `http://www.cse.ogi.edu/DISC/CQ/`. We welcome comments, bug reports, and feedback that will bring us closer to the ideal of getting the right information at the right time.

References

1. S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.
2. S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.
3. J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 61–71, Washington, DC, May 1986.
4. M. Franklin and S. Zdonik. Dissemination-based information systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(3):20–30, September 1996.
5. E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 1987.
6. B. Kahler and O. Risnes. Extending logging for database snapshot refresh. In *Proceedings of the International Conference on Very Large Data Bases*, pages 389–398, Brighton, England, September 1987.
7. B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 53–60, Washington, DC, May 1986.
8. L. Liu and C. Pu. The diom approach to large-scale interoperable information systems. Technical report, TR95-16, Department of Computing Science, University of Alberta, Edmonton, Alberta, March 1995.
9. L. Liu and C. Pu. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *DISTRIBUTED AND PARALLEL DATABASES: An International Journal*, 5(2), 1997.
10. L. Liu and C. Pu. Dynamic query processing in diom. *IEEE Bulletin on Data Engineering*, 20(3), September 1997.
11. L. Liu and C. Pu. A metadata approach to improving query responsiveness. In *Proceedings of the Second IEEE Metadata Conference*, Maryland, April 1997.
12. L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 27-30 1996.
13. N. Roussopoulos and H. Kang. Preliminary design of adms+: A workstation-mainframe integrated architecture fopr database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 355–364, Kyoto, Japan, August 1986.

This article was processed using the LaTeX macro package with LLNCS style