CQ: A Personalized Update Monitoring Toolkit [*]

Ling Liu, Calton Pu, Wei Tang[†] David Buttler, John Biggs,
Tong Zhou, Paul Benninghoff, Wei Han

Oregon Graduate Institute of Science and Technology
Department of Computer Science and Engineering
P.O.Box 91000 Portland Oregon 97291-1000 USA
{lingliu,calton,wtang,buttler,biggs,tzhou,benning,weihan}@cse.ogi.edu

1 Introduction

The World Wide Web (WWW) made an unprecedented amount and variety of data accessible to all Internet users. However, users suffer from information starvation under this data overload, due to the daunting challenge of navigating, collecting, evaluating, and processing data in this dynamic and open information universe. The problem is aggravated when information changes constantly, but unpredictably. For example, as more aspects of business and commerce migrate online, system-supported update monitoring and event-driven information delivery becomes increasingly important because it reduces the time users spend hunting for the updated information as well as unnecessary traffic on the net.

The CQ project at OGI, funded by DARPA, aims at developing a scalable toolkit and techniques for update monitoring and event-driven information delivery on the net. The main feature of the CQ project is a "personalized update monitoring" toolkit based on *continual queries* [6]. 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 new query matches are automatically returned to the user or the application that issued the query. 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 net, what they can do, and the changes happened to them. Whenever updates at the data sources result in new query matches or reach a specific update threshold, the CQ robot will compute and integrate the new results and present them to the user in a consolidated, easy-to-follow report. Comparing 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 technology by supporting personalized update monitoring through a combined client-pull and server-push paradigm.

[*]This research is partially supported by DARPA contract MDA972-97-1-0016, Intel, and Boeing.

2 Overview

The goal of the CQ project is to develop 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 quality and improve system scalability and query responsiveness. The second dimension is to build a working system that demonstrate our ideas, concepts, and techniques developed for continual queries using real-world application scenarios.

2.1 Continual Query Concept

A continual query is defined by a triple (Q, T_{cq}, `Stop`), consisting of a normal query Q (e.g., written in SQL), a trigger condition T_{cq}, and a termination condition `Stop`. The initial execution of a continual query is performed as soon as Q is issued and the whole result is returned to the user. The subsequent executions of Q are performed whenever the trigger condition T_{cq} becomes true. For each execution of Q, only the new query matches since the previous execution are returned to the user unless specified otherwise. We support 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 (such as execute Q every Monday or every first day of the month); and (3) at a time interval (such as execute Q every two weeks). For content-based event triggering, we support a variety of content-based condition. Examples include: a simple condition on the database state (e.g., execute Q whenever a deposit of \$10,000 is made), an aggregate condition on the database state (e.g., execute Q when a total of 1 million dollar of deposits have been made), and a relationship between a previous query result and the current database state (e.g., execute Q when a total of 1 million dollars of deposits have been made since the previous execution of Q). The `Stop` condition specifies the termination condition of a continual query. Both the trigger condition T_{cq} and the termination condition `Stop` will be evaluated prior to each subsequent execution of Q.

2.2 Continual Query Examples

Suppose we have a continual query request "*notify me in the next six months whenever IBM stock price rises by 5%*". This request is captured in the following three components:

- Query: Result(SC,SP) = `SELECT Stock.company, Stock.price`
  `FROM Stock WHERE Stock.company = 'IBM';`
- Trigger: `Stock.company = 'IBM'` .and. `Stock.price > SP*1.05`
- Stop: `six months from query creation`

Other examples of continual queries: "*tell me the flight number whenever a plane has been in this sector for more than 5 minutes*", "*place an order of 1000 units per warehouse whenever the average storage level of item A is below 200 units*", or "*At 5pm every day, notify me the total amount and classification of materials coming into or going out from these ports and their origin or destination*".

2.3 Continual Semantics

Let us denote the result of running query Q on database state S_i as $Q(S_i)$. We define the result of running a continual query CQ as a sequence of query answers $\{Q(S_1), Q(S_2), \ldots, Q(S_n)\}$ obtained by running query Q on the sequence of database states $S_i, 1 \leq i \leq n$, each time triggered by T_{CQ}, i.e., $\forall S_i,\ T_{CQ} \wedge \neg \texttt{Stop}$.

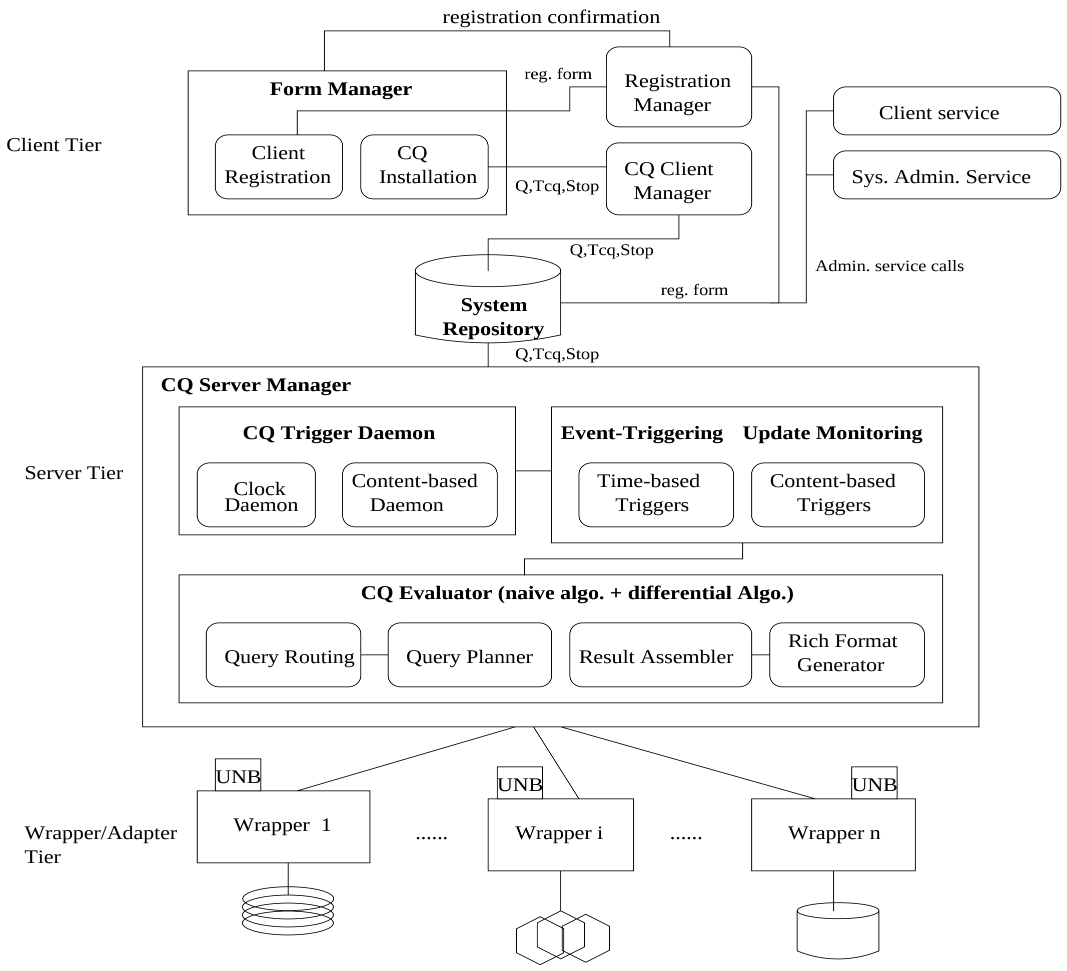

Figure 1: System architecture

2.4 System Architecture

As shown in Figure 1, the CQ system has a three-tier architecture: client, server, and wrapper/adapter. The client tier currently has four components: (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, T_{cq}, `Stop`), 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 IBM stock price rises by 5%*". 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 `Stop` condition for this request. The results can be returned to the user either by email, by fax, or bulletin posting.

The second tier is the CQ server which consists of three main components: (1) the update monitor with event-driven delivery, (2) the CQ-trigger-firing daemon, and (3) the continual query evaluator. The update monitor 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. The server also removes the continual queries when their stop conditions become true. The CQ trigger daemon is in charge of when to call the event-driven update monitor to evaluate the CQ trigger condition for each installed continual query. Two kinds of events are supported at this time: (1) a clock daemon checks specified date and time events, and (2) a content-based trigger-firing daemon checks specific update thresholds. The CQ query evaluator is responsible for processing the query Q when the trigger condition T_{cq} is true. It also provides a guard for the `Stop` condition of a query to guarantee the semantic consistency of the continual query (Q, T_{cq}, `Stop`). The key components of CQ evaluator include: the query router [5, 3], the query planner [2, 4] 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 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.

2.5 Client-Server Implementation

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 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.

3 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.

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] 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.

[3] L. Liu and C. Pu. Dynamic query processing in diom. *IEEE Bulletin on Data Engineering*, 20(3), September 1997.

[4] L. Liu and C. Pu. A dynamic query scheduling framework for distributed and evolving information systems. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Baltimore, May 27-30 1997.

[5] L. Liu and C. Pu. A metadata approach to improving query responsiveness. In *Proceedings of the Second IEEE Metadata Conference*, Maryland, April 1997.

[6] 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.