

Supporting Internet Applications Beyond Browsing: Trigger Processing and Change Notification

(Extended Abstract)

Ling Liu, Calton Pu and Wei Tang

Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280 USA
{lingliu, calton, wtang}@cc.gatech.edu

Abstract. This paper presents the design and implementation methodology of the JCQ system, a Java-based Continual Query system for update monitoring over Web information sources. A continual query is a standing query that monitors updates of interest using distributed triggers and notifies users whenever the updates reach specified thresholds. In this paper we focus on the strategies and techniques developed in JCQ for scalable and efficient trigger firing and the execution model for flexible and robust change notification. We evaluate our approach through a performance study of the most recent release of the JCQ system and a comparison with related work.

1 Introduction

The World Wide Web (the Web) publishes a vast amount of information that changes continuously. These rapid and often unpredictable changes to the information sources create a new problem: how to detect and represent the changes, and then notify users of the changes. Many application systems today have the need for tracking changes in multiple information sources on the web and notifying users of changes if some condition over the information sources is met. Example applications include military and civilian situation assessment, network management, business process control, program trading, to name a few.

We have developed a continual query system, which offers a system-supported update monitoring facility. A goal of the continual query system is to provide timely response (and alert) to critical situations, while reducing the effort and time users spend hunting for the updated information and avoiding unnecessary traffic on the net. The most recent version of the continual query system software is developed in pure Java. We refer to this version of the continual query system JCQ. One of the salient features of our JCQ system is its ability to provide scalable continual query services for active delivery of the desired information at the right time to the right user.

The main focus of this paper is to describe the strategies and mechanisms developed in JCQ to address the problem of scalable distributed trigger processing in the presence of large numbers of continual queries. It is well known that built-in trigger facilities in most of commercial RDBMS products are quite popular with application developers because it is a convenient mechanism for integrity constraint checking and update alerting across all applications of a database. Unfortunately, current trigger systems

in commercial database products have very limited scalability. Numerous commercial systems allow only one trigger per table to be defined for each of the three basic types of update events (Insert, Delete, and Modify). Our experience with the initial prototype of the continual query system [7] shows that many web applications could effectively use large numbers of triggers in individual information sources (for instance by installing large numbers of continual queries). Thus a new challenge for an Internet-scale update monitoring system (e.g., JCQ) is to provide mechanisms for *responsive* and *scalable* trigger processing in the presence of thousands or even millions of continual queries over the online information sources.

The approach we propose for implementing a highly scalable continual query system achieves scalability by exploring a number of types of parallelism at query-level, event-level, condition-level, and data-level and a sophisticated continual query index based on various trigger patterns. This approach gives good response time for continual queries, while supporting a large number of distributed triggers that would be expensive if processed naively. A key idea behind the JCQ approach is to classify all installed continual queries into a number of groups, according to the trigger structures (called primitive trigger patterns) they use. This idea is motivated by our observation that many of the installed continual queries use the same primitive trigger patterns, except for the appearance of different constant values in the trigger specification. An example of such coincidence is when a very large number of continual queries are installed over a single web (such as online retail bookstores, the stockmaster.com web source, or the national weather service web site `nws.noaa.org`). By grouping continual queries that have the same (or similar) trigger patterns together, the overall trigger processing cost can be dramatically reduced. This approach also scales well to the large numbers of concurrently running continual queries.

2 JCQ Overview

JCQ is built on a three-tier architecture: client, server, and wrapper/adaptor. This architecture was motivated by the need for providing scalable and reliable continual query processing, and the need for sharing information among structured, semi-structured, and unstructured remote data sources. The first tier is the client tier. The client manager coordinates client requests and the communication with the JCQ server. The second tier is the JCQ server. It is responsible for coordinating with the trigger condition evaluator and event detection manager to monitor updates of interest, and coordinate with JCQ wrappers and adaptors to track the new updates to the source data. The third tier is the JCQ wrappers/adaptors tier. A wrapper is a source-specific program that translates a server-tier query into a source-specific data fetch request. Once the source data is fetched and filtered, the wrapper returns the result or the location where the result is stored to the CQ manager. In addition to the common data wrapping capability, a JCQ wrapper installs a source-specific event detector that, on behalf of the JCQ server, continually watches the update events at the corresponding data source site(s), and signals the CQ manager whenever an update event of interest occurs. The implementation platform includes a Sun Ultra-sparc system running Solaris 2 operating system (the host environment), Java as the programming language (although there are still some Perl and native languages for downward compatibility considerations), and Java servlets (replacing CGI scripts) as the networking gateway programs for better performance and maintenance. All the client-side GUI interfaces are Web-accessible, which means using a standard Web browser, such as Netscape 3.0 or Internet Explorer

4.0 above, a user can interact with the JCQ system. We plan to develop two sets of GUI interfaces: simple interface is CGI-Javascript-HTML based; advanced interface is pure Java Applet+Servlet based. For people who only have slow network connections, simple interface is appropriate. The pure Java interface would respond better with faster Internet connections.

3 Scalable Trigger Processing

In JCQ there is no restriction on how many continual queries a user can create and how many users can register with the system. Quite often, the system faces the situation where thousands or even millions of continual queries are running concurrently (50 users with the average of 20 continual queries per user will reach a thousand). Thus, one of the main challenges that the JCQ system needs to address is the *scalability* of the continual query processing strategy, namely how to guarantee the responsiveness of a continual query system in the presence of large numbers of concurrently running continual queries.

Motivation. General speaking, a continual query (Q, T_{cq}, Stop) can be seen as a “long-running query” in the sense that it, once started, runs continually until its *Stop* condition becomes true. Typically, after the first run of the query component Q , the subsequent run of Q will be fired only if the trigger condition T_{cq} is evaluated to be true. Thus, most of the cost in running a continual query is spent on the continuous testing of its trigger condition T_{cq} . The bottleneck in a single test of the trigger condition T_{cq} is the time spent on the network connection to the remote site(s), the network bandwidth consumption in fetching the relevant page, and the server processing cost in detecting the newly updated information.

In addition, we have observed that, quite often among a large number of continual queries being installed, many of them are either against a single web information source or have a similar trigger structure (such as all monitor Intel stock price changes with different update thresholds). An obvious solution is to find smart indexing structure and indexing algorithms that can index all installed continual queries based on their trigger structure. As a result, the number of triggers that must be tested continuously against remote site(s) (web data sources) can be dramatically reduced.

Based on this observation, an obvious guideline in designing optimization strategies for continual queries is to look for opportunities throughout the process of a continual query, and design mechanisms that can reduce the server processing and network bandwidth consumption, while enhancing the scalability of distributed trigger processing. There are several alternative ways in which continual queries (CQs) can be processed. For example, a continual query can be processed independently, or be grouped with some other continual queries according to the data source(s) they monitor over or the trigger structures they share.

CQ Indexing Techniques. There are two motivations behind the indexing strategies for large number of continual queries. The first motivation is based on the general premise that a large number of triggers often share some of the predicate variables but often take different constant values in their trigger conditions. Thus by some canonical transformation, trigger patterns can be derived, and each pattern groups a set of change monitors sharing the same trigger expression template. The second motivation is to use the indexing techniques as both a facilitator for deriving polling queries that can serve for a group of continual queries, and a mechanism for speeding up the testing and execution of the CQ triggers (i.e., the filter query portions of the CQ triggers).

The first indexing technique is called *indexing by source*. It forms groups of CQs by indexing CQs on the data sources identified in the trigger expressions. For each group of CQs, a polling query is derived from the set of trigger expressions of the CQs. Since different data sources may offer different search capability, this indexing approach may not produce the most efficient polling plan, especially when more than one network connection is required for each group of CQs. For example, the weather.noaa.gov web site offers only search capability based on a single location. Therefore, to carry out a CQ that requests to monitor the weather condition at more than one location, say Portland and Seattle, two parallel polling requests to the weather.noaa.gov will be generated. Here is the basic idea: Given an indexing strategy, we first utilize the groups partitioned by index to construct one polling query per partition. Then we refine (optimize) the polling query expression by taking into account the query capability descriptions of the corresponding information sources and the parallelization capability at the JCQ server.

The second indexing technique is called *indexing by trigger pattern*. It forms groups of CQs by indexing CQs on both the source names listed in their trigger expressions and the trigger patterns derived from their trigger expressions. See [13] for more detail.

As many factors may affect the choice of each indexing strategy for distributed triggers, in the JCQ system we expect to produce guidance to update monitor implementors as to which strategies will be most effective for optimizing continual queries in which situations. This guidance will consist, for each application area, and for each access pattern of the Web data sources, of a description of the optimization process, the expected performance gains, and the performance metrics used for each of the alternative strategies. A key issue here is to investigate each potential alternative from a set of related and possibly conflicting factors that affect the optimization time, the complexity of coding, the quality of resulting plan, and the scalability of the method, and to understand the tradeoffs between the choices.

Implementation Strategy. The main components of the JCQ trigger manager are shown in Figure 1. The CQ trigger index manager coordinates the work among

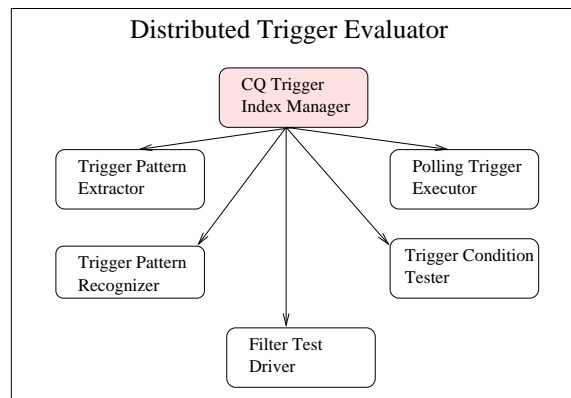


Fig. 1. Distributed trigger manager architecture

trigger extractor, trigger pattern recognizer, polling trigger executor, and trigger condition tester. Trigger pattern extractor transforms the trigger condition into a canonical representation that can be recognized by the trigger pattern recognizer. If the trigger pattern is new, a new trigger pattern will be created. Otherwise, the CQ is indexed into one of the existing pools of CQs according to its trigger structure. The polling trigger executor is responsible for multi-threaded execution of polling trigger queries. Each polling trigger corresponds to a trigger pattern. The triggers of the similar pattern is grouped together though the continual query index on trigger pattern. Once the polling trigger returns the result of a polling trigger query, the multithreaded trigger condition tester is invoked to evaluate the rest of the CQ triggers of the same pattern concurrently. Each thread evaluates the filter portion of a trigger expression T_{cq} .

Due to the space limitation, readers may refer to [13] for further details.

4 Event Notification in JCQ

Event notification is a software facility that provides mechanisms for notification of event occurrences. Generally speaking, an *Event Notification Service* (ENS) protocol has minimum and maximum latency bounds on both event observation and update notification. The notification can be provided by either a server-initiated push or a client initiated pull delivery. The notification service may enforce application-specific delivery constraints, such as delivery based on user priority, access control and security constraints.

In the JCQ system, events may be defined in terms of time dimension (e.g., every 10 minutes or 10am everyday) or in terms of physical or information space (e.g., a hand clap, a light being switched on, or IBM stock price dropped by 10%). Events are either primitive or composite. The JCQ event notification facility is utilized in at least three ways: first, we use the event notification facility to notify users of their continual query service subscription, including both the subscription for JCQ system service, the expiration of their installed continual queries, and the expiration of their JCQ subscription. Second, we use the event notification facility to send the users the update alert and the differential results of the data items being monitored. Third, the JCQ event notification facility is designed to allow an external event observer to be loosely coupled with the change notification. Such coupling enhances both the tracking capabilities as well as the update monitoring functionality of JCQ. It also offers JCQ users with services for alerting and notifying users of the information changes of interest, and the capability of taking appropriate actions to react to the changes.

Consider a complex update monitoring request “*start monitoring the stock price updates of IBM, Intel, Microsoft, and Dell for three days when NASDAQ reaches 2000, and report to me whenever the stock price of any of these companies drops by 5%*”. This monitoring request can be easily modeled by two continual queries:

```

create CQ_A as
  Query:   none
  Trigger: When NSASDAQ reaches 2000
  Notify:  Install CQ_B
  Stop:    a month (by default)

Create CQ_B as
  Query:   stock price changes on IBM, Intel, Microsoft, Dell

```

Trigger: when stock price of IBM or Intel or MSFT or Dell drops by 5%
 Notify: wtang@cse.ogi.edu
 Stop: after 3 days

In the current implementation of JCQ, we require the users to explicitly register the event observation function to be called and the method of remote invocation when a notification service is subscribed. In the rest of this section, we discuss the research and engineering issues involved in the design and implementation of the JCQ event notification service. For example, what types of events should be considered at the notification phase, when to notify users of the changes, how to guarantee in-time delivery of the notification, and what mechanism to use to deliver the notification.

Notification Service Architecture. The design goal of the notification service is to make it reusable and configurable, so it can easily be incorporated into other system architectures. The first design choice is to decide whether to build the notification service as a standalone one or as a component of JCQ.

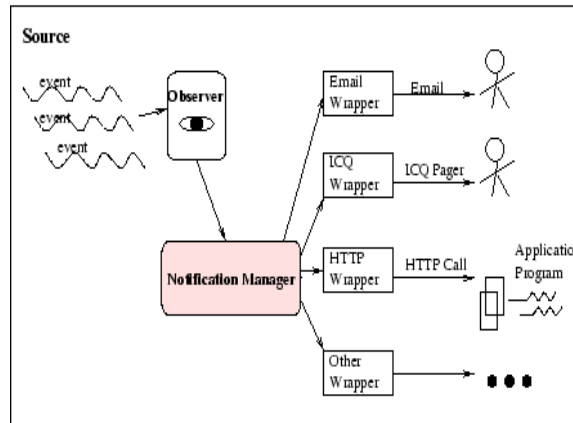


Fig. 2. Standalone notification service

In the standalone architecture (Figure 2), the main components are: source, observer, notification manager, and wrapper set. The wrapper set is bundled with the notification service. This architecture is considered “loosely coupled” with the source event observer. And the observer could be provided by or combined with the source and connect to the notification manager through HTTP connection.

The notification service can also be designed as a component of a system. In this case, the notification service may cooperate with other system components in a tightly coupled fashion. The source observer is usually provided by the software system using a set of source wrappers. For example, in JCQ we choose to implement the notification service as a component of JCQ so that the trigger facility can be reused to implement richer set of notification conditions such as “notify Todd whenever CQ trigger condition becomes true twice consecutively”. The JCQ notification service can reside on the same server as the JCQ engine or reside on a machine connected to the JCQ server within a

local area network. Compared with the “loosely coupled” notification service, a “tightly coupled” one has the advantage of less network communication overhead, but it loses some flexibility and harder to be implemented generically. Due to the space limitation, readers may refer to [13] for further details.

Discussion. Latency is an important factor in the JCQ notification service. The *source observer* (whether it is source-initiated or polling-based) sets the sampling latency, bounding the delay from source event occurrence until observation. We refer to such delay as the *observation interval* or the *sampling latency*. This “observation interval” determines the minimum resolution of event frequency. The notification manager controls the latency from observation until notification. We refer to such latency as the *delivery latency* or *delivery interval*, which determines the maximum resolution of notification frequency.

Achieving low sampling latency (less than delivery latency) may require source-initiated observation. Hard-to-decide polling intervals often make polling-based event observation inefficient. But in case of unavailability of source-initiation, the polling approach is the way to go.

In determining which side (source or notification server) should initiate notification delivery, one needs to know which party has enough information on its counterparts. If an event source knows its listeners, the source can initiate notification delivery upon event observation. For example, a printer can send a message to the owner of the print job upon print completion (in Unix, this is achieved by simply using “-m” option for *lpr*). However, if an unknown set of users want to monitor a single source (e.g., the current job queue of some printer), a client-polling policy is more appropriate. For example, Unix users can use “lpq” to list a printer job queue.

Event notifications can be exchanged directly between sources and end-receivers, or through an intermediate party (server) using “end-to-end” delivery or “store-and-forward”. In a very large scaled environment (such as the Internet), the use of intermediation parties (or proxies) is common.

In terms of delivery constraints, the *Event Notification Service* (ENS) is responsible for ensuring certain guarantees before delivering a notification to the recipients. For example, whether it supports real time delivery or not, whether to retry in case of message lost. Currently, we use “no retry” policy in the JCQ system for simplicity. The JCQ system can support up-to-minute delivery currently. Performance improvements on reducing both the sampling and the delivery latency are under way. In addition, security constraint is also an important responsibility of the ENS. For example, in case of a firewall, the ENS need to have the knowledge of proxies to be used to pass the firewall. To ensure correct notification, the notification manager maintains a subscription list, which could be an editable and audible “first-class” object. An initial verification is performed at the time of a notification service subscription. For example, in JCQ system, the email addresses should be checked upon user registration in order to guarantee that the correct email addresses are used for the users to receive notification. We are interested in incorporating more advanced quality of service (QoS) properties into the event notification service of the continual query system.

5 Performance Measurement and Evaluation

We have reported our strategies for scalable trigger processing and change notification facilities in JCQ. An immediate question one would ask is what is the performance characteristics of the CQ system and how well does it scale? In this section, we

report our initial experiments conducted on OpenCQ, an earlier version of the Continual Query system, developed using Perl CGI and Java. This is partly because the JCQ system has not fully completed the testing phase at the time of the writing and partly because we believe these initial measurements are interesting and useful for understanding the Continual Query system in general and the implementation strategies adopted by JCQ. It is important to note that we expect JCQ to perform much better than OpenCQ in all aspects. This is primarily because JCQ is a multithreaded system with in-depth exploration of parallel processing.

The parameters and factors we use in the performance study are shown below.

Parameters & Factors	Values
Network bandwidth (Kbps)	100Mbps/sec (local), 2Mbps/sec (remote)
Source update frequency (times/minute)	1 for stock source
Server average CPU load	dynamic factor
Query size	from 8KB to 200KB
Trigger size	from a few bytes to less than 10KB
Max number of active CQs	25
Retry interval	there is no retry after a failure
Minimum time interval for time-based CQ	10 minutes
Polling interval for content-based CQ	5 minutes

The following measures are selected for the initial performance evaluation.

1. *Execution time*: including trigger time, query time, notification time and total execution time.
2. *Average CPU load (in # of jobs)*: on Unix, using “uptime”.
3. *I/O cost*: including database access and file operations (reads and writes).
4. *Workload characterization*: including trigger size distribution, query size distribution, and locality of interests.

A CQ status history log (March 9 - April 30, 1999) is shown in Figure 3. Figure 4 shows the trigger evaluation time, the query processing time, the notification time in comparison with the total roundtrip execution time for a continual query. The trigger time and the query time both include cache creation or refresh time and differential result generation time. From the set of CQs we run for this performance study, we observe that the curve of CQ execution time matches the curve of the CPU load of the server machine. The CPU load was measured using the CPU uptime at the beginning of a CQ evaluation and the CPU uptime at the end of the evaluation.

There are many factors that affect the system performance and CQ execution time. For example, *database access time* is a known performance factor. By grouping database accesses into batch jobs, we can efficiently reduce the CQ evaluation time. It also helps if we keep the database connection open (persistent) until the end of CQ evaluation. Another factor is *CGI vs. Java servlets*. CGI is process-based, while Java servlets are thread-based. By replacing the CGI calls to servlet calls, we can save valuable system resources. Furthermore, *cache management* is also a critical factor during a normal CQ evaluation. We are working on implementing more efficient cache structure to speedup CQ evaluation. A promising solution is to employ the *Differential Evaluation Algorithm* [5] for version control and cache management. In addition, strategies for handling *server failures due to timeout* are important. Timeout may be caused by either slow network connections or host failure. In either case, we may consider the host is unavailable. To find out as early as possible the host unavailability can certainly improve system performance by reducing idle waiting time. Our performance

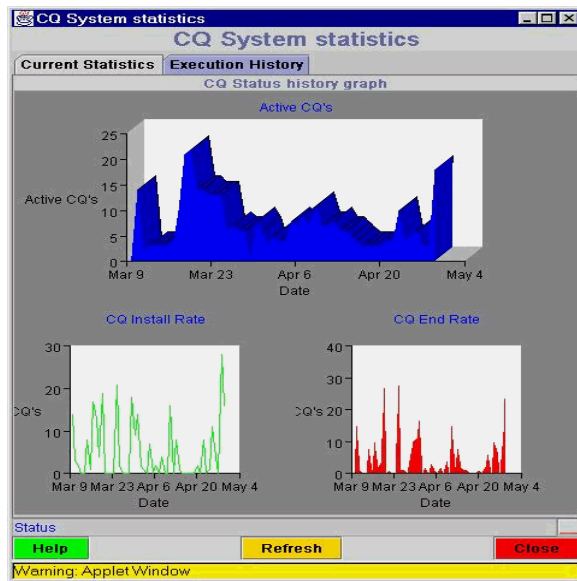


Fig. 3. CQ system status history screenshot

evaluation also considers the *logging overhead*. To provide higher availability and reliability of the JCQ system, we adopt the “write-ahead-logging” protocol in the CQ system. Before any changes are written to stable storage, a log is written to stable storage first. However, logging overhead is not negligible. One possible workaround is to use separate resource manager as the logging manager and/or divide the application into groups of small “transactions” and exploit maximum parallelism. Finally, exploring asynchronous execution as much as possible will also help reducing the overall idle waiting time of the system.

6 Related Work and Concluding Remarks

The paper describes the design and implementation of a prototype system for *data update monitoring* in a distributed open environment (such as the Internet). It uses the *continual query* concept as a powerful means for supporting continual monitoring of information updates. The work presented here is built on top of previous work in *Continual Queries* [5] as well as in DIOM [6]. One of the main design goals of JCQ is the scalability of distributed trigger processing for a large number of continual queries. We explore a number of parallel processing opportunities at the query level, the event level, the condition level, and the data level. The main idea is to group continual queries according to their trigger structures (commonality in the predicates specifying the triggers). Since the variants of the same structure can be processed together at low cost, successful grouping leads to highly scalable trigger and query processing.

There has been considerable research done on data update monitoring in databases. Powerful database techniques such as active databases and materialized views have

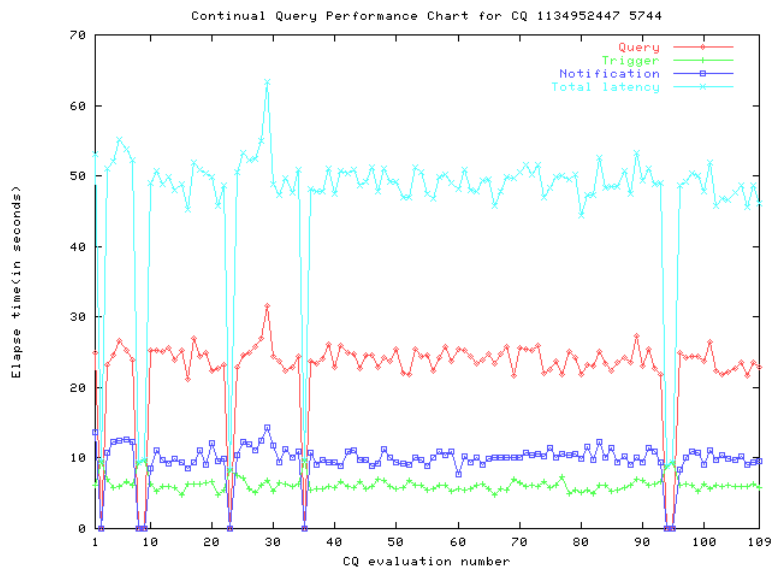


Fig. 4. CQ execution times

been studied extensively. These techniques have been proposed primarily for “data-centric” environments with a close-world assumption, where data is well organized and centrally controlled in a database. When applied to an open information universe as the Internet, these assumptions no longer hold, and some of the techniques do not easily extend to scale up to the distributed interoperable environment. Comparing with the state-of-art of research in active databases [12, 9, 10, 4, 11, 3], the JCQ system differs primarily in the following three ways: First, the JCQ system targets at update monitoring on the Web, handling both structured database sources and semi-structured sources such as HTML files. Second, the continual query concept can be seen as a practical and useful simplification of the ECA rules. In continual queries, user-defined actions are queries only and actions defined by the systems are side-effect free operations such as email notification and differential result display. This simplification made an efficient implementation practically feasible. Third, the JCQ system provides efficient and scalable trigger processing to handle large numbers of concurrently running continual queries on a variety of data sources. There are several systems developed towards monitoring Web source data changes. A main problem with these systems is that they are tailored to a particular domain of data source or application such as news in SIFT [14] or pre-wrapped sources such as [2], lacking of the generality and extensibility of the JCQ system.

References

1. E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings of ACM SIGMOD Conference*, 1993.
2. S. Chawathe, S. Abiteboul, and J. Widom. Managing and querying changes in semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.
3. L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 377–388, March 1990.
4. E. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of ACM SIGMOD Conference*, 1992.
5. 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.
6. L. Liu, C. Pu, and Y. Lee. Adaptive approach to query mediation across heterogeneous information sources. In *International Conference on Cooperative Information Systems (CoopIS)*, Brussels, Belgium, June 1996.
7. L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.
8. L. Liu, C. Pu, W. Tang, and W. Han. CONQUER: A Continual Query System for Update Monitoring in the WWW. *International Journal of Computer Systems, Science and Engineering*, 1999. Special Issue on WWW Semantics, edited by Dan Suciu and Letizia Tanca.
9. D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 215–224, May 1989.
10. U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings of the International Conference on Very Large Data Bases*, pages 469–478, Barcelona, Spain, September 1991.
11. M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of 1985 SIGMOD International Conference on Management of Data*, pages 374–387. ACM/SIGMOD, 1985.
12. J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann, 1996.
13. Ling Liu, Calton Pu and W. Tang. Supporting Internet Applications Beyond Browsing: Trigger Processing and Change Notification *Manuscript*, Oregon Graduate Institute, Department of CSE, April 1999.
14. T. W. Yan and H. Garcia-Molina. SIFT - a tool for wide area information dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177-186, 1995.