CONQUER : A Continual Query System for Update Monitoring in the WWW *

Ling Liu, Calton Pu, Wei Tang, 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,weihan}@cse.ogi.edu

Abstract

The World Wide Web (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. Commonly used search engines (e.g., AltaVista) and directory services (e.g., Yahoo) have practical but limited success. The exponential growth of the Web is increasing the haystack rapidly. Instead of pull-based browsing, *update monitoring* is a promising area of research where the system brings the right information to the right user at the right time. In this paper we present the design and implementation of the CONQUER continual query system, designed for update monitoring over the Web information sources. A Continual Query (CQ) is a standing query that monitors update of interest using distributed triggers and notifies the user of changes whenever an update of interest reaches specified thresholds or some time limit is reached. In contrast to normal queries whose scope is limited to past and present data, the scope of a continual query also includes future data. The first contribution of the CONQUER system is the specification language and the inherent semantics of continual queries. The second contribution consist of the mechanisms for efficient and scalable processing of large numbers of continual queries. The third contribution is the three-tier architecture that provides active capabilities at both mediator tier and wrapper tier and interoperability among multiple web information sources.

Keywords: Update Monitoring, World Wide Web, Continual Queries, Distributed Triggers, Distributed Event Monitoring.

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

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 the users. 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.

Despite their practical value, most Web search engines (e.g., AltaVista, Infoseek, Hotbot, Excite) are limited by their primarily pull-based, passive information delivery nature. Queries are executed only when explicitly requested by a user or application program and each transfer of data from servers to clients is initiated by an explicit client pull. Consequently, applications requiring timely response to critical situations are poorly served by these passive pull-based data management systems. For those time-constrained applications, it is important to monitor events of changes occurring in the information sources dynamically, and provide notification service whenever the updates reach some specified thresholds or time limits. For example, inventory control in an automated factory web site requires that the quantity on hand of each item be monitored. If the quantity on hand falls below a specified threshold for some item, then a notification for replenishment should be initiated either immediately or at the end of the working day.

Based on these observations, we have developed a continual query system, called CONQUER, which implements a system-supported update monitoring facility. CONQUER provides 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. One of the salient features of the CONQUER system is its ability to provide scalable continual query services for the active delivery of the desired information at the specified time to the user.

The concept of continual queries was introduced [18] as an effective means for update monitoring in structured data sources such as relational and object-oriented databases. In this paper we focus on the formal semantics and the mechanisms for scalable processing of continual queries over semi-structured web information sources. The semantics of continual queries is divided into three parts: (1) the semantics of queries over semi-structured information sources, (2) the semantics of distributed event and trigger specification, and (3) the semantics of validity and determinism of continual queries. This is the first contribution of the paper and the topic discussed in Section 2.

The second contribution of the paper consists of the strategies and mechanisms developed 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 CONQUER system [20] 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., CONQUER) is to provide mechanisms for responsive and scalable trigger processing in the presence of thousands or even millions of continual queries over the web information sources.

The approach we propose for implementing a highly scalable continual query system achieves scalability by exploring a number of types of parallelisms 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 CONQUER 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. The coincidence occurs when a very large number of continual queries are installed over a single web source (such as the national weather service site `nws.noaa.org`) or a collection of information sources of the same domain (such as online retail bookstores or online car sales). 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. In Section 3 we describe the CONQUER system architecture. In Section 4 we describe the distributed scalable trigger processing techniques currently developed in the CONQUER system.

Closely related to the responsive and scalable continual query processing is the problem of change detection over the semi-structured web information sources where no built-in trigger facility is available and data changes autonomously. We believe that solutions relying on update notification by each individual source to a centralized system are unrealistic for many Internet information sources. In Section 5 we discuss briefly the strategies being developed in CONQUER for change detection over semi-structured web sources, but omit detailed algorithms due to the space limitation. Then, we illustrate the first prototype implementation of the CONQUER system by a walk through example, and conclude the paper with an overview of related work, a summary, and an outline of directions for future work.

2 The Continual Query Specification Language

2.1 Semantics

We define a continual query as a quadruple (Q, T_{cq}, `Start`, `Stop`), consisting of a normal query Q (e.g., a SQL-like query or a keyword-based query), a trigger condition T_{cq}, a start condition `Start`, and a termination condition `Stop`. T_{cq}, `Start`, and `Stop`, in general, may depend on many different parameters. In the sequel, we omit their parameters for clarity. In contrast to *ad hoc* queries in conventional DBMSs or web search engines or query systems, a continual query, once activated (i.e., installed and started), runs continually over the set of information sources. Whenever the trigger condition T_{cq} becomes true, the new result will be returned. The trigger part of a continual query specifies the events and situations to be monitored.

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 data source states $S_i, 1 \leq i \leq n$. $Q(S_1)$ starts when the `Start` condition is true. The subsequent execution of $Q(S_i)$, at each given state S_i $(i > 0)$, is triggered by $T_{cq} \wedge \neg$`Stop`.

The initial execution of a continual query is performed as soon as its `Start` condition is verified. The first run of its query component Q is performed over past and present data represented by the state

of information sources, and the integrated result obtained by executing Q is returned to the user. The subsequent executions of Q are performed whenever a new update event is *signaled* and the trigger condition T_{cq} becomes true. For each subsequent execution of Q, only the new query matches since the previous execution are returned to the user unless specified otherwise. Thus the domain of continual queries is defined over past, present, and future data, whereas the domain of pull queries is limited to past and present data.

In CONQUER we support two types of events: *time events*, which involve clock times, dates, and time intervals; and *object events*, which involve changes to non-temporal objects. Accordingly, we distinguish three types of trigger conditions: time-based trigger condition, which consists of only time events; content-based trigger condition, which consists of only object events; and the hybrid trigger condition, which consists of any combinations of time events and object events. Three types of temporal events are supported for *time-based* or *hybrid* trigger condition:

- (1) absolute points in time, defined by the system clock (e.g., 7:30:00 pm., March 30, 1998);
- (2) regular time interval (e.g., execute Q every Monday or every two weeks) or irregular time interval (e.g., execute Q the first day of every month);
- (3) relative temporal event (e.g., 50 seconds after event A occurred).

A *content-based* trigger condition can be defined in terms of a database query, a built-in situation assessment function (e.g., report to me whenever the inventory level of any item changes), a user-defined method (e.g., alert whenever the price of IBM stock drops by 10%), or an application-generated signals (e.g., a failure signal from a diagnostic routine on a hardware sensor at the water temperature forecast station). Furthermore, the trigger conditions to be monitored may be complex and may be defined

- on sets of distributed data objects (e.g., the total of pending legal cases exceeds the given threshold),
- on transitions between states (e.g., the new position of the ship is closer to the destination than the old position),
- on trends and historical data (e.g., e.g., the output of the sensor increased monotonically over the last two hours), or
- on a relationship between a previous query result and the current database state (e.g., send me the temperature maps and time series whenever the water temperature at Tansy Pt. changes 50% since the last reporting time).

Furthermore, both the `Start` condition and the `Stop` conditions can be specified in terms of time events or object events. Both the trigger condition T_{cq} and the termination condition `Stop` are evaluated prior to each subsequent execution of the query component Q. In the current prototype of CONQUER, we restrict the `Start` condition and `Stop` conditions to be time events to simplify the implementation effort, since most frequently used start and stop conditions are time events.

2.2 Syntax

Commands in CONQUER have a keyword delimited, SQL-like syntax. The following commands are used to specify a continual query. Those commands that are marked by a square bracket are optional.

```
<continualQuerySpec>::
    create CQ [<continualQueryName>] as
    Query: <queryExpression>
    Trigger: <triggerExpression>
    [Start: <timePoint>]
    [Calendar: <calendaExpression>]
    Stop:  <timePoint>
    [NotifyCondition: <timeIntervalExpression>]
    [NotifyMethod:] <methodType><methodSignature>
```

Syntactically, each continual query has at least a query component, a trigger component, and a `Stop` condition. Users may give each of their continual queries a meaningful name (such as `Savannah_weather_watch` in Example 1, Section 2.3). The query expression of a continual query is specified in the SQL-like `SELECT-FROM-WHERE` clauses. The trigger of a continual query defines the situations (i.e., the events or conditions) to be monitored. In CONQUER we distinguish three classes of events. The first class is called *update events* and includes both the basic update operations such as INSERT, DELETE, UPDATE and the user-defined routines. The second class is referred to as *conditional events*. It is further classified into atomic conditional events such as `Stock.price INCBYPERC 5% WHERE symbol = 'IBM'` and composite conditional events such as `Stock.price INCBYPERC 5% AND Stock.price DECBYPERC 5% WHERE symbol = 'IBM'`. The third class is called composite synchronization events such as `E1` precedes `E2`, denoted by `E1; E2`, or `E1` and `E2` occur at the same time, denoted by `E1` || `E2`. We provide a number of language constructs to express trigger conditions. The syntax of the trigger expression is defined as follows. Due to the space limitation, we omit the complete syntax of the CQ specification language in this paper. Readers who are interested in more details may refer to [27].

```
<triggerExpression>::= <timeEventExpression> | <contentTriggerExpression>
<contentTriggerExpression>::=
                      from <fromList>
                      [on <eventSpec>]
                      [when <conditionExpression>]
<fromList>::= [<sourceName>.]<ObjClassName> [<objectVariable>] | ,<fromList>
<eventSpec>::= <timeEvent> where <contextCondition> | <objectEvent> where <contextCondition>
<objectEvent>::= INSERT | DELETE | UPDATE | <userDefinedEvent> | <compositeSynEvent>
<conditionExpression>::= <triggerCond> | <triggerCond><logicOp><conditionExpression>
<triggerCond>::= <atomCond> [where <contextCondition>]
<atomCond>::= <fieldName><comparisonOp><constant>
<comparisonOp>::=<algorithmicOp> | <stringOp>
<contextCondition>::= <atomCond> | <atomCond><logicOp><contextCondition>
```

Consider a continual query "notify me whenever the stock price of IBM or Intel drops 5%". The query component can be described as

```
SELECT symbol, price
FROM Stock;
```

The trigger component can be described by the following trigger expression

```
FROM Stock
WHEN price DECBYPERC 5
     WHERE symbol like 'IBM' OR symbol like 'MSFT';
```

By taking a closer look at the trigger condition of this example, one may observe that what we are interested in monitoring is the stock price change with the update threshold 5% not the change of company symbols. The condition in the `WHERE` clause specifies the context of the situation to be monitored (i.e., the company of interest is IBM or Intel). Hence the field `Stock.price` should be guarded for change notification but the field `Stock.symbol` should be only used for constraining the search. In CONQUER we explicitly distinguish the conditional events to be monitored, such as the stock price drops by 5%, from the context condition used to constrain the context of the conditional events of interest, such as the `WHERE` clause `symbol like 'IBM' OR symbol like 'MSFT'` in this example.

The `Start` and `Stop` clauses define a time interval within which the continual query is eligible to run. The `Start` clause is optional and the default `Start` time point is the time when the continual query is installed. In addition, we design the optional clause `Calendar` to specify "on" and "off" time periods for the trigger of a continual query. The main motivation for allowing a Calendar specification for each continual query is to reduce the amount of unnecessary pulling when it is known prior that a specific data source will not change for certain periods of time. Recall the above continual query, which monitors IBM and Intel stock prices. We may specify the "on" periods for this continual query to be Monday through Friday from 8:00am to 5:00pm since most of the stock markets are closed during evenings and weekends. A trigger is eligible to be fired only if (1) the continual query to which it belongs is installed and activated, (2) the current time is between the `Start` time and the `Stop` time, and (3) if it has a calendar specified, then the current time should also be in an "on" period (see Example 2 in Section 2.3).

The optional clause `notifyCondition` is designed to support the situations where the desired notification condition is different from the trigger condition. For example, if we have a time trigger: `every 10 minutes`, but we wish to be notified for stock price updates every 2 hours, then we can use the `notifyCondition` to specify the notification interval `two hours`. Thus, every notification will provide a summary of 12 CQ query results, each of which is collected at a ten-minute interval during each notification period. However, for a content-based trigger, every notification will provide a summary of the number of differential query results, each of which is collected at a successful testing of the trigger condition. Those trigger tests that fail the trigger condition specification will not result in computing the differential query. No change notification is generated. The default notification condition is set to be the same as the trigger condition. The optional clause `NotifyMethod` defines the set of notification methods to be supported in the system. This clause is especially useful for the situations where the notification of a change is not supposed to send to a user by email but send to an application program to trigger an action.

The notification model of the CONQUER system is beyond the scope of this paper. Interested readers may refer to [19] for some further detail.

2.3 Examples

We below provide three examples of continual queries written in a SQL-like expression enhanced with user-defined or system built-in functions. The first one uses a time-based trigger, the second one uses a content-based trigger, and the third one uses a hybrid trigger.

Example 1 We first define a continual query `weather_watch` that monitors weather condition changes in the region from port of Savannah in Georgia to Fort Stewart Military Reservation every 10 minutes and send mail to Todd every two hours using the function `send_mail` whenever the specified update event on weather condition is detected. Suppose that this continual query is defined over a Semi - structured data source – the national weather services center (NWS) web site (`www.nws.noaa.gov`), and the continual query name is specified in the `Create CQ` clause. The trigger condition is specified in the `Trigger` clause, the termination condition is specified in the `Stop` clause, and the query component is specified in the `Query` clause. Here is the specification of this continual query:

```
Create CQ Savannah_weather_watch as
Query:   SELECT *
         FROM weather_source@www.wns.noaa.gov
         WHERE location like 'Savannah' AND state  'Georgia';
               OR location like 'Fort Stewart';
Trigger: every 10 minutes;
Stop:    1 year (default);
```

This continual query specifies the request for monitoring updates on weather conditions in the region from port of Savannah to Fort Stewart every 20 minutes, and detects the update on weather condition at this region using a temporal event detector. Whenever an update event is signaled, the system takes the action of notifying Todd by email and delivering the updated result using a specific web URL pointer.

Note that the action of displaying the updates of weather condition at the specified Savannah region, and the action of reporting to Todd by mail is implicitly inferred by the system, based on the fact that Todd is the owner (creator) of this continual query `Savannah_weather_watch`. We can always use the `NotifyMethod` clause to add more receivers for this continual query and/or to define a notification interval that is longer (say every two hours) than the trigger condition (e.g., every 10 minutes).

Interesting to note is that the trigger condition and the query component in a continual query both can be specified in SQL -like expressions. When the trigger condition is defined over the same set of objects as the query component, the FROM clause may be omitted. Here is an example.

Example 2 Consider the continual query, "notify me in the next two weeks whenever the stock price of Bayer drops by 5%". By using the system-defined operator `DECBYPERC`, this continual query can be expressed conveniently as follows:

```
Create CQ Bayer_Stock_watch as
Query:   SELECT company_symbol, stock_price, hi_last_wk, lo_last_wk
         FROM   Stock
         WHERE  company_name  'Bayer AG';
Trigger: WHEN DECBYPERC(stock_price) >= 5% WHERE company_name = 'Bayer AG';
Start:    9:00:00 am, Nov. 1, 1998;
Stop:     9:00:00 am, Oct. 26, 1999;
Calendar: Monday through Friday, 8:00:00 am through 5:00:00 pm;
```

Note that the trigger condition of this example CQ is content-based. Instead of testing triggers at a user-specified time interval as done in time-based CQs, a system-controlled polling interval (say time units T) is set up. Once a content-based CQ is started, the trigger condition is tested at every T interval. The differential query result, however, will only be computed at the times when the trigger passes the test (e.g., satisfies the trigger condition that "`Bayer`'s stock price has droped 5% since last notification" in this example).

This example demonstrates another interesting feature of CONQUER, which is to allow users to specify their trigger conditions using system built-in functions in addition to the common string comparison operators such as `CONTAINS`, `LIKE`, and algorithmic operators $<, \leq, >, \geq, =, \neq$. For example, the current CONQUER prototype has a number of system built-in functions for trigger specification, including increased by x percent, denoted as `INCBYPERC(A)` $\geq x$, and decreased by y percent, denoted by `DECBYPERC(B)` $\geq y$, where A and B are field names of the source data items.

In CONQUER it is also possible to specify a continual query with its query component defined over one set of data sources and its trigger condition defined over another set of data sources. Here is an example where the trigger condition is defined the noaa weather web site and the query component is defined against the transportation Oracle database.

Example 3 Consider the following continual query that "monitoring the weather condition between port of Savannah and Fort Stewart every 10 minutes in the next 3 months, provide me with a list of alternative plans whenever the weather condition changes in the region between Port of Savannah and Fort Stewart reservation".

```
Create CQ Savannah_weather_watch as
Query:
   SELECT plan_no, plan_desc., plan_alt_routes
   FROM   Transportation_plan
   WHERE  plan_route like 'Savannah to Fort Stewart';
Trigger:
   FROM  Weather@www.nws.noaa.gov
   ON EVENT: UPDATE AND every 10 minutes
   WHERE location like 'Savannah' AND state = 'Georgia'
   OR location like 'Fort Stewart';
Stop: next 3 months.
```

Note that if we would like to be notified every 10 minutes or when weather changes, then the trigger condition should be specified as `ON EVENT: UPDATE OR every 10 minutes`. Note also that the query component of this CQ is against the `Transportation_plan` stored in ORACLE database, a structured data source; and, however, the trigger is defined over the weather information source available from the NWS web site, a semi-structured data source.

Generally speaking, in specifying a continual query, the `Query` clause, `Trigger` condition clause, and `Stop` condition clause are essential and thus mandatory. In addition, we provide a set of optional clauses to extend the applicability of a continual query. Other optional properties that can be added for a continual query may include quality of service parameters, execution timing constraints, contingency plans, and external events. Timing constraints include deadlines, priorities/urgencies or value functions. Contingency plans describe alternative actions to be executed in case the timing constraints cannot be met.

Continual queries, like all other forms of data, are treated as first class objects. CONQUER maintains a continual query entity type, and every continual query is an instance of this type. Methods defined on the continual query entity type include query execution method, trigger testing method, `Start` time and `Stop` time testing methods, as well as various notification methods.

3 System Architecture

CONQUER is built on a three-tier architecture: client, server, and wrapper/adapter. 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. A sketch of the CONQUER system architecture is given in Figure 1.

The client tier currently has four components. The *form manager* provides the CQ clients with fill-in forms to register to the system, install or drop their continual queries, etc. The *registration manager* allows clients to register the CONQUER system with valid user id and password, and return the clients a confirmation on their registration. The *client and system administration services* component provides utilities for browsing, updating, or deleting installed continual queries, for testing time-based and content-based CQ triggers, and for tracing the status or performance of the installed continual queries. Finally, the *Client manager* 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 key components of a continual query (Q, T_{cq}, `Start`, `Stop`, `Notify`), before storing it in the CONQUER system repository.

The second tier is the CONQUER server which consists of three main components: a continual query (CQ) manager with event-driven delivery, a trigger condition evaluation manager, and the event detection manager (including time-based event detector using clock event manager and content-based event detector). The CQ manager is responsible to coordinate with the trigger condition evaluator and event detection manager to monitor updates of interest, and coordinate with CONQUER wrappers and adapters to track the new updates to the source data. The trigger condition evaluation manager is in charge of evaluating the trigger condition for each installed continual query whenever time events or update events of interest are signaled by the event detection manager. The main task of the event detection manager is to decide what to detect, when to detect, and how to detect. The decision is made based on the update events identified from the trigger condition specification and the type of events to

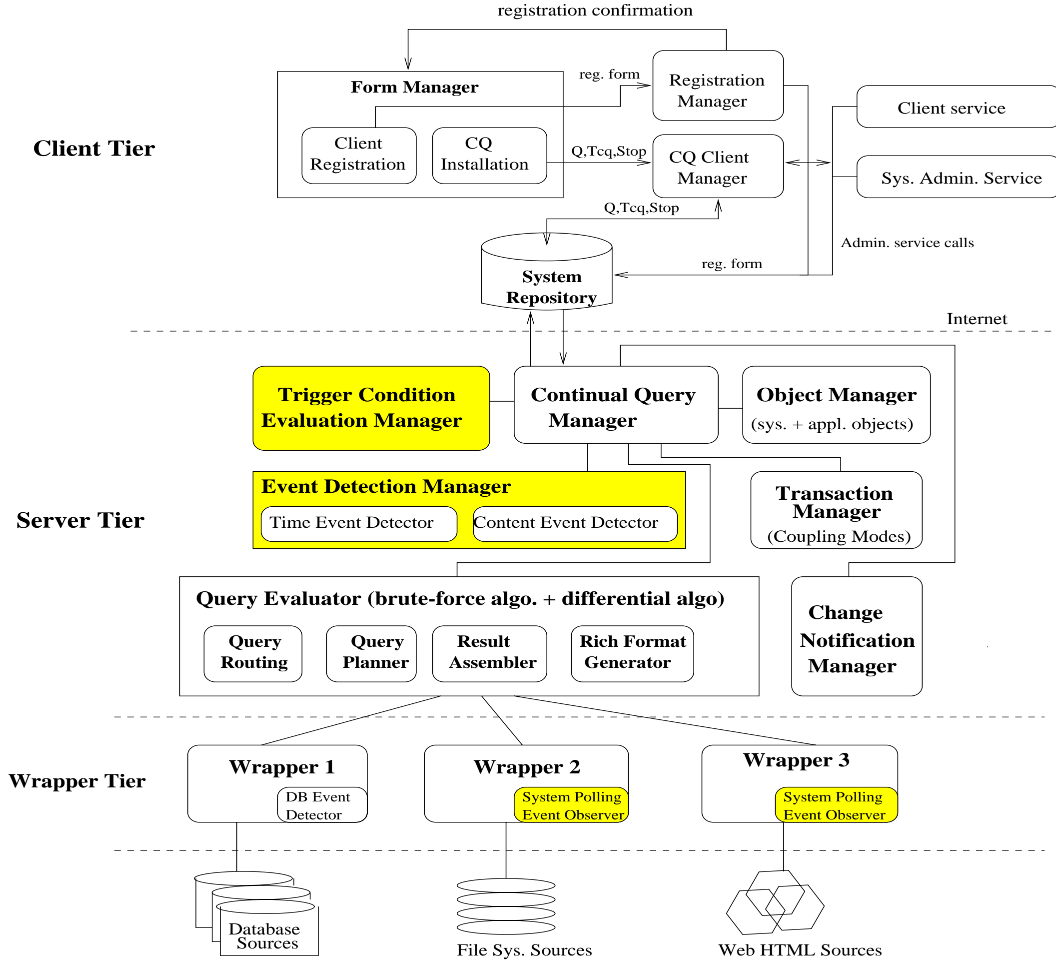

Figure 1: Architecture of a Push-enabled Continual Query system

be detected. We build the time-based event detector on top of the Cron clock event manager in the first prototype of CONQUER. The content-based event detector is built based on the change event observer and a set of specialized event observers, each designed for a particular event composition operator. We discuss the design of the CONQUER trigger evaluator, event detection manager, and change event observers (see the shaded boxes in Figure 1 in detail in the next two sections.

In addition, the CONQUER server uses the query evaluator, an extension of the DIOM query scheduler, for execution of the query Q whenever the trigger condition T_{cq} is evaluated to be true. It also provides a guard for the `Stop` condition to guarantee the semantic consistency of the continual query (Q, T_{cq}, `Start`, `Stop`, `Notify`). The key components of this query evaluator include: the query router [16, 13], the query planner [17] and the query result assembler. The query router is a key technology that enables the CONQUER continual query system to scale up to thousands of different information sources. When the user poses a query, the query router examines the query and determines which sites contain information that is most relevant to the user's request. Instead of contacting all the available data sources, the CQ query evaluator only contacts the selected sites that can actually contribute to the query.

The third tier is the CONQUER wrappers/adapters tier. The CQ manager, on behalf of the event observers and the query evaluator, talks to each information source using a CQ wrapper. 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 CQ wrapper installs a source-specific event detector that, on behalf of the CQ 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 source-specific event detector for RDBMS sources is a database trigger detector. For data sources with no built-in triggers on update operations, a system-controlled polling event detector is provided.

Depending on the need of applications, the client tier, the wrapper tier, and the server tier could all be located on a single host machine, or distributed in different combinations among several computers connected through local or wide area networks. CONQUER uses the most flexible client-server arrangement which is customizable with respect to the particular system requirement of the applications. For example, in the first version of the prototype, we have the client tier running remotely, and the CONQUER server running on a relatively powerful host machine, where we also maintain a library of all the current CQ wrappers including their source capability profiles.

A detailed description of CONQUER's components and interfaces is beyond the scope of this paper. However, we below walk through each of the main components of CONQUER by briefly tracing how the continual query firing process proceeds:

- Step 1: An update event occurs and is signaled by an event detector.
- Step 2: The CQ manager determines which continual queries are fired by the event. For each of these activated continual queries, the CQ manager calls On the corresponding wrapper managers to obtain the data that must be passed to condition evaluation manager, and possibly the query evaluator.
- Step 3: If any of the continual queries that are fired by the event have the immediate coupling mode for condition evaluation, then the CQ manager calls on the transaction manager to create a sub-transaction for condition evaluation, and then passes the event signal to the trigger condition evaluator.
- Step 4: The condition evaluator determines which continual queries are to be fired and returns a list of *cqid*s to the CQ manager. After condition evaluation is completed, the CQ manager calls the transaction manager to terminate the sub-transaction.
- Step 5: The CQ manager determines which of the continual queries to be fired have the immediate coupling mode for their query execution. The CQ manager calls on the transaction manager to create concurrent sub-transactions for each of these continual queries. Then the CQ manager calls on the wrapper manager(s) to execute the query in the corresponding sub-transaction(s).

The web site for the continual query project is `http://www.cse.ogi.edu/DISC/CQ/` and the demo system of the CONQUER development can be accessible at `http://www.cse.ogi.edu/DISC/CQ/demo`.

4 Conquer Trigger Processing Strategy

4.1 Overview

In Conquer 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 Conquer 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.

According to the discussion in Section 2, a continual query can be seen as a "long-running query" in the sense that a continual query (Q, T_{cq},`Start`,`Stop`), 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 and most of the cost in a single test of a trigger condition is on the access to the remote site(s) to fetch 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 against a single web information source or a set of web information sources of the same domain. Therefore, 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. Consider a typical example. Assume we have n users who want to monitor stock price update information (e.g., `Stock.price`) of some US companies over the stockmaster.com data source, each user is interesting in monitoring the stock price change of m companies separately. Thus we need to install $n * m$ content-based continual queries. Let T denote the time unit for the system-controlled polling interval for content-based CQs (e.g., every 30 seconds). If we index all $m * n$ installed CQs on the field `Stock.price`, then the number of concurrent trigger testing against the remote data source – www.stockmaster.com can be reduced from $m * n$ (one per CQ) to k. k is the number of companies that are being monitored by these n users. Obviously, using index will give us a dramatic saving on both network cost and system overhead, when a large number of users are interested in a smaller set of companies' stock price updates, i.e., k is significantly small comparing to the number of users and the number of installed CQs.

4.2 Canonical Representation of Trigger Conditions

As described in Section 2.2, trigger conditions of continual queries have the following general structure. The `FROM` clause refers to one or more object classes, and some may be suffixed with their data source URLs. For example, one may monitor the new books on particular authors or specific titles over amazon.com, BarnesNobel.com, or other online bookstore sources. Thus, he/she may either create CQs simply using `BOOK` as the object class in the `FROM` clause and let the Conquer system route the relevant online bookstore sites for his queries, or explicitly indicates that he would like answers from `Book@amazon.com`.

The `ON EVENT` clause may contain a list of events, and each is either a time event or a content-based event referring to the data sources specified in the `FROM` list. The `WHEN` clause of a trigger is a Boolean-

valued expression. For each combination of one or more objects referred in the data sources of the `FROM` list, the `WHEN` clause evaluates to true or false.

A canonical representation of the `WHEN` clause can be structured as follows:

- **Step 1**: Translate the `WHEN` clause to a conjunctive normal form (CNF), i.e., the and-of-ors notation. Each conjunct refers to one or more object classes. We denote each CNF by

  $$(C_{11} \vee C_{12} \vee ... \vee C_{1n_1}) \wedge (C_{21} \vee C_{22} \vee ... \vee C_{2n_2}) \wedge ... \wedge (C_{q1} \vee C_{q2} \vee ... \vee C_{qn_q}),$$

  where each C_{ij} $(1 \leq I \leq qand1 \leq j \leq n_q)$ denotes an atomic condition of the canonical form "A_{ij} `op` v_{ij}", v_{ij} is a constant value in the domain of A_{ij}, and `op` is a string or algorithmic comparison operator, depending on the type of A_{ij}.

- **Step 2**: Group the conjuncts by the set of object classes and data sources they refer to. Let CNF_i denote $(C_{i1} \vee C_{i2} \vee ... \vee C_{in_i}$ $(1 \leq i \leq q)$. The result of grouping CNF_i can be represented by

  $$(CNF_{11} \wedge ... \wedge CNF_{1h_1}) \wedge ... \wedge (CNF_{m1} \wedge ... \wedge CNF_{mh_m}),$$

  where $\sum_{i=1}^{m} h_i = q$.

  If a group of conjuncts refers to one object class, then we consider the logical AND of these conjuncts to be a selection predicate, and refer to this group as a selection predicate group. If it refers to two or more object classes, then we consider the logical AND of these conjuncts to be a join or a n-way-join predicate. These join predicates may or may not contain constants.

- **Step 3**: If the entire expression except the `WHERE` clauses has p constants, they are numbered 1 to p from left to right. For the constant number k $(1 \leq k \leq p)$, if it appears in C_{ij} in the original expression and C_{ij} is of the form "A_{ij} `op` v_{ij}", then the number k constant v_{ij} in C_{ij} is substituted with placeholder $\texttt{CONSTANT}_k$.

- **Step 4**: If the entire expression is a selection predicate group and it has q constants, they are numbered 1 to q from left to right. For the constant number h $(1 \leq h \leq q)$ if it is the substitute of v_{ij} in original expression C_{ij} of the form "A_{ij} `op` v_{ij}", then the operator `op` is substituted with placeholder $\texttt{OPERATOR}_h$

We call the resulting expression the *trigger pattern* of the original expression. A trigger pattern signature is defined as a triple consisting of the set of object classes, the set of events specified in the `ON EVENT` clause, and the trigger pattern itself. When the `ON EVENT` clause is NULL, an `UPDATE` event will be used as the default since every content-based CQ is monitoring updates over the data source(s) specified in the `FROM` clause.

Example 4 Suppose that we have two continual queries installed by two different users. Both are used to monitor the stock price of Microsoft, and one has the update threshold "decreased by 10%" and the other "decreased by 5%". The trigger conditions of these two CQs are expressed as follows:

```
(1) FROM Stock                              (2) FROM Stock
    WHEN Stock.price DECBYPERC 10               WHEN Stock.price DECBYPERC 5
         WHERE symbol = 'MSFT'                       WHERE symbol = 'MSFT'
```

By going through the first three steps of the canonical transformation, the trigger expressions of these two CQs are as follows:

```
(1)  WHEN Stock.price DECBYPERC CONSTANT_11 WHERE symbol = 'MSFT'
(2)  WHEN Stock.price DECBYPERC CONSTANT_21 WHERE symbol = 'MSFT'
```

Since these two trigger expressions are both selection predicates, by Step 4 of the canonical transformation the signatures of these two trigger conditions are constructed as follows:

```
(1)   ObjClass: Stock
      OnEvent:  UPDATE
      Pattern:  WHEN Stock.price OPERATOR_11 CONSTANT_11
                     WHERE symbol = 'MSFT'

(2)   ObjClass: Stock
      OnEvent:  UPDATE
      Pattern:  WHEN Stock.price OPERATOR_21 CONSTANT_21
                     WHERE symbol = 'MSFT'
```

Clearly, these two trigger conditions have the same trigger pattern signature although they have different constant values. We treat these two trigger conditions as the two instantiations of the same trigger patterns "WHEN Stock.price OPERATOR CONSTANT WHERE symbol = 'MSFT'. In general a trigger pattern signature defines a class of all instantiations of that trigger pattern, each instantiation is with a different constant value.

An interesting observation is that these two example CQs are also instantiations of a more specialized trigger pattern "WHEN Stock.price DECBYPERC CONSTANT WHERE symbol = 'MSFT'. Our experience tells that if there are too many continual queries falling into a single trigger pattern, it may be beneficial to use more specialized trigger patterns than the general one that replaces both constants and comparison operators. In current implementation of Conquer, the application of Step 4 in the canonical transformation algorithm is optional.

Another practical matter is that most of the ON EVENT clauses will have a time event for time-based trigger and an UPDATE event for content-based trigger. In the current prototype implementation of Conquer, only time events and UPDATE event are supported, although the ON EVENT clause conceptually contains other update events. Thus, for presentation convenience, in what follows we restrict the discussion on the ON EVENT conditions to time events, and treat the ON EVENT condition separately from the WHEN condition for convenience.

4.3 Continual Query Indexing Strategy

The key idea behind the continual query indexing strategy 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. In this section we discuss the continual query index structure and the indexing strategy based on the concept of trigger patter signatures.

When a continual query installation request is processed, a number of steps must be performed to update the Conquer system catalogs and CQ index structures, and prepare the trigger of the CQ to be ready to run. The primary table in the catalog is the `CQ_Info` table:

```
CQ_Info(cqid, cqName, Query_text, Trigger_text, Start_cond,
        Stop_cond, creationDate, status, ...)
```

Most of the fields are self-explanatory. The field `status` is used to indicate whether a continual query is currently active. The installation of a continual query is completed after it is successfully recorded in the `CQ_Info` table.

When a continual query is successfully installed, and the canonical transformation of a CQ trigger condition is completed, any new trigger pattern signatures detected are added to the following table in the Conquer system catalog:

```
TrigPatternSig(sigID, objClassID, OnEvent_desc, pattern_desc,
               constOpTableName, constOpTableSize, ...)
```

The `sigID` field is a unique ID for a trigger pattern signature. The `objClass` field indicates the object classes on which the trigger pattern is defined. The `pattern_desc` field is a text field with a description of the trigger pattern expression. The `constOpTableName` field is a string giving the name of the constant-operator table for a trigger pattern signature. The `constOpTableSize` field gives the number of constant-operator pairs appearing in the trigger pattern expression for the given signature.

In addition, each continual query is linked to a set of object class indexes, organized using a hash table on object class ID. Each *object class index*, contains a list of trigger pattern indexes, each with one entry for the trigger pattern signature that has been used by one or more CQ triggers as a predicate on that object class and a list of CQ identifiers. For each trigger pattern containing one or more constant placeholders, a *constant table* will be created. It contains a list of CQ identifiers with one row for each instance expression of the trigger pattern, which is equivalent to the original trigger expression in some CQ.

When a trigger pattern signature is processed at continual query installation time, it is broken into two parts: the indexable part and the non-indexable part. The non-indexable portion is NULL when the trigger condition is simply a CNF with a group of selection predicates. The format of the constant table for the trigger pattern signature containing N distinct constants is described as follows:

```
constOpTable_Num(indexable_exp_id, cqid, indexable_exp_desc, constOpHolder1,
                 ..., constOpHolderN, nonIndexablePart_desc)
```

The `Num` attached to the name of a constant table is the ID number of the trigger pattern signature. There is a row of the table `constOpTable_Num` for each continual query identified by `cqid`. The key components of each row includes the expression ID of the indexable part of the trigger expression, the unique ID of the CQ containing the indexable expression, the indexable expression description in text, the operator-constant pairs found in the indexable part of the trigger expression of this CQ, and the remaining portion of the trigger condition that is non-indexable.

In short, whenever a trigger pattern is derived from newly installed CQ, the system will check to see if its signature has been seen before. If it is not found in the table `TriggerPatternSig`, this new trigger

pattern will be added into the `TriggerPatternSig` table. If this signature has at least one constant placeholder in it, a constant table is created for this trigger pattern signature with a entry added to record the indexable part, the list of constant-operator pairs, and the non-indexable part of this newly installed CQ.

Example 5 Consider a number of continual queries that monitor the stock price change information of either IBM or Microsoft. Assume that each CQ has a different update threshold of interest.

```
cq1: Create CQ ibm_stock_watch_1 as
     Query: SELECT * FROM Stock WHERE symbol = 'IBM';
     Trigger: FROM Stock WHEN Stock.price DECBYVALUE 10 WHERE symbol = 'IBM';
     Stop: in 3 months

cq2: Create CQ ibm_stock_watch_2 as
     Query: SELECT * FROM Stock WHERE symbol = 'IBM';
     Trigger: FROM Stock WHEN Stock.price DECBYVALUE 20 WHERE symbol = 'IBM';
     Stop: in one year

cq3  Create CQ ibm_stock_watch_3 as
     Query: SELECT * FROM Stock WHERE symbol = 'IBM';
     Trigger: FROM Stock
              WHEN Stock.price DECBYPERC 5 WHERE symbol = 'IBM';
     Stop: in 6 months

cq4: Create CQ my_stock_watch as
     Query: SELECT * FROM Stock WHERE symbol = 'IBM';
     Trigger: FROM Stock, CompanyNew
              WHEN Stock.price INCBYVALUE 10 WHERE symbol = 'IBM'
                   AND CompanyNew.cname = Stock.cname;
     Stop: in 18 months

cq5: Create CQ msft_stock_watch as
     Query: SELECT * FROM Stock WHERE symbol = 'MSFT' OR symbol = 'MSFT';
     Trigger: FROM Stock
              WHEN Stock.price = 140 WHERE symbol = 'IBM'
                   AND Stock.price DECBYPERC 5 WHERE symbol = 'MSFT';
     Stop: in 35 weeks
```

According to the indexing strategy discussed in this section, the first three CQs share the same trigger pattern "`Stock.price OPERATOR`$_I$ `CONSTANT`$_I$" ($I = 1, 2, 3$). The continual query cq4 has a join predicate in its trigger condition. Thus, the indexable part of the trigger expression in cq4 is "`Stock.price INCBYVALUE 10 WHERE symbol = 'IBM'`", and the non-indexable part of the trigger is "`AND CompanyNew.cname = Stock.cname`". cq5 has the entire trigger expression indexable and it conforms to the trigger pattern below:

```
Stock.price OPERATOR_1 CONSTANT_1 WHERE symbol = 'IBM'
AND Stock.price OPERATOR_2 CONSTANT_2 WHERE symbol = 'MSFT'.
```

However, cq5 does not conform to the trigger pattern that the first three CQs shares and is not grouped with the first three CQs for batch processing, although the first part of cq5's trigger pattern is the same as the one that cq1, cq2, and cq3 have in common.

An obvious extension of the CQ indexing strategy described in this section is to explore the alternative of indexing CQs on a subset of the indexable expressions. This is especially desirable when a large number of CQs have more sophisticated trigger conditions sharing only partial indexable selection predicates. One idea is to choose the most selective conjunct for indexing, which is the topic of the next section.

4.4 Index on Most Selective Atomic Trigger Patterns

Recall Section 4.2, every trigger condition pattern is a CNF. The simplest trigger pattern is a CNF with a single conjunct containing no OR operators and has the form "`[<objClassName>.]<fieldName> <comparisonOp> <CONSTANT>`". We call such a trigger pattern an *atomic trigger pattern.*

For convenience of discussion, in this section we consider each trigger pattern TP, produced from the canonical transformation of the original trigger expression, to be a selection predicate, consisting of a set of atomic trigger patterns. We delay the treatment of OR predicates and join predicates to later sections.

Let a trigger pattern be denoted by $AC_1 \wedge AC_2 \wedge ... \wedge AC_m$, where each AC_i $(1 \leq i \leq m)$ is of the form "F_i op_i $\texttt{CONSTANT}_i$" and F_i denotes a filed name possibly with a class name referenced in the `FROM` clause as the prefix (e.g., `Stock.price` in Example 4).

When a continual query is routed to a single source, these conjuncts can be processed together as a single selection predicate. When the continual query is routed to a set of n data sources, then the CQ trigger manager will spawn n threads, each thread processes the selection predicate at one source.

For each given trigger pattern, the next decision is to select the atomic trigger pattern that is most selective to index. More concretely,

- If $n = 1$, then AC_1 is the only conjunct and it is indexed directly.
- If $n > 1$, then a single conjunct is identified as the most selective one. Only this one is indexed directly.

The rest of conjuncts are located and tested only if a data object fetched from the remote data source(s) matches the most selective conjunct. If the remaining conjuncts in the trigger condition match, then the data object has completely matched the trigger condition.

Using the alternative indexing approach based on the most selective atomic trigger patterns of each CQ, the five continual queries will all be eligible to be grouped by indexing on the atomic trigger pattern "`Stock.price OPERATOR_1 CONSTANT_1 WHERE symbol = 'IBM'`". Thus, all the five CQs can be processed using a single remote access to the relevant data sources, instead of a single remote access for each CQ.

4.5 Optimization of Other Complex Trigger Expressions

A potential topic for future work is to optimize the processing of selection predicates containing ORs, joins, or very expensive functions [10]. The main idea is to explore optimization opportunities inherent in the particular properties of each type of operators. For example, when a conjunct of the given trigger condition containing ORs, in general none, some or all of the ORed clauses it may be true. One extreme case is that if one of the ORed clauses is true, then we can conclude that the entire conjunct is true. By properly ordering the ORed clauses, this observation will help in speeding up the evaluation of the conjunct with ORs. However, such a solution may not perform well all the time. The worst case is the other extreme situation where all ORed clauses are not true. Our initial proposal for handling OR predicates is to index each atomic ORed clause whenever the number of CQs sharing the same atomic trigger pattern reaches certain system-defined threshold. Other solutions are also being investigated.

We also work on the strategies for offering both memory-based and disk-based data structure as alternatives for organization and utilization of CQ indexes. The main idea along this line of work is built on top of the Rete [6] and TREAT [22] algorithms for efficient implementation of AI production systems. It is known that the Rete and TREAT algorithms that were primarily based on the implicit assumption that the number of rules in AI rule system architectures is small enough to fit in main memory. A main challenge for `Conquer` is to develop strategies and mechanisms that can automatically manage the utilization of memory-based or disk-based CQ indexes at runtime.

4.6 Parallel Processing of Continual Queries

It is commonly recognized that the use of parallelism and concurrency to make scare CPU and I/O resources available to multiple tasks can give better throughput and response time even for a single processor. In Conquer we promote the use of parallel processing as an important way to obtain better scalability. A number of parallel processing methods are currently exploited for improving scalability of Conquer in processing large numbers of continual queries.

In the previous sections we have talked about concurrent processing strategies used in the Conquer continual query processor at query level, trigger level, event level, and data level. The *query-level* of parallelism refers to the strategy that processes a CQ query at multiple web information sources concurrently. The *event-level* of parallelism is also called trigger-pattern-level of concurrency, by which we mean that multiple groups of CQ triggers can be processed in parallel through the use of CQ indexes on trigger patterns. Each trigger pattern is processed at the set of information sources selected by the query router through the event observers at the Conquer wrapper tier (recall Section 3). Thus each trigger pattern corresponds to multiple concurrent change event observation tasks. The *trigger-level* of concurrency means that multiple trigger conditions within each partition group can be tested concurrently against a single set of data objects, which are often fetched remotely using the indexed trigger pattern and filtered at the wrapper tier and the mediator tier for each CQ conforming to the given trigger pattern. The *data-level* of concurrency is referring to the support of concurrent access to the same set of data objects.

In order to capitalize on the various types of concurrent processing, Conquer has been implemented primarily using Java multi-thread programming language features. Instead of using a task queue kept in shared memory to store incoming or internally generated work, the current implementation of the Conquer system spawns multiple threads to execute multiple tasks rather than explicitly managing a task queue. Once a continual query is fired, a CQ driver process is invoked. Multiple driver processes

can call `Conquer` trigger evaluator concurrently, each driver process makes such a call every T time units (such as every 200 milliseconds for local processes and every 5 minute for remote connection). The default value of T should allow timely trigger execution without excessive or unnecessary overhead for communication between the driver processes and the remote information source servers. The factors to be used to determine the best value of T is an interesting issue in our list of further work. Once a driver program is started, it waits for T time units to make the next call to `Conquer` again.

Another strategy we use is to keep the execution time inside `Conquer` reasonably short. A long execution could result in higher probability of faults such as running out of memory or deadlocks and the problem of excessive work to be lost if a rollback occurs during continual query processing. The detail of continual query recoverability feature is beyond the scope of this paper and interested users may refer to our incoming technical reports.

5 Event Observers for Change Detection on the Net

Another interesting challenge to Conquer is the fact that Conquer targets at the web information sources and most of the web sources either have no built-in trigger capability or do not export such change notification function, which can locally detect and notify a change when it occurs. As a consequence, Conquer needs to build a change event observer for monitoring and detecting changes at a remote web information source.

A change event observer is a program that performs a pull query or pull-based page fetch request periodically over a remote information source and discovers what changes were occurred recently and what are the types of these changes. Currently, we are working on designing and developing of content-based change event observers that can detect the specific content changes as well as the types of content changes. All Conquer event observers are built on top of the wrappers generated semi-automatically using the XWRAP technology [14].

Once a continual query is entered in the system and classified to a specific index partition identified by a trigger pattern signature. The trigger evaluation manager will generate a pull query per data source for each trigger pattern (such as a pull query over `Stock.price@stockmaster.com` WHERE symbol = 'IBM' for the trigger pattern derived from Example 5). Once the change event observer receives a pull query, it first calls the wrapper of the corresponding source to fetch the data objects of interest, and then compute the recent changes as well as the types of the changes before sending its response to the trigger evaluator. When a trigger evaluator receives a change report from the change event observer, it will locate and test the rest of the trigger condition for each CQ contained in the partition. A change notification is sent out to the CQ users only if the data objects discovered by the change event observer satisfy all the rest of trigger condition of a CQ.

Figure 2 shows an example of a continual query for monitoring the national weather report site and detecting and notifying content-sensitive changes to the perspective users. Figure 3 shows a differential query result collected in a notification to the specific user. In this case the user is the owner, namely the person or application program who installed this continual query.

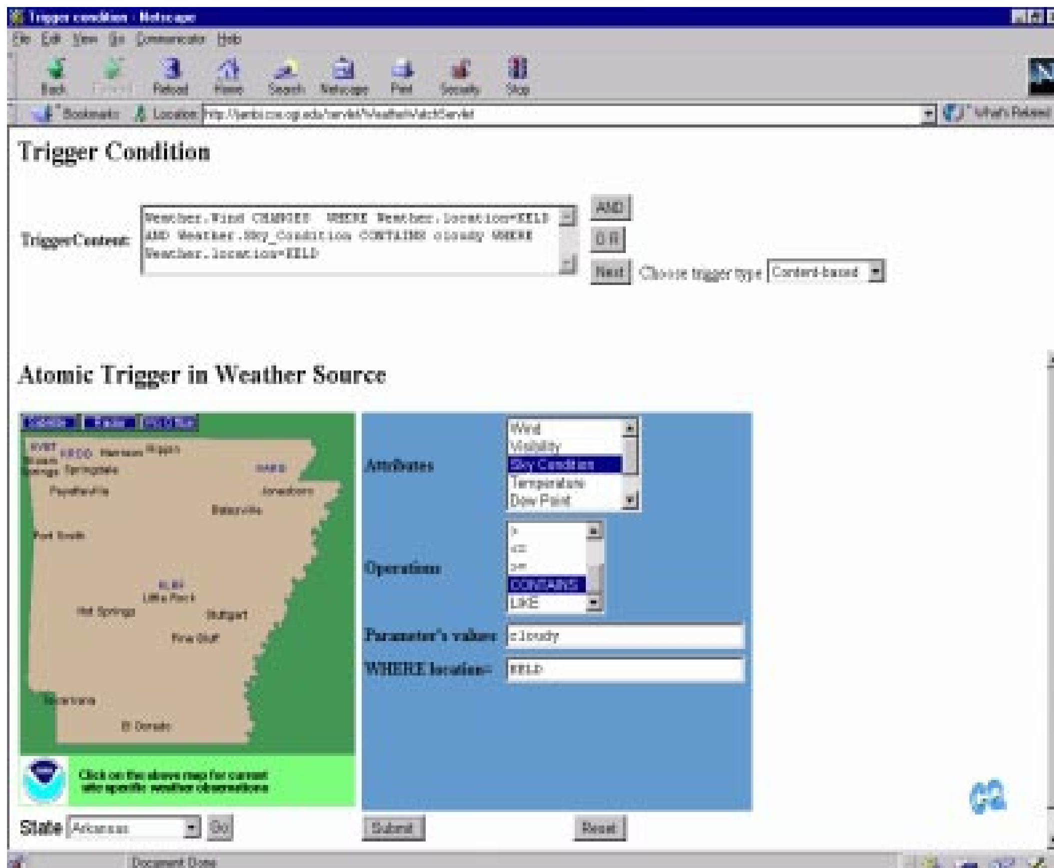

Figure 2: Continual Query Installation: An example

6 Performance Evaluation Issues

We have presented a design and selection of alternative architectures and algorithms for a distributed continual query system CONQUER. Research on push-enabled continual query systems must be accompanied by a careful performance evaluation effort. For the CONQUER project, such effort is under way. The goal of the first effort is rather modest, that is to verify that a continual query system can indeed outperform a pull-based passive data delivery system for applications that require time-constrained update monitoring. Towards this objective, a simple condition monitor and a small situation monitoring application were implemented using C, Perl, JDK1.1, JDBC, and Oracle 7.0, upgraded to Oracle 8.0. Three types of data sources are used in this prototype: (1) an Oracle database which is remotely accessible through SQL, OraPERL, and SQLNet; and a Microsoft SQL server database which is remotely accessible through JDBC and SQL; (2) a collection of semi-structured UNIX files which are accessible through Java Applets and Java Servelets. (3) a World Wide Web HTML source which is accessible through our html wrapper and filter utility. We are planning to do a simple experiment, making a comparison between user polling and continually monitoring using continual queries. We expect (with confidence) that this simple experiment will verify the hypothesis that push-enabled data delivery system can outperform (ad-hoc) polling over a pull-based passive data delivery system when the number of objects being updates and monitored is proportionally large.

Figure 3: The change report by a change event observer at nws.noaa.org site

We are currently planning careful controlled experiments for comparing the performance of alternative condition evaluation tactics. This effort also includes studying architectural alternatives for the push-enabled continual query systems and their impact on performance. We are also interested in building a performance test bed for studying the extent to which the final design of the Conquer continual query system is able to meet or exceed the processing requirements of a distributed time-constrained update monitoring system.

7 Related Work

The concept of continual queries was motivated by the increasing demand on event-driven information delivery. The work on continuous queries in append-only databases by Terry et al [28] started this line of research. At the same time, there has been considerable research done in the monitoring of information changes in databases. The design of Conquer system has been influenced by the work in active databases and view materialization. Both areas have primarily for "data-centric" environments, where data is well structured, well organized and controlled. When applied to an open information

universe as the Web, many of the assumptions no longer hold (see [15] for a summary of desired system properties in the Internet). CONQUER bridges this gap.

Active Databases: Event-Condition-Action (ECA) systems are rule-based programs in which an event triggers the testing of a condition, which in turn (if true) triggers an action. Most of active database systems [29] provide facilities [3, 21, 24, 7, 26, 9, 5] that support ECA rules referring to actions and events such as changes of database state. Some popular active database research prototypes include HiPAC [5], ODE [7], Postgres [26], Starburst [9]. These systems support powerful rules and allow very general events, conditions, and actions, and therefore are difficult to implement efficiently. The result is fairly restricted implementations (e.g., built-in triggers [11] in relational database management systems such as Oracle, Sybase, and Informix). Active queries, introduced in Alert [24], are more sophisticated than database triggers, since they can be defined on multiple tables, on views, and can be nested within other active queries. However, active queries rely heavily on a number of extensions specific to the IBM Starburst DBMS [9].

Comparing with the state of art of research in active databases, the CONQUER system differs primarily in the following three ways: First, the CONQUER system targets at update monitoring in 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 CONQUER system provides efficient and scalable trigger processing to handle large numbers of concurrently running continual queries on a variety of data sources.

Materialized Views: Materialized views store a snapshot of selected state of the database. When a database is updated, the materialized views must be refreshed to reflect the updates. System performance is improved by the reads satisfied by a materialized view before updates happen. Furthermore, incremental update algorithms lower refresh costs if changes to the database are moderate [12]. However, updates do happen and refreshes are necessary. One could refresh the view eagerly, immediately after each update to the base table [2]. A lazy approach defers the refresh to the moment an important query is issued against the view [23]. Somewhat in the middle, one could refresh the view periodically [12]. The main tradeoff in choosing among these approaches is the staleness of the view data vs. the cost of updating it. Most of the algorithms in the literature [2, 8] work in a centralized database environment, in which the materialized view and its base tables co-reside. The study on distributed materialized view management has been primarily focused on determining the optimal refresh sources and timing for multiple views defined on the same base data [25]. Other works on distributed environments include quasi-copies for replication [1] and the approach proposed by Stanford [31] for resolving update anomalies in distributed data warehouses.

Web-based push-enabled systems There are several systems developed towards monitoring source data changes. One type of systems is the extension of Web search engines or search software by monitoring URL changes and notifying the uses whenever the URLs of the data sources of interest have changed. A representative system is the URL-minder (`www.netmind.com`). Another interesting web page change notification tool is the news monitor by AT&T (`research.att.com/ chen/web-demo/`. It watches some selected web pages for the users from a small and fixed selection of web sites. Another pioneer news monitoring system is the SIFT [30] information dissemination system at Stanford.

The most closely related project at the time of the writing, to our knowledge, is the C3 project [4] for change management. C3 develops a query subscription mechanism that allows users to subscribe the

data sources they are interested in detection of changes as well as query over the change databases. The C3 system periodically goes to the subscribed web sites, fetches the pages, applies the HTML-based `diff` functions to derive the types of changes, marks up the result pages with INSERT, DELETE or UPDATE changes, and archives the changes in the change databases.

The main difference between the Conquer continual query system and the C3 system is the following. First, the continual query system supports both time-based and content-based update monitoring (such as notify me whenever the temperature pressure drops 5%), whereas the current implementation of the C3 system only supports time-based change detection. Second, the continual query concept provides a simple and clean mechanism that ties the query subscription, the trigger condition, and the stop condition using a continual query; whereas the C3 system provides no conceptual glue for query subscription and user-defined polling interval. It is also unclear how a user query subscription may terminate. On the other hand, there are several interesting features that C3 provides which can be applied or integrated into the future continual query system development. For example, the difference functions provided in C3 use hypertext tree structure and thus are quite flexible and powerful. The mechanisms used for change notification in C3 include the choice of change notification by explicit query over the change database the C3 system maintains. The main problem is how long the system keeps the historical update information. Quite differently, in the current Conquer implementation, change results are maintained only until the corresponding continual query expires. No change databases are created for supporting search over historical data updates.

8 Conclusion

Conquer is a continual query system for update monitoring in the Web. In this paper, we describe the formal semantics of continual queries and the design and implementation of Conquer. An important part of Conquer is a well-defined continual query specification language. Its specification is derived from the division of continual query semantics into three parts: (1) queries over semi-structured data, (2) distributed event and trigger specification, and (3) semantics of validity and determinism of continual queries.

One of the main design goals of Conquer 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 key 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.

Conquer is designed in a modular way. Its key components include a continual query (CQ) manager, a trigger condition evaluator, and a set of event observers. The prototype implementation of Conquer is built on top of the distributed interoperable information mediation system DIOM [17] plus the reuse and extension of conventional DBMS components. The prototype provides push-enabled services by incorporating distributed event-driven triggers, and combining pull and push services in a unified framework. An example of future research topics is the experimentation and performance evaluation of alternative architectures and algorithms. Another example is the performance improvements by incorporating research results in incremental query evaluation and multiple query optimization techniques into the event detection and condition evaluation algorithms.

Acknowledgement

We would like to thank the special issue editors Dr. Dan Suciu and Dr. Letizia Tanca for their effort in putting this special issue together and for the reviewers for their helpful comments and suggestions. We would also like to thank the entire CQ team for the many interesting discussions and the development endeavor, specially to David Buttler for his coordination and implementation on the query router component and various pieces of earlier wrappers; and to John Biggs for his contribution in the early tage of the CONQUER system development.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[2] 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.

[3] S. Chakravarthy. Architectures and monitoring techniques for active databases: An evaluation. In *Technical Report TR-92-041, University of Florida*, Gainesville, FL, 1992.

[4] S. Chawathe, S. Abiteboul, and J. Widom. Managing and querying changes in semi-structured data. In *Proceedings of ACM SIGMOD Conference*, 1997.

[5] U. Dayal, B. Blaustein, A. Buchmann, C. Chakravarthy, M. Hsu, R. Latin, D. MaCarthy, A. Rosenthal, S. Sarin, M. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. In *ACM SIGMOD Record 17(1)*, March 1988.

[6] C. L. Forgy. A Fast Algorithm for the Many PatternMany Object Pattern Match Problem. *Artificial Intelligence*, 19, 1982.

[7] N. Gehani, H. Jagadish, and O. Shmueli. Composite event specification in active database: Model and implementation. In *Proceedings of ACM SIGMOD Conference*, 1992.

[8] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[9] 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.

[10] J. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, (To appear), 1988. Available at www.cs.berkeley.edu/ jmh.

[11] Informix Software, Inc. *Informix Guide to SQL: Syntax (Version 6.0)*, 1994.

[12] 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.

[13] L. Liu. Query routing in large-scale digital library systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, IEEE Press, March 1999.

[14] L. Liu, W. Han, D. Buttler, C. Pu, and W. Tang. XWRAP: An XML-based Wrapper Generation Toolkit. In *Technical Report, OGI/CSE, October*, 1998.

[15] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 1995.

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

[17] 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.

[18] 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.

[19] 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.

[20] L. Liu, C. Pu, W. Tang, J. Biggs, D. Buttler, W. Han, P. Benninghoff, and Fenghua. CQ: A Personalized Update Monitoring Toolkit. In *Proceedings of ACM SIGMOD Conference*, 1998.

[21] 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.

[22] D. P. Miranker. Treat a better match algorithm for ai production systems. In *Proceedings of the AAAI Conference*, August 1987.

[23] 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.

[24] 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.

[25] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the 6th International Conference on Data Engineering*, pages 512–520, Los Alamitos, February 1990.

[26] M. Stonebraker, E. Hanson, and C. H. Hong. The design of the Postgres rules systems. In *Proceeding of the International Conference on Data Engineering (ICDE)*, 1987.

[27] W. Tang. Personalized update monitoring toolkit using continual queries. MSc. thesis, University of Alberta, Department of Computing Science, 1998.

[28] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA, January 1992.

[29] J. Widom and S. Ceri. *Active Datanase Systems*. Morgan Kaufmann, 1996.

[30] 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.

[31] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.