Continual Queries for Internet Scale Event-Driven Information Delivery

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

Abstract

In this paper we introduce the concept of continual queries, describe the design of a distributed event-driven continual query system – OpenCQ, and outline the initial implementation of OpenCQ on top of the distributed interoperable information mediation system DIOM [21, 19]. Continual queries are standing queries that monitor update of interest and return results whenever the update reaches specified thresholds. In OpenCQ, users may specify to the system the information they would like to monitor (such as the events or the update thresholds they are interested in). Whenever the information of interest becomes available, the system immediately delivers it to the relevant users; otherwise, the system continually monitors the arrival of the desired information and pushes it to the relevant users as it meets the specified update thresholds. In contrast to conventional pull-based data management systems such as DBMSs and Web search engines, OpenCQ exhibits two important features: (1) it provides push-enabled, event-driven, content-sensitive information delivery capabilities, and (2) it combines pull and push services in a unified framework. By event-driven we mean that the update events of interest to be monitored are specified by users or applications. By content-sensitive, we mean the evaluation of the trigger condition happens only when a potentially interesting change occurs. And by push-enabled, we mean the active delivery of query results or triggering of actions without user intervention.

Keywords: Distributed Information Systems, Internet Scale Information Delivery, World Wide Web Technology, Distributed Triggers and Event Monitoring.

1 Introduction

1.1 Motivation

With the ongoing advance of World Wide Web (WWW or the web), everyone can publish information on the web independently at any time. While the flexibility and autonomy of information production and sharing is phenomenal, it is equally daunting to navigate, collect, process, and track data in this dynamic and open information space. The problem is aggravated when source information changes constantly and unpredictably. As a result, users have to frequently poll the web sites of interest and fuse the newly updated information manually to keep track of changes of interest, which is a great pain.

As more aspects of business and commerce migrate online, there is increasing interest in having software systems execute and interoperate over the Internet. Execution and interoperation at Internet scale imply a degree of loose coupling and heterogeneity among the components. A promising architectural style for distributed, loosely-coupled, heterogeneous software systems is a structure based on push-enabled event generation, observation (event detection and condition triggering), and notification. Event-driven information delivery systems becomes increasingly important because it offers system-supported update monitoring, event-driven information delivery, and it provides timely response (or alert) to critical situations, while reducing the time users spend hunting for the updated information and avoiding unnecessary traffic on the net. The technology to support event-based systems is well developed for local area networks (e.g., Field's Msg [29], Softbench's BMS [10], Yeast [16]) and for active databases (e.g., [5, 26, 32, 38]), but off-the-shelf software is still premature for Internet scale applications. Hence new technologies are needed to support the construction of Internet-scale, event-driven software systems.

Recently there have been a small number of proposals and initial prototypes for Internet-scale event facilities, such as OMG CORBA Event Service [27] and the TINA Notification Service [9]. But the definitions of these facilities address only a limited portion of the full problem space. Furthermore, most of the implementations of CORBA Object Request Brokers are quite low level, missing the semantics of applications. Therefore, we design and develop a new facility for event observation and notification using distributed triggers, in the context of the Continual Queries project, to better serve the need of Internet-scale applications.

In this paper we describe the concept of continual queries and a distributed event-driven information delivery system, OpenCQ, that is designed for supporting continual queries. Continual Queries (CQ) are standing queries that monitor update of interest and return results whenever the update reaches specified thresholds. Continual queries can be specified over system built-in as well as user-defined operations. OpenCQ is a distributed continual query system for event-driven information delivery, equipped with distributed event observation, notification, and triggering facility. In OpenCQ, users may specify to the system the information they would like to monitor (such as the events or the update thresholds they are interested in). Whenever the information of interest becomes available, the system immediately delivers it to the relevant users; otherwise, the system continually monitors the arrival of the desired information and pushes it to the relevant users as it meets the specified update thresholds. In contrast to conventional pull-based data management systems (such as DBMSs, Web Search Engines) where the scope of a query is limited to past and present data, in OpenCQ, the scope of a query in a event-driven continual query system includes past, present, and future data. For example, a query "*give me the time-series of water temperature at Tansy Pt. during the last month*" is a typical query in a pull-based DBMS, which is defined over the time-series of water temperature at Tansy Pt. that are available up to the moment when this query is issued. In contrast, a request "*report to me in the next four weeks the time-series of water temperature at Tansy Pt. every day at 10:00am*" is a continual query in the OpenCQ continual query system. It has a query component

asking for the *time-series of water temperature at Tansy Pt.*, a trigger condition: *10:00am everyday*, and a stop condition: *the next four weeks*. The scope of this continual query covers the time-series of water temperature at the installation time of this continual query, plus the time-series of water temperature changes that may arrive in the future.

1.2 Overview of OpenCQ

The ultimate goal of the OpenCQ research is to develop a toolkit for Internet-scale update monitoring with event-driven evaluation and delivery. Our framework is organized around five abstract models, each of which focuses on a different dimension of concern in the design.

- An *object model*, which characterizes the information producers (that generate events) and the consumers (that receive notifications about events). Both user-defined objects and system-defined objects are components of the object model. An object can be a hardware or a software component, e.g., a clock, a file, a program, a process, or a communication packet. The entities that receive notifications about events are also objects. The object model incorporates the standard notion of encapsulation of functionality.
- An *event model*, which characterizes distributed events and a *time model*. Events may be defined in the time dimension (e.g., every 10 minutes or 10am everyday) or in the information space (e.g., *currents at a station in the Columbia River estuary exceed 3m/s* or *IBM stock price drop by 10%*). Events are either primitive or composite. Primitive events are further characterized as being *predefined* or *user-defined*. Predefined events can be automatically detected by the server, in particular, the server polls the system environment for their occurrences. User-defined primitive events must be registered in the server by a user or a program such that the server can have information about the semantics of user-defined events in order to detect their occurrences automatically. The time model concerns the temporal and causal relationships between events and notifications and addresses the problems of synchronizing clocks across distributed systems as well as associating times with events in distributed systems. According to this event model, an event can be uniquely characterized by the identity of the object of interest involved in the event, the identity of the operation, the identity of the invoker, and the time of occurrence of the event. An event is observable if some object other than the object of interest and the invoker can detect the occurrence of the event; and it is up to the object of interest to determine which of its events can be observable.
- An *observation model*, which defines the mechanisms by which event occurrences and patterns of event occurrences are observed. We refer to the objects who observe or detect the update events of interest the event observer or event detector. The observation model addresses issues such as: (1) what mechanisms can we use to achieve the observation of an event, (2) how event-specific information is to be requested and observed, (3) what kinds of event patterns need to be specified, (4) how should event patterns be recognized and observed, (5) in which way observation tasks are partitioned among observers, (6) how event-specific information is used to select events for notification, and (7) when, where, and how event observation objects (observers) are created and destroyed.
- A *notification model*, which defines the mechanisms that users or applications use to express interest in events and receive notifications. In OpenCQ we use the continual query specification language to express the update event monitoring requests. We treat notifications as independent communications between event detectors (observers) and recipients. This becomes particularly important when there are multiple independent observers involved.

- A *resource model*, which defines where in the Internet the observation and notification components are located, and how resources for the computations are allocated and accounted. Typically each information source will be wrapped in a resource model that establishes the correspondence between local resource entities and OpenCQ objects defined in the object model. We can view the resource models as the wrappers to the information sources. For those data sources that have no built-in trigger facility as those in RDBMSs, their wrappers have to provide base event detection capability.

To illustrate the concepts of these five models, let us consider the example of forecasting currents at the mouth of the Columbia River as an aid to navigation. The entity Columbia River and the entity Forecasted Currents are conceptual level objects that we are interested in monitoring, thus they are elements of the object model. If we are interested in monitoring the Columbia River forecasted currents every 15 minutes during ebb tide, then the event entity 15 minutes and the constraint ebb are the elements of the event model. If the monitoring condition is *whenever the maximum forecasted current at the mouth exceeds 3m/s*, then the base event consists of updates of the maximum forecasted current and the condition is such current exceeding 3m/s. The observation model watches the circulation forecasts and enters the notification model when the maximum currents reaches 3m/s. The resource model describes the results obtained from forecast data sources for the geographic region at the mouth of the Columbia River. One of the most popular notification mechanisms is by email whenever the event of interest is detected or observed, but in the case of this example it could also be a broadcast at a marine radio frequency.

We believe that by incorporating distributed events and triggers into the query evaluation and search process, the information quality, system scalability, and query responsiveness can be greatly improved.

1.3 Scope and Organization of the paper

OpenCQ presents an extensible architecture for Internet scale event-driven information delivery systems. We envision that the five models and their coordination policies will provide a useful reference framework for in-depth study of the efficiency, performance, scalability of distributed trigger facility and distributed event detection facility. However, due to the complexity involved, we restrict the initial system development effort to the specification of continual queries, and the design and implementation of event observation model, in particular the event detection and trigger condition evaluation upon the occurrences of events of interest. In this paper we report our initial experience with the development of OpenCQ system with the emphasis on the execution semantics of continual queries and the event observation model. We discuss the specification of continual queries and the event model only to the extend that is required for understanding of the observation model. We will report on our development of the OpenCQ object model, notification model, and resource model in other forthcoming reports.

The rest of the paper is organized as follows: We describe the concept of continual queries and outlines the execution process of a continual query in Section 2. In Section 3 we outline the OpenCQ continual query specification language and illustrate the continual query (CQ) specification, in particular CQ trigger definitions, through a number of examples. We discuss the execution semantics of continual queries, in particular the initial design of the OpenCQ event observation model, in Section 4. Section 5 presents the implementation design and implementation architecture of the first prototype of the OpenCQ continual query system. Section 6 discusses performance evaluation issues. We conclude with an overview of related work in Section 7 and a summary and directions for future work in Section 8.

2 Continual Queries

Continual queries are standing queries that monitor updates and return results whenever the updates have reached specified thresholds. A continual query consists of three key components: query, trigger, and stop condition. In contrast to ad-hoc queries in conventional DBMSs or web search engines or query systems, a continual query, issued once, runs continually over the set of information sources. Whenever its trigger condition becomes true, the new result since the previous execution of the query will be returned. The trigger part of a continual query specifies events or situations to be monitored. We distinguish primitive events from conditional (logical) events and allow events to be composed of other events. We use primitive events to model basic database operations (such as INSERT, DELETE, UPDATE), basic time events (such as at 10:00am September 15 1998, every 10 minutes, and after a hour), or signals from arbitrary processes. We use conditional events to model various conditional situations to be monitored. We provide a rich set of event composition operators (such as logic operators: conjunction, disjunction, negation; and execution dependency operators: serial, serial alternative, parallel, parallel alternative) to support composition of events.

Continual queries can be useful both to external applications and as a convenient mechanism for implementing push-based data delivery functions beyond conventional storage, retrieval, and update of data in conventional DBMSs. Some examples of pull-based functionality can be implemented in a unified way using continual queries and are described in this section.

2.1 Continual Query Concept

A continual query is defined by a triple (Q, T_{cq}, `Stop`), consisting of a normal query Q (e.g., written in SQL), a trigger condition T_{cq}, and a termination condition `Stop`. T_{cq} and `Stop` in general may depend on many different parameters, in the sequel we omit their parameters for clarity. The initial execution of a continual query is performed as soon as it is installed. The first run of Q is performed over past and present data represented by the state of information sources, and the whole result obtained by executing Q is returned to the user. The subsequent executions of Q are performed whenever a new update event occurs (is *signaled*) and the trigger condition T_{cq} becomes true. For each execution of Q, only the new query matches since the previous execution are returned to the user unless specified otherwise. Thus continual queries are defined over past, present, and future data, whereas the domain of pull queries is limited to past and present data.

Continual Semantics. Let us denote the result of running query Q on database state S_i as $Q(S_i)$. We define the result of running a continual query CQ as a sequence of query answers $\{Q(S_1), Q(S_2), \ldots, Q(S_n)\}$ obtained by running query Q on the sequence of database states $S_i, 1 \leq i \leq n$, at each given state S_i $(i > 0)$, $Q(S_i)$ is triggered by $T_{cq} \wedge \neg$`Stop`.

The basic events that cause continual queries to fire may be standard database operations such as INSERT, DELETE, UPDATE, or the events that cause clock signals (e.g., check the balance of all bank accounts at *5:00pm everyday*), or any user- or application-generated signals (e.g., a failure signal from a diagnostic routine on a hardware component). Furthermore, the trigger conditions to be monitored may be complex, and may be defined not only on single data values or individual database states, but also on sets of data objects (e.g., the total of orders of items exceeds the current inventory level), transitions between states (e.g., the new position of the ship is closer to the destination than the old position), trends and historical data (e.g., the output of the sensor increased monotonically over the last two hours).

In OpenCQ we support two types of trigger conditions: time-based trigger condition and content-based trigger

condition. Three types of temporal events are supported for *time-based* 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 every first day of the month); (3) relative temporal event (e.g., 50 seconds after event A occurred). A *content-based* trigger condition can be expressed in terms of a database query, a built-in situation assessment function, or a user-defined method. Examples include: a simple condition on the database state (e.g., execute Q whenever a deposit of $10,000 is made), an aggregate function on the database state (e.g., execute Q when a total of 1 million dollars of deposits have been made, or execute Q when the stock price of IBM drops 5%), and a relationship between a previous query result and the current database state (e.g., execute Q when a total of 1 million dollars of deposits have been made since the previous execution of Q). One extreme case of content-based trigger is immediate: report to me whenever a change to the source data occurs. In addition, composite events made up from these primitive events (e.g., the serial sequence of two events: event B occurs after event A) are also supported.

The `Stop` condition specifies the termination condition of a continual query. `Stop` conditions can be specified in terms of time-based or content-based event expressions. Both the trigger condition T_{cq} and the termination condition `Stop` are evaluated prior to each subsequent execution of the query component Q.

2.2 Continual Query Examples

We below provide two continual query examples, one uses time-based triggering event and one uses content-based triggering event.

Example 1 Given a continual query *"Report to the manager every day at 6:00pm all the banking activities of the day for those customers whose total withdraws reach $2,000"*. It is expressed as follows:

```
Create CQ banking_activity_sentinel as
Query:
    SELECT customer_id, account_no, withdraw_amount
    FROM   Account
    GROUP BY customer_id having SUM(withdraw_amount) > 2,000;
Trigger: 6:00pm everyday
Stop: 1 year (by default)
```

The trigger condition is specified by a regular time interval (everyday) and a starting time point (6:00pm).

Example 2 Suppose we have a continual query installation request *"notify me in the next six months whenever the total of quantity on hand and quantity on order of items drops below their threshold"*. This request is captured by the following continual query expression:

```
Create CQ inventory_monitoring as
Query:
    SELECT item_name, item_no, qty_on_hand, qty_on_order, threshold
    FROM Item_Inventory;
Trigger:
    qty_on_hand + qty_on_order < threshold;
Stop:  six months
```

Here are some other examples of continual queries: "tell me the flight number whenever a plane has been in this sector for more than 5 minutes", "notify me whenever IBM stock price rises by 5%", or "report to me the most recent transportation plan between port of Savannah and Fort Stewart Military Reservation, whenever there is snow, heavy rain, or any other unexpected weather changes in that region".

2.3 Continual Queries vs. ECA Rules

Continual queries, at the first glance, may seem to bear resemblance to ECA rules in active databases [4, 6, 5, 26]. One might view continual queries as a subset of ECA rules. However, they are quite different not only in functionality coverage and usage perspective but also in execution model and implementation architecture. In this section we briefly discuss the subtle differences between continual queries and ECA rules.

First, update events in ECA rules are explicitly specified by the users, whereas update events in continual queries are implicitly implied in the trigger condition, and derived by the system during the installation phase of the continual queries. Recall Example 2 given in Section 2.2, At the installation phase of the continual query `inventory_monitoring`, the update events identification module identifies three basic update events to be relevant to the trigger condition of the given continual query. They are `UPDATE qty_on_hand(item)`, `UPDATE qty_on_order(item)`, `UPDATE threshold(item)` for `item` in `Item_Inventory`. This means that any update on `qty_on_hand`, `qty_on_order` or `threshold` will signal the evaluation of the trigger condition "qty_on_hand + qty_on_order < threshold". However, using ECA rules, one may specify the follow rule:

```
Event: Update qty_on_hand(item)
Condition: qty_on_hand(item) + qty_on_order(item) < threshold(item)
Action: submit_order(item)
```

Note that this ECA rule has the same trigger condition as the continual query in Example 2. However, this rule means that the condition is evaluated only when the update on `qty_on_hand` occurs, even though the updates on `qty_on_order` or `threshold` may equally be possible to cause the violation of the trigger condition "`qty_on_hand + qty_on_order < threshold`". In short, ECA rules provide flexibility to allow users to explicitly specify what update events are of interest at will, rather than restricting the update events to those that have direct impact on the trigger condition within the same rule. Such flexibility, however, may results in passing over the situations that should be alerted according to the trigger condition.

Second, continual queries require explicit specification of termination condition. In the absence of `Stop` condition, a system default value (such as one year) will be used. Introducing termination condition as a necessary component of continual queries guarantees that alerts or update reports will send only to the right users at right time or under the specific constraints. While ECA rules terminate a rule execution by requiring users to manually delete the rule from the rule base. No system controlled termination is provided. We consider the support for system-controlled termination condition as a desirable and practical capability for an event-driven active information delivery system.

Thirdly, although situation monitoring is one of the canonical applications of ECA rules, they are designed as building blocks for general purpose active database systems or production rule systems [38] in centralized data management systems, whereas continual queries are specifically designed for update monitoring in distributed event-driven information delivery systems. Continual queries emphasize on effective and specialized support for personalized update monitoring. For readers who are familiar with the program specialization systems [37, 28], popular in OS and PL communities, there is a close analogy between continual queries and program specialization.

If we view program specialization as a means for improving program performance and obtaining better control of program behavior and consistency, then continual queries can be seen as an interesting and effective specialization of ECA rules, which aims at providing more efficient support for personalized update monitoring in distributed open environments such as the Internet.

Last but not least, actions in ECA rules can be update events which may in turn trigger the same rule again, directly or indirectly (i.e., the cascading effects of rules); whereas actions fired, when the trigger condition of a continual query is evaluated to be true, are restricted to the execution of the same query expression, the change notification functions, and the methods to compute the differential result; They are side-effect free actions with respect to the data set over which the trigger condition and query component are defined. This feature simplifies many complex issues in ECA Rules, especially those related to the consistency and concurrency issues in advanced transaction management. Such simplification allows us to focus on addressing the issues that are specific to update monitoring and push-based information delivery. We provide a further discussion on active databases and other related work in Section 7.

3 Continual Query Specification

We have implemented OpenCQ continual query system, based on the concept of continual queries introduced in Section 2. In what follows, we will describe how continual queries are defined (in this section), installed (activated), and executed (in Section 4).

3.1 Specification Semantics

Continual queries, like all other forms of data, are treated as first class objects. There is a continual query entity type, and every continual query is an instance of this type. The difference between continual query entity type and other entity types is that OpenCQ understands the semantics of continual queries and invokes a particular operation – *fire* automatically. The functions that define the key components of the structure of a continual query are:

- **Continual query identifier** (cqid). Like any other entity, each continual query (CQ) has a unique entity identifier. Such identifier is generated by the system after the installation of the CQ is successful and the first run of the CQ is fired.
- **Continual query name.** This is a user-defined and optional attribute.
- **Trigger** condition. The trigger component specifies the event that causes OpenCQ to fire the subsequent executions of the continual query (CQ). Typed formal parameters may be defined for the event; these parameters are bound to actual arguments when the next execution of this CQ is *fired*.
- **Stop** condition. The `Stop` condition specifies the termination semantics of the continual query. It is described by an event expression. Both time-based events and content-based events can be used.
- **Query** component. The query is one of the side-effect free action to be executed when the trigger condition is evaluated to be true and the `Stop` condition is not met. The execution coupling mode between the trigger condition and the query action can be specified explicitly at the continual query installation time to override the system default (see Section 4 for further detail).

Both `Trigger` and `Stop` conditions are specified in terms of event expressions. We distinguish between primitive events, composite events, and conditional events. A primitive event is either a basic update event (such as `UPDATE qty_on_hand(item)`) or a temporal event (such as every Monday, 9:00:00pm, March 2, 1998). A conditional event is a conjunction or a disjunction of events, of which at least one of the component events is a conditional event. An atomic conditional event is an event of the form `attribute_name <comparison_op> value`, such as "`stock.price > 100`". A composite event is defined by an event composition expression following the BNF syntax below:

```
composite_event ::= <element_event> <event_op> <composite_event>
element_event   ::= <primitive_event> | <atomic_conditional_event>
primitive_event ::= <basic_update_event> | <temporal_event>
atomic_conditional_event ::= <attribute_name> <comparison_op> <value>
conditional_event  ::= <atomic_conditional_event> <logic_op> <conditional_event>
basic_update_event ::= <db_operations> | <external_signals>
logic_op       ::= <conjunction> | <disjunction> | <not>
comparison_op  ::= <string_op> | <algorithmic_op> | <built_in_op> | <user_defined_op>
temporal_event ::= <absolute_time> | <regular_interval> | <irregular_interval>
db_operations  ::= UPDATE | INSERT | DELETE
event_op ::= <logic_op> | <user_defined_op> | <system_built_in_op>
```

A complete BNF description of the OpenCQ event specification language and the formal semantics of continual query specification model, including the specification of primitive and composite events, and the algorithm for decomposing the trigger condition components into basic update events and conditional events, are beyond the scope of this paper. Readers who are interested in further details may refer to our technical reports [23, 35].

3.2 Specification Syntax

Syntactically, continual queries are defined by specifying trigger condition components in the SQL-like `FROM` and `WHERE` clauses, by specifying `Stop` condition in temporal event expressions, and by specifying query components in the SQL-like `SELECT-FROM-WHERE` clauses. Users may give each of their continual queries a meaningful name (such as `banking_activity_sentinel` in Example 1). Continual queries may be defined across over a set of data sources that are autonomous and possibly heterogeneous in nature. These data sources may be structured, semi-structured, or unstructured. Mediators and wrappers are used to decompose the query or trigger condition according to the number of data sources used to evaluate the query or the trigger condition. Details for distribution aspect of the query processing and trigger condition evaluation are beyond the scope of this paper, and will be addressed in a forthcoming technical report.

We below provide some examples of continual queries written in SQL-like expression enhanced with user-defined or system built-in functions. We first define a continual query `weather_watch` that monitors weather condition updates in the region from port of Savannah in Georgia to Fort Stewart Military Reservation every 20 minutes and send mail to Todd 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 Web site (`www.nws.nova.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:

```
Creat CQ Savannah_weather_watch as
```

```
Query:   SELECT *
         FROM www.wns.nova.gov
         WHERE location like% 'Savannah' AND state = 'Georgia';
               OR location like% 'Fort Stewart';
Trigger: 20 minutes;
Stop:    1 year (default).
```

This continual query specifies the request for monitoring updates on weather conditions at 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 sending mail is implicitly inferred by the system, based on either the fact that Todd is the owner (creator) of this continual query `Savannah_weather_watch` or the fact that the creator of this continual query has entered a special request that the update results be sent also to his/her manager, Todd, at the CQ installation time.

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 (recall Example 2). Here is an example where the trigger condition is defined over a set of object classes that are different from those over which the query component is defined:

```
Create CQ Transportation_Route re-planning as
Query:
   SELECT plan_no, plan_desc., plan_alt_routes
   FROM   Transportation_plan
   WHERE  plan_route like 'Savannah to Fort Stewart';
Trigger:
   FROM  www.wns.nova.gov
   WHERE location like% 'Savannah' AND state = 'Georgia'
         OR location like% 'Fort Stewart';
Stop:  next 3 months.
```

This continual query amounts to saying that "monitoring the weather condition between port of Savannah and Fort Stewart 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". Note also that the `Transportation_plan` may be stored in a relational DBMS (e.g., Oracle), a structured data source, and the weather information is available from the NWS Web site, a semi-structured data source.

Another interesting feature of OpenCQ 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 OpenCQ system built-in functions for trigger specification include increased by x percent, denoted as `IncreaseBy%`$(X) \leq x$, and decreased by y percent, denoted by `DecreaseBy%`$(Y) \leq y$, where X and Y are field names of the source data items. Using these system built-in functions, the continual query, "notify me in the next two weeks whenever the stock price of Bayer drops by 5%", can be expressed conveniently as follows:

```
Creat 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: company_name = 'Bayer AG' AND DecreaseBy%(stock_price) >= 5%;
Stop:  9:00:00 am, Oct. 26, 1998
```

Generally speaking, in specifying a continual query, the `Query` clause, `Trigger` condition clause, and `Stop` condition clause are essential and thus mandatory. When there is nothing entered for the `Stop` condition, a default value (e.g., two weeks) is used. When nothing is filled in the `Query` clause, an error message is generated. When the trigger condition is empty, the default is set to a time-based trigger at a default time interval (say everyday). In addition, one can specify other optional properties for a continual query, such as 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.

4 Continual Query Execution Semantics

We have explained how one defines continual queries in the previous section. We now describe the implementation of how the OpenCQ system triggers and executes continual queries.

It is well known that in a conventional pull-based DBMS user application programs are executed when explicitly requested to do so. Execution of such programs typically results in the processing of a sequence of *transactions*, where each transaction is a unit of consistency and recovery. The system guarantees *atomicity* (all updates issued by the transaction are installed in the database or none are), *serializability* (the concurrent interleaved execution of a set of transactions is equivalent to a serial no-interleaved execution), and *durability* (once a transaction is committed, its updates will never be rolled back). In contrast, a continual query system must evaluate installed continual queries under system control (not user or application control). More concretely, once a continual query is installed, the system must decide how to detect the update events of interest, how to evaluate the trigger condition, and when to fire the subsequent execution of the query component, and how should the execution of these tasks be treated with respect to user transactions? This section is is an attempt to answer these questions.

4.1 A Quick Look at Continual Query Execution

Before entering the details on the execution semantics of continual queries, let us first take a quick look at the execution process of a continual query.

Recall the continual semantics described in Section 2.1, it specifies that, for each continual query CQ_i, denoted by $(Q, T_{cq}, \texttt{Stop})$, the first execution of CQ_i is activated by the installation of the CQ_i, without going through the evaluation process of the condition $T_{cq} \wedge \neg\texttt{Stop}$; whereas the subsequent executions of this continual query are fired only when the condition $T_{cq} \wedge \neg\texttt{Stop}$ becomes true. More concretely, when a continual query CQ_i is entered (installed) the first time, the following activation actions take place:

- CQ_i is registered with a unique continual query identifier (cqid);
- For the first run of CQ_i, the query component Q will be modified by rewriting Q to include the portion of the trigger condition T_{cq}, which is defined over the same set of object classes as the query component Q. Let us refer to this modified query component as Q';

- Rather than activated by the condition evaluation manager, the first run of CQ_i is fired by the continual query activation manager (see Section 4.3 for further detail), which executes the modified query component Q'. There is no verification on trigger condition or stop condition. The first run of CQ_i will return the whole answer of Q', and cached the answer as the previous execution result of CQ_i;
- The trigger condition T_{cq} is activated in the sense that the update events of interest are identified, each associated with a conditional event; and the trigger activation variables (such as transaction coupling mode, dependency coupling mode, schedule coupling mode, and execution coupling mode) are initialized.

The subsequent runs of CQ_i will be fired whenever the trigger condition T_{cq} is evaluated to be true, and the termination condition `Stop` is not expired. Each subsequent execution of CQ_i proceeds as follows:

- Step 1: *Update Events Identification*
  This step is to identify the update events of interest from the trigger condition expression of CQ_i. It is done by decomposing the trigger condition T_{cq} into a list of T_{cq} triplets, each triple consists of a basic update event, an atomic conditional event, and a connector to the next triple in the list;
- Step 2: *Update Events Detection*
  This step is to decide when to detect the changes and what to detect for the given trigger condition, and which event detectors should be used. For each triple generated in the Step 1, the atomic condition is evaluated when the basic update event is *signaled*;
- Step 3: *Logical Events (Condition) Evaluation*
  This step is carried out by the condition evaluation manager, which first select a triplet from the list of T_{cq} triplets generated in Step 1, if the connector is an AND connector (or an OR connector), the AND logical event detector (the OR event detector) is invoked; if the connector is WHERE, the next triplet in the list will be used as an add-on condition to the basic update event component of this triplet; and so on. For further detail, see Section 4.4;
- Step 4: *Differential Query Execution and Result Delivery*
  If the condition evaluation in step 3 returns a *true* value, then the following pre-defined actions are scheduled to execute: (1) fire the next execution of the query component Q, (2) compute the difference between the current run of Q and the result of the previous run, (3) notify the user of the arrival of new updates of interest, and (4) deliver the differential result to the user.

A walkthrough example is provided in Section 4.4 to illustrate this process.

4.2 Basic Coupling Modes

Continual queries in practice are often defined over multiple, autonomous and possibly heterogeneous data sources. The local update transactions are usually orthogonal to the continual queries specified over the same set (or a subset) of data. Furthermore, both trigger condition evaluation component and query component of a continual query are side-effect free transactions. Due to the autonomy and distribution of data sources and the side-effect free nature of continual queries, it is not only important but also practical to allow a more flexible execution model.

A flexible execution model allows trigger condition evaluation and query execution to be broken off into different execution threads from the triggering transaction (the transaction that carried out the update operations). More

concretely, it should be possible to allow the continual query evaluation to be separated from the (triggering) transaction that carried out the actual updates. This would allow the triggering transaction to commit earlier, and would potentially increase concurrency and reduce wasted work (rollback of incomplete transactions after a crash). The OpenCQ execution model for continual queries uses the notion of coupling modes to provide this flexibility.

In OpenCQ we support four basic coupling modes: transaction coupling mode: *separate* or *same*, execution coupling mode: *asynchronous* or *synchronous*, dependency coupling mode: *causally dependent* or *causally independent*, and schedule coupling mode: *immediate* or *deferred*. We view the execution model of each continual query to consist of the following four participating transactions:

- (1) the triggering transaction that carries out the update operations,
- (2) the update event detection transaction that detects if the data of interest has been updated,
- (3) the trigger condition evaluation transaction that evaluates the condition based on the newly updated data, and
- (4) the transaction that carries out the subsequent execution of the query component and sends out the alerts or change notification messages.

Such arrangement provides more flexibility for utilizing multiple execution threads and parallel execution for continual query processing, which are critical techniques to the effectiveness and responsiveness of an event-driven distributed information delivery system.

In OpenCQ, it is possible that the coupling case for transaction types (1) and (2) may be different from the coupling case for transaction types (2) and (3) as well as the coupling case for transaction types (3) and (4).

We illustrate the meanings of each coupling mode using the coupling scenario for transaction types (2) and (3), which relates to the trigger condition part of the continual queries. For the trigger condition part of a continual query, the coupling mode specifies when the condition is to be evaluated relative to the triggering event (i.e., the update event being monitored):

- **Transaction coupling mode**: separate or same

  The transaction coupling mode *separate* means that the condition evaluation triggered by the update event runs as a separate transaction with respect to the transaction that detects the update events of interest.

  The transaction coupling mode *same* means that the condition evaluation triggered by the update event runs either as part of the transaction for detecting the update event in the case that the updates performed by the triggering transaction are local operations, or as part of the triggering transaction in the case that the updates are performed by the same user or application program who installed the continual query.

- **Execution coupling mode**: asynchronous or synchronous

  The asynchronous coupling mode means that the update event detection transaction may run in parallel with the trigger condition evaluation transaction.

  The *synchronous* coupling model means that if the trigger condition evaluation transaction is triggered by the transaction that detected the update events, then the trigger condition evaluation transaction is executed, and the execution control returns to the 'triggering' transaction only after the condition evaluation transaction is committed.

- **Dependency Coupling Mode**: casually dependent or casually independent

  The *casually dependent* coupling mode means that the trigger condition evaluation transaction can be scheduled only after the 'triggering' transaction that detected the update events has committed.

  The *casually independent* coupling mode means that the scheduler is free to schedule the trigger condition evaluation transaction independently of the update event detection transaction when the update transaction is local.

- **Schedule Coupling Mode**: immediate or deferred

  The schedule coupling mode *immediate* means that the trigger condition evaluation transaction is fired as soon as the triggering transaction commits. When the updates are carried out by a global update transaction issued by the same user or application program, the triggering transaction refers to this global update transaction. When the updates are carried out by local transactions or other remote and autonomous transactions, the triggering transaction refers to the update event detection transaction.

  By looking into the semantics implication of these coupling modes, We come to the following conclusion: The schedule coupling mode *deferred* must be used in conjunction with the *same* transaction coupling mode. This mode means that the CQ trigger condition evaluation is fired at the end of the update event detection transaction and before it commits.

The *same* transaction coupling mode can be used only in conjunction with synchronous execution coupling. The *deferred* schedule mode is applicable only in conjunction with the *same* transaction coupling mode. However, the immediate schedule mode can be used in conjunction with both *same* and *separate* transaction couplings. Also both dependency couplings are applicable only to *separate* transaction coupling, *immediate* schedule coupling, and *asynchronous* execution coupling.

In a similar manner, we may illustrate the possible coupling cases for transaction types (1) and (2), the event detection part of the CQ, and for transaction types (3) and (4), the query scheduling part of the CQ. For the query scheduling part of a CQ, each coupling case specifies when the subsequent run of the query component is to be fired relative to the trigger condition evaluation transaction.

In OpenCQ we allow users to define their application-specific coupling modes for any of the three pairs of the participating transaction types. In the absence of user-specified coupling modes, the system default coupling case will be used. The default coupling modes between transaction (1) and transaction (2) are separate, asynchronous, causally independent. The default coupling modes between transaction (2) and transaction (3) are separate, synchronous, causally dependent, and so are the coupling modes between transaction (3) and transaction (4).

4.3 Continual Query Installation

Once a continual query CQ_i, denoted by $(Q, T_{cq}, \texttt{Stop})$, is defined, the user may install it directly to the OpenCQ continual query system. At the installation time, the `Install` module of the client manager takes the continual query and passes it to the OpenCQ continual query server. The server activates it using the `activate` command. The activation process consists of the following three main tasks:

- making this continual query a persistent object and generating a unique identifier (`cqid`) for it;
- Modifying the expression of the query component to incorporating the trigger condition semantics. This task is accomplished by checking if the trigger condition component T_{cq} and the query component Q are

defined over the same set of data, i.e., DataSet(Q) = DataSet(T_{cq}), where DataSet(Q) is the set of instance variables used in Q and DataSet(T_{cq}) is the set of instance variables used in T_{cq};

  - if yes, merge the trigger condition into the `WHERE` clause of the query component Q, and denote the modified query expression as Q', execute Q' instead of Q for the first run of CQ_i, and cache the answer as the previous run result;
  - if not, identify if there is a common part of the data set shared by T_{cq} and Q, i.e., checking if DataSet(Q) $\cap$ DataSet(T_{cq}) $\neq \emptyset$, if yes, merge the common portion of the T_{cq} into the query component Q, and denote the modified query component by Q'; otherwise, let $Q' := Q$; then perform the following two actions: (1) execute Q' for the first run of CQ_i and cache the answer as the previous run result of the query component; (2) fetch the other portion of T_{cq}, i.e., DataSet(T_{cq}) $-$ DataSet(Q), and cache the result for the subsequent trigger condition evaluation of the CQ_i;

- Initializing the execution attributes and data structures used for event detection and condition evaluation of this given CQ. This task includes decomposing the user-specified CQ trigger condition into a set of triplets, each triple is described by a basic update event, an atomic conditional event, and a connector; and setting up the initialization for the transaction coupling mode, the dependency coupling mode, the schedule coupling mode, and the execution coupling mode (recall Section 4.2).

The `Activate` command also returns a handle that will be used to deactivate this continual query when its termination condition is expired.

Users can use the `activate` command to define the coupling modes according to application specific requirements. The syntax of the `activate` command is given below:

```
Activate <cqid>
  define communication protocol between
      <trans1> and <trans2>
  TransactionCoupling = same | separate
  ExecutionCoupling = synchronous | asynchronous
  DependencyCoupling = causally dependent | causally independent
  ScheduleCoupling = immediate | deferred
```

Once a continual query is activated, it runs continually following the communication protocol defined by the specific coupling case. The continual query is terminated when its `Stop` condition is evaluated to be true. To terminate an installed continual query, the command `Deactivate <cqid>` is invoked, which removes from the OpenCQ system catalog the corresponding continual query object identified by `cqid`, deactivates the related event detectors that are still active, and sends to the owner of this CQ a notification that this CQ is expired.

4.4 Event Detection

The event detection model is also called event observation model. It defines the mechanisms by which event occurrences and patterns of event occurrences are observed. There are two classes of event observation methods:

- *Synchronous observation*, in which the fact of an event occurrence is communicated explicitly to and in synchronization with the event observer. The database triggers built-in as components of commercial RDBMSs such as Oracle, Informix, Sybase are examples of this kind.

- *Polling*, in which the observer periodically checks for the occurrence of an event.

The main task of 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 types of the events to be detected. As discussed in Section 3.1, the trigger condition part of a continual query may be a primitive event, such as a *temporal event*: every two days or every first day of the month; an *atomic conditional event*: the stock price is greater than 100 (`price > 100`); or a *composite event*, which is formed by an event composition expression of the form "E_1 `<event_op>` E_2", where E_1 and E_2 are primitive or composite events. Typical examples of composite events are

```
Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5
keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'
qty_on_hand(item) > threshold(item)
qty_on_hand(item) + qty_on_order > threshold(item)
```

Each primitive event is detected by using a primitive event detector, which is either a basic temporal event detector or an atomic conditional event detector. An operation *signal* is defined for the event entity type, and is executed by the event detector components of the system.

4.4.1 Time-based Event Detection

For time-based continual queries, a temporal event detector, or so-called time-based event detector, is used, which translates the time-based trigger condition into a clock event and installs the clock event script to the OpenCQ clock daemon. Whenever the clock event occurs, the trigger condition is signaled. Thus the subsequent execution of the query component is fired. A distinct feature of time-based continual queries is the use of *user-controlled polling* for update monitoring.

There are two key implementation techniques useful for time-based event detection: The first technique is to design a generic transformation program that takes the user-defined time condition and transforms it into a clock event expressed in the clock event scripting language; the clock manager (daemon) will then take over the control and trigger the update event detection according to the clock event installed; whenever the update event is signaled, the continual query manager will call the query evaluator to fire the subsequent run of the query component, and call the change notification manager to deliver the change notification message as well as the update result. The second technique is to develop a clock event manager which, on one hand, provides a scripting language to allow users to specify an arbitrary clock event and the action to be taken if the clock event occurs, and on the other hand, provides triggering capability so that it can fire the specified action (e.g., invoke a program) when a specific clock event is signaled.

The implementation of a clock manager is a system-specific decision. One may either choose to design a clock manager specifically for this purpose, or reuse the clock manager provided by an operating system (such as Cron by Unix and Scheduler by NT). In the first prototype of the OpenCQ system, we make use of Cron as the clock manager. We are considering to write our own clock manager in the next prototype release to further enhance the efficiency of the system.

4.4.2 Content-based Event Detection

In contrast to time-based continual queries, the content-based continual queries use the *system-controlled polling* or *synchronous* approach for update monitoring. Thus, there are more than one strategies possible for implementation of the CQ trigger condition monitoring and event detection.

In order to carry out the content-based event detection, the first thing we need to do is to identify what update events are of interest to the given continual query. As mentioned in the continual query activation procedure (recall Section 4.3), for each installed continual query (Q, T_{cq}, `Stop`), its trigger condition T_{cq} is decomposed into a list of T_{cq} triplets, each triple is described by a basic update event, an atomic conditional event, and a connector. For example, if the trigger condition is "`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`", then the following triplets are generated:

```
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = IBM, OR)
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = Intel, END)
```

For the trigger condition: `qty_on_hand(item) > threshold(item)`, two triplets are generated. They are: `(qty_on_hand, true, >)` and `(threshold, true, END)`. Note that the connector `WHERE` means that the next triple is not an update event of interest but a constraint on the current update event. In this case, UPDATE on the stock price is the event we would like to monitor, and the condition `Stock.company = IBM` is simply a constraint, saying that we are only interested in monitoring UPDATE on the stock price of IBM but not other companies' stock prices.

Now we can determine **what to detect** based on the basic update events identified by the list of T_{cq} triplets.

Example 3 Given the trigger condition:

"`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`",

the basic events of interests are UPDATE operations on `Stock.price` and `Stock.company`, as well as INSERT and DELETE operations on the object class `Stock`. For trigger condition `keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'`, if the condition field name `keyword` is mapped to `Document.title` and `Document.abstract` available at the corresponding data source(s), then the basic events of interests are INSERT and DELETE operations on `Documents` objects, and UPDATE operations on `Document.title` and `Document.abstract`.

The next question is **how to detect**, namely we need to decide which mechanisms may be used to detect the changes made by the update operations, possibly from some transactions that are local to the data source; In OpenCQ, we distinguish between the data sources that have built-in trigger capability such as the data sources managed by trigger-enabled RDBMSs (incl. Oracle, DB2, Informix, Sybase) and the data sources that have no built-in trigger capability such as most of the web sites and file systems.

- For the data sources with built-in trigger facility, the OpenCQ system may install the database triggers on the data columns or objects of interest. Whenever there is an update, the database transaction that carries out this update will send an update signal to the corresponding CQ wrapper. We provide the host-specific trigger installation program (such as Oracle trigger installation program) to install triggers on those data objects and data columns that are accessible to the OpenCQ continual query system.

- For the data sources with no built-in trigger facility, we use system-controlled polling with system-defined interval (such as every 30 seconds). We detect the events of interest by comparing the two snapshots generated from two different polling time points, find what has been inserted, or modified, or removed. The richness of the update operations to be used in the difference computation varies among different *Diff* programs [13, 15, 8, 40]. The current implementation of the OpenCQ uses the Xdiff program which compares two XML snapshots. The detail of the Xdiff design is beyond the scope of this paper.

Important to note is that the capabilities of database trigger supported in commercial DBMSs today are not sufficient, particularly in those cases where remote installation of customized database triggers is required. In these situations, a system-controlled polling will be used in conjunction with the database triggers. Our experience tells that not all the RDBMSs allow database triggers to be installed by a remote program through JDBC. In the first prototype of OpenCQ, we implement the content-based event detection using the system-controlled periodic polling for those database sources that do not provide built-in trigger facility in their APIs for remote access (e.g., JDBC or Oraperl for Oracle database sources).

Now, let us walk through the event detection process. Given a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Suppose that the trigger condition T_{cq} has been transformed into a list of T_{cq} triplets, denoted by TripleSet($\texttt{cqid}, T_{cq}$). To simplify the steps (that) we need to walk through, let us assume that the connectors we use in this walkthrough are the most commonly used ones, namely `WHERE, AND, OR, END`. For each triplet in TripletSet($\texttt{cqid}, T_{cq}$), we form a event detection query, denoted by Q_{detect}, which is to be submitted to the relevant data sources to detect if an update is occurred.

- For a triple of the form $(T.A, T.A\vartheta v, \texttt{AND})$ or $(T.A, T.A\vartheta v, \texttt{OR})$ or $(T.A, T.A\vartheta v, \texttt{END})$, where T denotes the object class, A, B are instance variables of T, and ϑ is the comparison operator, let `prev` denote the value of instance variable A contained in the result of previous execution of the given CQ. Thus, the corresponding event detection query Q_{detect} is expressed as `SELECT A FROM T where A` $\neq$ `prev`.
- For a triplet of the form $(T.A, T.A\vartheta w, \texttt{WHERE})$, we fetch the next triple, say $(S.B, S.B\vartheta w, \texttt{END})$ from the remaining list of TripletSet($\texttt{cqid}, T_{cq}$). Thus, the event detection query Q_{detect} is expressed as `SELECT T.A, S.B FROM T, S WHERE S.B` ϑ `w AND T.A` $\neq$ `prev`.

4.5 Condition Evaluation

In principle, one may want to detect all the update events of interest before starting the trigger condition evaluation process. In practice, the CQ trigger condition evaluation is carried out in conjunction with the process of basic update event detection, to guarantee the efficiency of the condition evaluation. For example, if a condition is of the form $(T.A\vartheta v_A) \wedge (T.B\vartheta v_B)$, and if the event detection query over the triplet $(T.A, T.A.\vartheta v_A, \texttt{AND})$ returns empty answer, then we can conclude that the trigger condition is false without looking into the second triplet $(T.B, T.B\vartheta v_B, \texttt{END})$.

Now, let us walk through the condition evaluation process for a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Let TripleSet($\texttt{cqid}, T_{cq}$) denotes the list of T_{cq} triplets generated by the CQ activation process. Simiar to the discussion on event detection, we simplify the steps we need to walk through by assuming that the connectors used in this walkthrough are `WHERE, AND, OR, END`. The condition evaluation process of CQ_i proceeds as follows:

- Step 1: It starts by selecting a triple in TripleSet($\texttt{cqid}, T_{cq}$), and then check the connector type of this triple:

- Step 2: if it is an `END` connector, then this content-based trigger condition is evaluated to be true, and the subsequent query execution is fired.
- Step 3: if it is an `WHERE` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{WHERE})$, and the next triplet is fetched from the remaining list of TripleSet(`cqid`, T_{cq}), denoted by $(S.B, S.B\vartheta w, \texttt{AND})$, then the update event detection query Q_{detect} is expressed as `SELECT T.A, S.B FROM T, S WHERE S.B` ϑ `w AND T.A` $\neq$ `prev`. If Q_{detect} returns a non-empty answer, it means the update event has occurred; go to step 6. If Q_{detect} returns an empty answer, we can conclude that the corresponding trigger condition is false.
- Step 4: if it is an `AND` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{AND})$, then the update event detection query Q_{detect} is expressed by `SELECT T.A FROM T where T.A` $\neq$ `prev`. If the answer to this query Q_{detect} is empty, then the condition evaluation is false. Otherwise (i.e., if the answer is non-empty), go to Step 6.
- Step 5: if it is an `OR` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{OR})$, then the update event detection query Q_{detect} is the same as the case for an `AND` connector, i.e., `SELECT T.A FROM T where T.A` $\neq$ `prev`. However, unlike the `AND` connector case, if the answer to this query Q_{detect} is non-empty, then we conclude that the condition evaluation is true. Otherwise (i.e., the answer is empty), we need to go to Step 6.
- Step 6: select another triplet from the remaining list of triplets in TripleSet(`cqid`, T_{cq}), and go back to Step 2.

Obviously, the richer set of event composition operators is used, the more sophisticated the event detection process will be. A complete description of event composition operators and their formal semantics is beyond the scope of this paper. Readers may refer to [23] for further details.

4.6 Issues on Efficient Condition Evaluation

Users and application programs may define as many continual queries as they wish. Once these continual queries are installed, they run continually as long-running side-effect free transactions with checkpoints [1]. Despite all the query components, each from one installed continual query, the set of all trigger conditions forms a potentially large set of predefined queries (i.e., event detection queries) that have to be evaluated efficiently. Furthermore, the trigger condition component of a continual query may be more sophisticated than the query component when the update monitoring threshold is defined over several different object classes and uses special operators (such as `IncreaseBy%`) that are not supported by the data sources upon which the condition is evaluated. Several techniques have been identified as being useful for performance optimization of the condition evaluation:

The first technique is *Multiple Condition Optimization* and also called multiple query optimization in the literature [34]. This technique represents conditions (and the events that signal the condition evaluation) by condition evaluation graphs, which resemble the query graphs commonly used in query processing. The leave nodes of the graph are triples of the form $(R, R+, R-)$, where R corresponds to a set of entity instances before the update, $R+$ corresponds to the set of instances inserted into R by the update, and $R-$ the set of instances deleted from R by the update. The internal nodes correspond to operators of some convenient algebra into which the query language can be compiled (e.g., select, project, join). The key idea of multiple condition evaluation consists of identifying common subgraphs, and evaluating these subconditions once for a whole set of queries, instead of

[1] Each time when the trigger condition is evaluated to be true and the query is fired is referred to as a checkpoint.

once for every query [30, 34]. For a continual query system, the common subconditions may be detected at the algebraic level due to the distribution and autonomy of data sources, whereas in a centralize data base system the common subconditions may also be detected at the lower level (e.g., use common access paths). The multiple query evaluation problem is complicated by the need to ensure that the conditions will have to be evaluated simultaneously; e.g., they are triggered by the same update event.

The second technique is *Incremental Condition Evaluation*. A main task of continual query evaluation is to determine whether the answer to a previous execution of the query component (say at time t) has changed as a result of some update event to some of the query's operands at time t'. Let Q be a query defined over an entity set R, and $Ans(Q,t)$ be the answer to the query Q at time t. Let $R' = (R\ minus\ R{-}\ union\ R{+})$. A brute force method for computing the change in $Q(R,t)$ would be to compute $Ans(Q,t') = Q(R')$, and then the symmetric difference of $Ans(Q,t)$ and $Ans(Q,t')$. Incremental evaluation computes this symmetric difference directly from $R+, R-$, and Q. Sometimes R is also needed when Q involves joins [24]. Many algorithms have been proposed in view materialization research for incremental maintenance of materialized views (see Section 7 for reference), and may be directly deployable for incremental condition evaluation in the continual query systems.

An extreme case of incremental condition evaluation is the situation where it may be possible to infer that there is no change in a query's answer with respect to an update event without evaluating the query. Put differently, we can ignore an update event E at t' with respect to the execution of query Q at t, if we can tell that the symmetric difference between $Ans(Q,t)$ and $Ans(Q,t')$ is empty by looking only at the update event E and query expression Q. A trivial example is the update event that modifies a data object that is irrelevant to the query Q. A less trivial example is an update that modifies the Intel stock price to a higher value; clearly, this update event is ignoble with respect to the trigger condition `stock.price(Intel) DecreaseBy% 5`.

Also more opportunities for optimization may arise out of the interplay between the event detection, the condition evaluation, and the subsequent execution of the query component. Generally speaking, more work is needed to develop heuristics and cost models that the condition monitor can use to explore the tradeoffs and benefits of these tactics and algorithms.

5 System Architecture

The OpenCQ continual query system deals with a new class of objects: continual queries. The architecture for such a system adds functions to the data/object manager and possibly transaction manager, and introduces some new components: a continual query manager, a condition evaluator and a variety of event detectors for the different types of events to be detected (recall Section 4). The data/object manager must now support the definition, storage, and retrieval of continual queries. The transaction manager is expected to implement the execution model, including the support for nested transactions, various coupling modes, and causality constraints.

OpenCQ proposes a three-tier architecture: client, server, and wrapper/adapter. This architecture is motivated by the need for providing efficient support to composite event detection and complex condition monitoring of installed continual queries, and the need for sharing information among structured, semi-structured, and unstructured remote data sources. A sketch of the OpenCQ system architecture is given in Figure 1.

The client tier currently has four components: (1) The form manager that provides the CQ clients with fill-in forms to register and install their continual queries; (2) The registration manager which allows clients to register the OpenCQ system with valid user id and password, and return the clients a confirmation on their registration; (3) The client and system administration services which provide utilities for browsing or updating

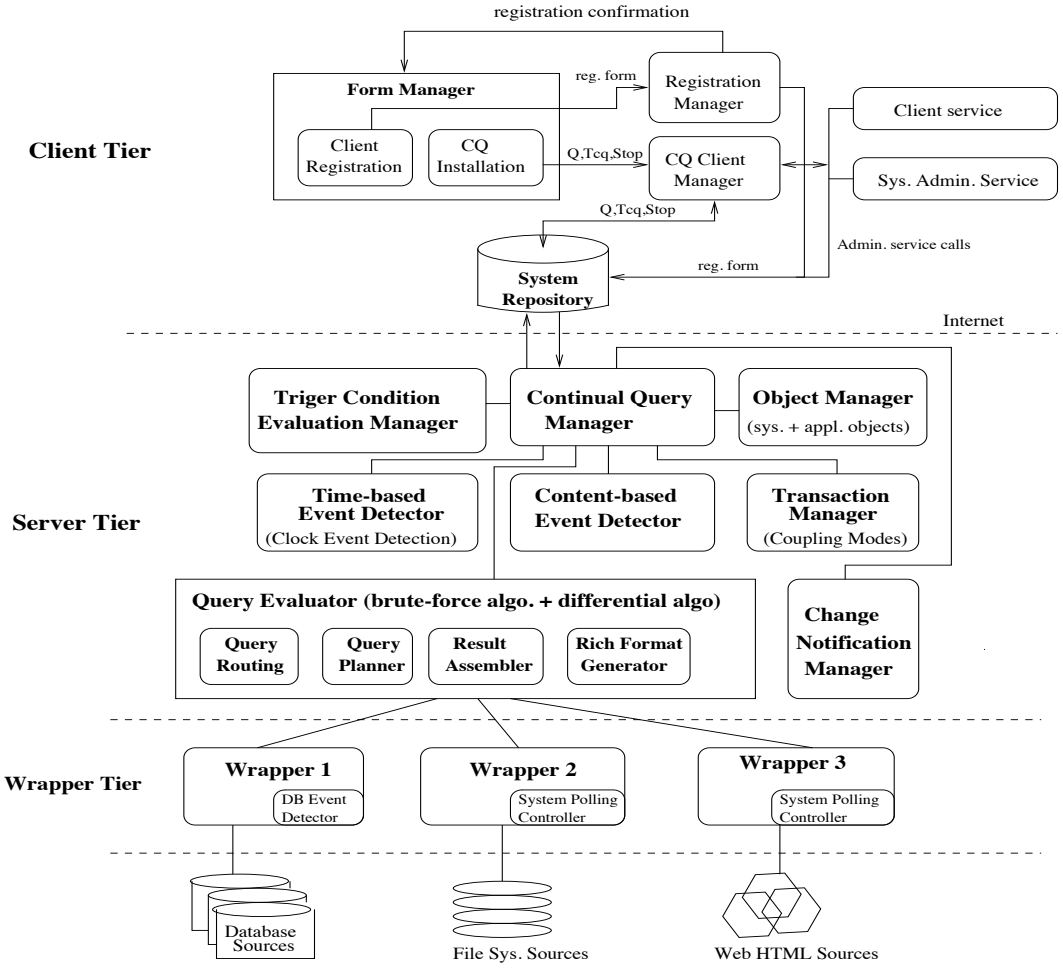

Figure 1: Architecture of an event-driven Continual Query system

installed continual queries, for testing time-based and content-based CQ triggers, and for tracing the performance of update monitoring of source data. (4) The Client manager which coordinates different client requests and invokes different external devices. For instance, once a continual query request is issued, the client manager will parse the form request and construct the three key components of a continual query (Q, T_{cq}, `Stop`), before storing it in the OpenCQ system repository. Although not a direct part of the OpenCQ development, one could imagine value-added update monitoring services such as posting a continual query request in natural language through typing, voice or hand-writing or multi-modals combined. Recall the example given earlier: "*notify me whenever IBM stock price rises by 5%*". By hooking up the CQ client with an English language text recognizer, or hand-writing recognizer, or voice recognizer, we can parse this request and automatically generate the query, the CQ trigger, and the `Stop` condition for this request. The results can be returned to the user also by multiple modes, such as by email, fax, phone, bulletin posting, or displaying signals on users' desktop screens.

The second tier is the OpenCQ server which consists of three main components: a continual query (CQ) manager with event-driven delivery, a trigger condition evaluation manager, and the event detectors (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 detectors to monitor updates of interest, and coordinate with OpenCQ 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 the update events of interest are detected and signaled by the event detectors. We build the time-based event detector on top of the Cron clock event manager in the first prototype of OpenCQ. The content-based event detector is built based on the primitive event detector and a set of specialized event detectors, each designed for a particular event composition operator.

In addition, the OpenCQ 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}, `Stop`). The key components of this query evaluator include: the query router [22, 20], the query planner [19, 21] and the query result assembler. The query router is a key technology that enables the OpenCQ continual query system to scale up in order to handle 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 relevant to the user's request. Consequently, instead of contacting all the available data sources, the CQ evaluator only contacts the selected sites that can actually contribute to the query.

The third tier is the CQ wrappers/adapters tier. The CQ manager, on behalf of the event detectors and the query evaluator, talks to each information sources using a CQ wrapper. A wrapper is needed for each data source because individual data sources may have different ways of accessing data and different formats for representing query results. The expected functionality of an ordinary wrapper is to translate the query into the format understood by the remote site. As the result comes back, the wrapper packages (translates) the response from the corresponding data source site into the OpenCQ object format used by the CQ system. In addition to the common data wrapping capability, a CQ wrapper installs the source-specific event detector (a database trigger detector for RDBMS sources, a system-controlled polling event detector for data sources with no built-in triggers on update operations), which, on behave of the OpenCQ 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.

Depending on the need of applications, the client tier, the wrapper tier, and the server tier, including the continual query manager, the trigger condition evaluation manager, and the variety of event detectors, as well as the query evaluator, could all be located on a single host machine, or distributed in different combinations among several computers connected through local or wide area networks. OpenCQ 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 OpenCQ 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 OpenCQ's components and interfaces is beyond the scope of this paper. However, we below walk through each of the main components of OpenCQ 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 subtransaction 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 subtransaction.
- 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 subtransactions for each of these continual queries. Then the CQ manager calls on the wrapper manager(s) to execute the query in the corresponding subtransaction(s).

The online demo of the OpenCQ system can be accessed using any JavaScript-enabled Web browser through the following URL `http://www.cse.ogi.edu/DISC/CQ/demo`. Figure 2 shows the screen shot of an example continual query which requires alternative transportation routes whenever the weather condition between Fort Stewart, GA and the port of Savannah, GA becomes severe (e.g., temperature below 32F, Wind speed is higher than 15 knots, visibility is less than 0.5 miles, or pressure changes more than 20 hPa in 20 minutes). This continual query involves two types of data sources, one is the transportation data sources (e.g., database sources in this example) and another is the weather watch sources (e.g., noaa.org Web site in this example). Figure 3 shows the execution trace of the installation of this continual query. Due to the space restriction, readers who are interested in more detailed description of the first prototype implementation of OpenCQ may refer to [25, 35].

6 Performance Evaluation Issues

We have presented a design and selection of alternative architectures and algorithms for a distributed event-driven information delivery system that supports continual queries. Research on event-driven continual query systems must be accompanied by a careful performance evaluation effort. For the OpenCQ 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 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 event-driven 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.

We are also interested in planning careful controlled experiments for comparing the performance of alternative condition evaluation tactics, as well as effort on studying architectural alternatives for the distributed event-driven continual query systems and their impact on performance, and possibly building a performance testbed for studying the extent to which the final design of the OpenCQ continual query system is able to meet or exceed

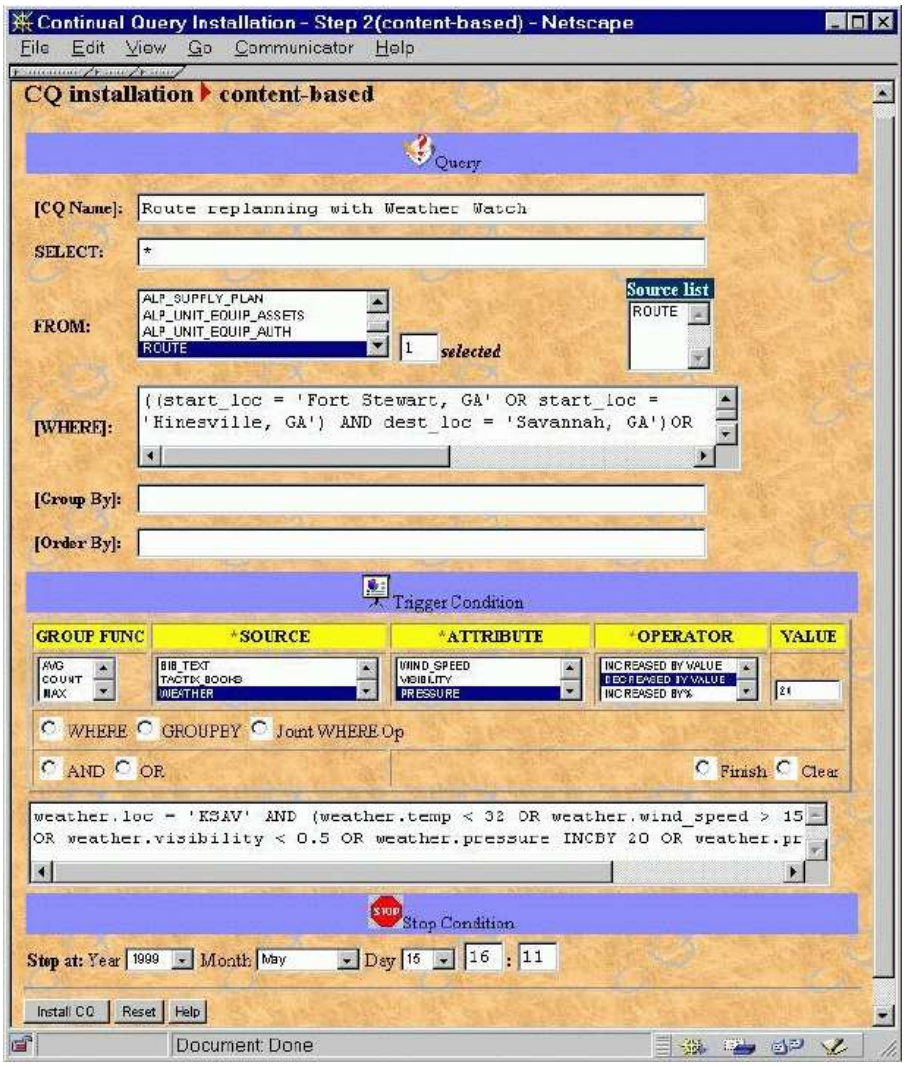

Figure 2: Continual query installation: an example

the processing requirements of a distributed time-constrained update monitoring and event-driven information delivery system.

7 Related Work

The concept of continual queries was motivated and evolved by the increasing demand on event-driven information delivery. It was also inspired by the work on continuous queries by Terry et al [36] at the early stage of the development. Comparing with Terry et al [36]'s proposal, there are a number of functionality differences. First, their proposal made several assumptions that seriously restricted the applicability of their technique to the Internet. Perhaps the most significant assumption is the limitation of database updates to append-only, disallowing deletions and modifications. Since this assumption is used in their query transformation algorithm, it has been

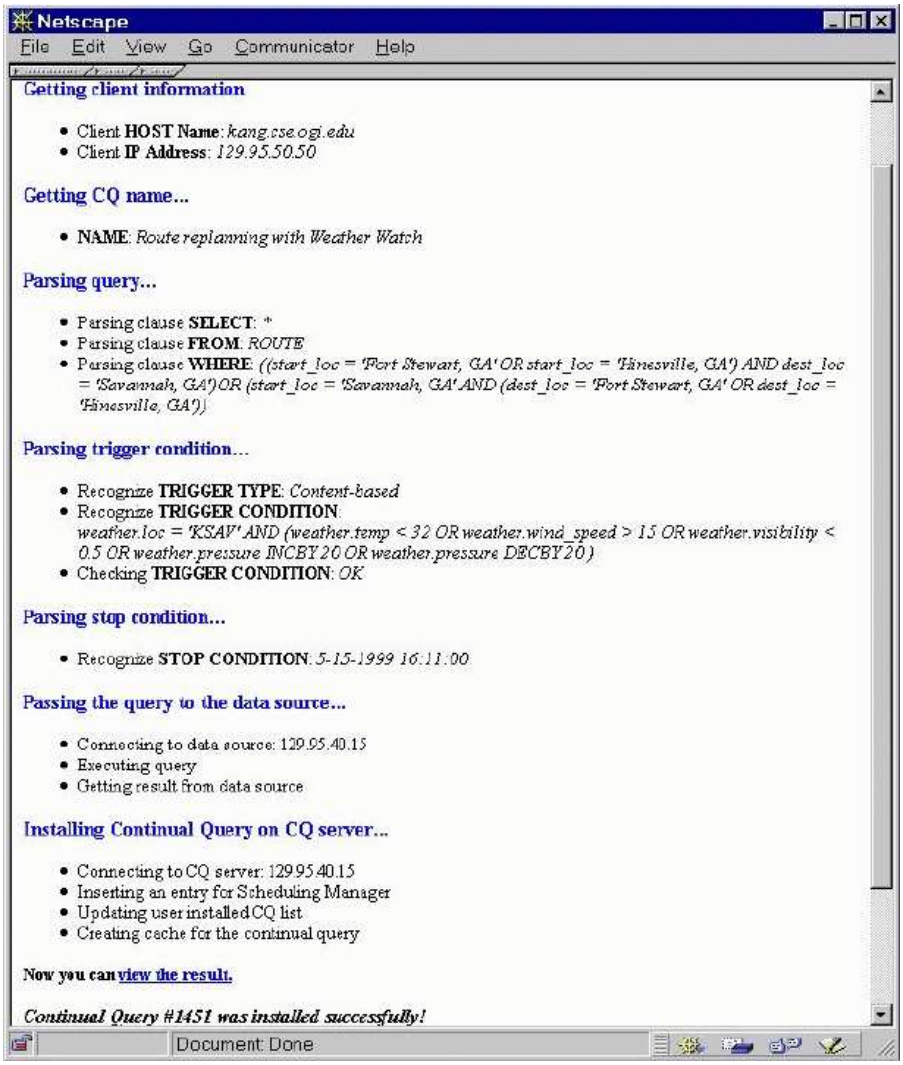

Figure 3: The execution trace of a continual query installation

difficult to relax it [2], when following their definition of continuous queries. Second, the specification model for update monitoring is purely time-based. Also there is no clean separation of query and trigger condition in the specification.

Furthermore, there has been considerable research done in the monitoring of information changes in databases and Web-based search systems. Powerful database techniques such as active databases, materialized Views, and database triggers have been developed. The design of OpenCQ continual query system is also inspired by the research in these fields. We below briefly review each of these fields. The following discussion should not be seen as a critique of these techniques. Rather, these techniques have been proposed primarily for "data-centric" environments, where data is well organized and controlled. When applied to an open information universe such as the Internet, these assumptions no longer hold (see [18] for a summary of desired system properties in the Internet), and some of the techniques do not easily extend to scale up to the open distributed interoperable

environment.

Active Database: Most of active database systems [38] provide facilities [5, 26, 32] that allow users to specify, in the form of rules, actions to be performed following changes of database state. Despite their conceptual generality, rules have been so far supported in a fairly restrictive form in practical systems, for example, by built-in triggers in relational database management systems such as Oracle, Sybase, and Informix (see a further discussion On Commercial database trigger below). Active queries, introduced in Alert [32], is yet another form of ECA rules. Active queries 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 heavily rely on the use of active tables as system built-in capability and a number of concrete extensions to a particular system – IBM Starburst DBMS [12].

Materialized View: Materialized views store a snapshot of selected database state. When a database is updated, the materialized view must be refreshed to reflect the updates. A naive solution is to rematerialize the view from the base data. In contrast, incremental update algorithms are believed to carry lower execution cost if changes to the database are moderate [14, 17]. Three approaches have been described previously. The first approach refreshes the view immediately after each update to the base table [3]. The second defers the refresh until a query is issued against the view [31]. The third refreshes the view periodically [17]. 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 [3, 14, 11] 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 [33]. Other works on distributed environments include quasi-copies for replication [1] and update anomalies in data warehouses [41].

Commercial database triggers Conceptually a database (built-in) trigger is an event–condition–action (ECA) rule in a restrictive form. Commercial DBMSs have been introducing support for database triggers at various levels, mainly due to the customers' need for better support for integrity constraints. In the SQL standard, checking of constraints, such as `price > 0` or referential integrity constraints, is triggered by the DBMS. Users can specify whether constraints are to be checked at the end of each SQL statement. However, support for triggers in SQL standard is limited. The trigger events can only be built-in SQL operations (update, insert, delete) on a single base table. The triggers can be specified only on a single base table. Triggers over views are not allowed. Database triggers can only be part of the triggering transactions and triggers can not be nested. For instance, Unlike continual queries, Sybase allows only one trigger to be associated with an operation on a table. The action part of the trigger is limited to a sequence of SQL statements. Further, triggering is restricted to one level where triggered actions themselves do not cause triggers to be fired.

Web-based Tools and Systems In addition to the Continual Queries project, 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 the URL changes and notifying the users when the URLs of the data sources of interest have changed. A representative system is the URL-Minder (`http://www.netmind.com/`). Another interesting type of change notification tools includes the news filtering services that harness news items and news groups (such as SIFT [39] and Pointcast), and the domain-specific change monitoring services, each targets at a specific domain of interest. Examples of the domain-specific change monitoring service include FedEx Opportunity which watches the US funding news on multiple Web sites and emails the users the list of new funding announcements whenever the funding news of specific interests arrive, Amazon.com which provides notification service for new books of interest, and E*Trade.com which sends out alert messages whenever the stock prices of

interest reaches specified thresholds. The third type of projects is the content change detection over HTML pages, such as the $C3$ [7] project at Stanford. An interesting feature of $C3$ is to allow users to subscribe to the change notification system through a query subscription service and then to be notified of changes whenever changes of certain kind occurs, or by weekly or daily report, or by explicit request. A key feature that distinguishes the continual query system OpenCQ from these Web-based change monitoring systems is its fine-grained support for content-sensitive, event-driven information delivery and its unified framework that combines pull and push through the support of both polling and filtering based on the user or application requests. For example, the OpenCQ system, to our knowledge, is the only system that is able to install complex content-based continual queries such as "*Notify me whenever the stock prices of IBM and of AT&T both drop more than 5%*".

8 Conclusion

We have introduced *continual queries* (CQ), described a detailed design of OpenCQ – a distributed event-driven information delivery system that supports continual queries, and outlined a prototype implementation of OpenCQ. In summary, CQ strikes a conceptual balance between event-condition-action (ECA) models on one end of spectrum and CORBA event services on the other. ECA models are very powerful, but difficult to implement in large-scale heterogeneous systems such as the Internet. In contrast, CORBA services are simple to implement but miss application semantics. CQs focus on update monitoring (instead of generic actions as in ECA) but provides a rich event model including both time-based and content-based triggering conditions.

OpenCQ is a distributed event-driven information delivery system that implements CQs in a three-tier architecture (client, server, and wrapper). The key components of OpenCQ server are: a CQ manager, a trigger condition evaluator, and a variety of event detectors. The CQ manager is responsible for coordination tasks and communications between the client/server tiers and server/wrapper tiers. The condition evaluator and event detectors are responsible for monitoring updates according to specified update thresholds of interest and the time constraints. The client tier manages the repository of CQ information and GUIs. The wrapper tier handles the heterogeneous data sources for the server. This three-tier architecture decomposes the Internet data flow into modular components, enabling the OpenCQ system to be adaptable and scalable in the presence of constant changes in the information space. We have implemented the first prototype of OpenCQ to demonstrate the feasibility of the architecture. It builds on top of the distributed interoperable information mediation system DIOM [21, 19].

Our work on the continual query system for update monitoring continues. Several issues need to be further explored, including experimentation to evaluate the performance of alternative architectures and algorithms, recovery of CQs in the presence of failures at either CQ server or CQ client, performance improvement by incorporating research results and techniques for incremental query evaluation and multiple query optimization into the CQ event detection and condition evaluation process.

Acknowledgement

The authors would like to acknowledge John Biggs and Wei Han for implementing the CQ wrapper for html data sources and the CQ Weather Watch server, and Dave Buttler for implementing the CQ wrappers to bibliography files and the CQ Bibliography server, on top of the CQ event detectors and condition evaluation manager. This research is partially supported by DARPA contract MDA972-97-1-0016, and grants from Intel and Boeing.

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] D. Barbara and R. Alonso. Processing continuous queries in general environments. Technical report, Matsushita Information Technology Laboratory, Princeton, NJ, June 1993.

[3] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 61–71, Washington, DC, May 1986.

[4] A. P. Buchmann, J. Zimmermann, J. Blakeley, and D. Wells. Building an intergrated active oodbms: Requirements, architecture, and design decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–125, February 1995.

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

[6] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite events for active databases: Semantics, contexs and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

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

[8] S. Chawathe, A. Rajaraman, Hector Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of ACM SIGMOD Conference*, 1996.

[9] T. I. N. A. Consortium. *TINA Notification Service Description.* July, 1996.

[10] C. Gerety. Hp softbench: A new generation of software development tools. *Hewlett-Packard Journal*, 41(3):48–59, 1990.

[11] 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.

[12] 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.

[13] M. Haertel, D. Hayes, R. Tamllman, L. Tower, P. Eggert, and W. Davison. The gnu diff program. Texinfo system documentation, Available by anonymous ftp at `prep.ai.mit.edu`.

[14] E. N. Hanson. A performance analysis of view materialization strategies. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 440–453, San Francisco, CA, May 1987.

[15] M. Kiffer. Ediff-a comprehensive interface to diff for emacs 19. Available by anonymous ftp at `cs.sunnysb.edu`.

[16] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, 1995.

[17] 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.

[18] 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.

[19] L. Liu and C. Pu. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *DISTRIBUTED AND PARALLEL DATABASES: An International Journal*, 5(2), 1997.

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

[21] 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.

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

[23] L. Liu and C. Pu. Complex event specification and event detection for continual queries. Technical report, OGI/CSE, Portland, OR, March 1998.

[24] 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.

[25] 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.

[26] 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.

[27] OMG. *The Common Object Request Broker: Architecture and specification.* Object Management Group, Object Request Broker Task Force, Revision 2.0, 1995.

[28] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Colorado, December 1995.

[29] S. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, 1990.

[30] A. Rosenthal and U. Chakarvarthy. Anatomy of a modular multiple query optimizer. In *The International Conference on Very Large Data Bases*, 1988.

[31] 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.

[32] 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.

[33] 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.

[34] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 10(3), 1986.

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

[36] 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.

[37] E. N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *Proceedings of the OOPSLA Conference*, 1996.

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

[39] 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.

[40] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6), 1989.

[41] 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.

Ling Liu is an assistant professor at Oregon Graduate Institute, where she is the Co-PI and technical lead of the Continual Queries project described in this paper. Her current research interests are in Internet-scale distributed information systems, multidatabase systems, multimedia, object-oriented systems, data mining and data warehousing, and software evolution. In addition to the CQ project, she is also leading the DIOM project (Distributed Interoperable Object Model and its applications to Large Scale Interoperable Database Systems) and the TAM project (Transactional support for complex workflow activities). She received her Ph.D. in 1993 from Tilburg University. Before joining OGI in 1997, She was an assistant professor at the Department of Computing Science, University of Alberta. She has published more than 50 papers in international journals and international conferences, served on more than 15 program committees, and is currently an associate editor of ACM SIGMOD Record.

Calton Pu was born in Taiwan, but grew up in Brazil. He received his PhD from University of Washington in 1986 and is currently a Professor at the Oregon Graduate Institute where he has two research focuses and a number of collaborations. First, he has been working on next-generation operating system kernels to achieve high performance, adaptiveness, security, and modularity, using program specialization, software feedback, and domain-specific languages. This area has included projects such as Synthetix, Immunix, Microlanguages, and Microfeedback, applied to distributed multimedia and system survivability. Second, he has been working on new data and transaction management by extending database technology. This area has included projects such as Epsilon Serializability, Reflective Transaction Framework, and Continual Queries over the Internet. His collaborations include applications of these techniques in scientific research on macromolecular structure data, weather data, and environmental data, as well as in industrial settings. He has published more than 20 journal papers, 70 conference and refereed workshop papers, and served on more than 30 program committees, including the co-PC chair of SRDS'95, co-general chair of ICDE'97, and co-PC chair of ICDE'99. He is currently an associate editor of IEEE TKDE. His research is funded by NSF, DARPA, Intel, Tektronix, and other sources.

Wei Tang received BS degree in computer science from Peking University, China, in 1996 and MS degree in computing science from University of Alberta, Canada, in 1998. He is currently a Ph.D. candidate at Oregon Graduate Institute of Science and Technology (OGI). He was the main implementer of the work described in this paper. His research interests include data warehousing techniques, data mining, and distributed information systems on the Internet.