Information Monitoring on the Web: A Scalable Solution [*]

Ling Liu, Wei Tang, David Buttler, Calton Pu

Center for Experimental Research of Computing Systems
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280, USA
{lingliu,wtang,buttler,calton}@cc.gatech.edu

Abstract

This paper presents WebCQ, a continual query system for large-scale Web information monitoring. WebCQ is designed to discover and detect changes to Web pages efficiently, and to notify users of interesting changes with personalized messages. Users' Web page monitoring requests are modeled as continual queries on the Web and referred to as Web page sentinels. The system consists of five main components: a change detection robot that discovers and detects changes, a proxy cache service that reduces the communication traffics to the original information provider on the remote server, a trigger evaluation tool that can filter only the changes that match certain thresholds, a personalized change presentation tool that highlights Web page changes, and a change notification service that displays and delivers interesting changes and fresh information to the right users at the right time. This paper describes the WebCQ system with an emphasis on the general issues in designing and engineering a large-scale information change monitoring system on the Web. There are two main contributions. First, we present the mechanisms that WebCQ provides to support various types of Web page sentinels for finding and displaying interesting changes to Web pages. The large collection of sentinel types allows WebCQ to efficiently locate and monitor a wide range of changes in Web pages. The second contribution is the development of sentinel grouping techniques for efficient and scalable processing of large number of concurrently running triggers and Web page sentinels. We report our initial experimental results showing the effectiveness of the proposed solutions.

Keywords: World Wide Web, Information Monitoring, Fresh Information Delivery, Continual Queries, Change Notification, Web Service.

[*]An extended abstract of this paper was included in the proceedings of the International Conference on Information and Knowledge Management (CIKM), Nov. 7-10, 2000, Washington D.C., ACM Press. pp512-519.

1 Introduction

The World Wide Web (the Web), as one of the most popular applications currently running on the Internet, continues to grow at an astounding speed. Not only does the number of static web pages increase approximately 15% per month, the number of dynamic pages generated by programs has been growing exponentially, overwhelming the ability of users to keep up with the up-to-date information on the Web. Search engines have become necessary tools for finding and accessing information on the Web.

Motivation and Problem Statement
As the Web grows and evolves, we observe some rapid changes in the ways in which fresh information is delivered and disseminated. The mode of data transfer is shifting from a “pull-only” model to a “push-pull” model [4]. Many believe that the “push” style of information delivery and dissemination is, to some extent, a natural solution to the scale of the Internet. In the ”push-pull” model, some data is *pushed* to users without an explicit pull request. The “push” style enables services to be served asynchronously as they become available. Instead of having users tracking when to visit web pages of interest and identifying what and how the page of interest has been changed manually, the information change monitoring service is becoming increasingly popular, which enables information to be delivered automatically while it is still fresh. The push service is specially suited for busy individuals and for delivering transient data such as stock quotes, product prices, news headlines, and weather information.

However, designers of large-scale Web-based change monitoring and notification systems face a common problem: HTTP [13] is a pure request-response model and it does not allow servers to asynchronously notify clients of events on the server-side. As a result, search engines to date, although powerful in helping users locating and finding information of interest, do not support tracking changes on behalf of users and cannot deliver timely information to the right users at the right time.

From systems perspective, many distributed systems need the functionality of asynchronous event dissemination. Examples include callbacks in distributed file systems [36] and gossip messages in lazy replication systems [20]. Although several existing systems, such as ICQ [18] and Pointcast [33], support large-scale notification using centralized proxies that relay events from servers to clients, these notification mechanisms are specialized for their applications. Hence, it is desirable to design a general-purpose event dissemination infrastructure on which large-scale change monitoring systems can be implemented.

Now let us look at the problems from users’ perspective, namely what is the common behavior of users who wish to monitor changes in web pages. Individuals often use a search engine to find a page of interest, and then bookmark the pages that they wish to re-visit. Upon a revisit, a fresh copy of the page will be obtained, and the user needs to determine if it has changed in an interesting way manually. Obviously the first challenge is the ability to allow users to only revisit a page *when* the page has changed in a way that is interesting. Furthermore, if a user becomes interested in tracking changes over a large number of web pages, he or she may wish to be notified not only when to re-visit the pages of interest but also what the concrete changes are. This is because when the number of pages of interest grows large it will be difficult, if not impossible, for a user to remember the concrete details about every page in which he or she was interested. Therefore, the second important challenge is to identify *what* and *how* the page has changed, including the types of changes and the amount of changes between the fresh copy and the copy last seen.

The State of Art and Technical Challenges

Several tools are available to assist users in tracking of *when* web pages of interest have changed [12, 6]. Most of these tools are in .com domain and offer tracking service either from a centralized server or a client's machine. Server-based tools track pages that are previously registered or submitted by users and notify of users changes via email or over the Web upon a request. Examples include Netmind [2], TracerLock [3], AIDE [12], and Webclipping.com [40]. Client-based tools run on a user's machine, and track changes either periodically or on demand, such as WebWhacker^{TM}[1]. We can also classify these tools based on the coverage of the service. Some tools offer tracking service on a specific or a constrained set of URLs instead of on any registered URLs. For example, several client based tools, such as smart bookmarks [38] or bookmark surfbot [32], use the user's list of bookmarks; other tools either restrict the number of URLs to be monitored for a user, such as WWWFetch [44] or track the new web sites of chosen topics such as Webcatcher, or monitor changes of selected topics (e.g., current stock prices, weather forecasts, sports scores, headlines) such as WebSprite [42].

Up till now most tracking tool development has gone on at companies with little exposure of technical details, especially the efficiency, the scalability, and the tracking quality of such systems. Furthermore, from individual users' perspective, we observe three common problems with these tools. First, with the exception of Netmind [2] and AIDE [12], most of the tools only address the problem of when to re-visit a fresh copy of the pages of interest but not the problem of what and how the pages have changed. Second, all these tools handle the *when* problem with a limited set of capabilities. For instance, Netmind can only track changes on a selected text region, a registered link or image, a keyword, or the timestamp of the page. The third problem with all these tools is the scalability of their notification service with respect to individual users. Typically, these tools treat each Web page tracking request as a unit of notification. Users who register a large number of pages with the tracking service are easily overwhelmed with the large number of frequent email notification messages.

Scope and Contributions

In this paper we present an in-depth description of WebCQ, a prototype system for large-scale Web information monitoring. WebCQ capitalizes on the structure present in hypertext and the concept of continual queries [21, 24]. The WebCQ system is designed to discover and detect changes to the Web pages efficiently, and to provide a personalized notification of what and how Web pages of interest have changed. Users' Web page monitoring requests are modeled as continual queries on the Web and referred to as Web page sentinels. The system consists of five main components: a change detection robot that discovers and detects changes, a proxy cache service that reduces the communication traffics to the original information provider on the remote server, a trigger evaluation tool that can filter and highlight only the changes that match certain thresholds, a personalized change presentation tool that highlights Web page changes, and a change notification service that displays and delivers interesting changes and fresh information to the right users at the right time. This paper describes the WebCQ system with an emphasis on the general issues in designing and engineering a large-scale information change monitoring system on the Web. One of the key challenges for building a Web-based change monitoring system is that it must scale to large number of sources and users, as well as the number of notifications per user. The paper makes two main contributions. First, we present the mechanisms that WebCQ provides to support various types of Web page sentinels for finding and displaying interesting changes to Web pages. The large collection of sentinel types allows WebCQ to efficiently locate and monitor a wide range of changes in Web pages. The second contribution is the development of sentinel grouping techniques for efficient and scalable processing of large number of concurrently running triggers and Web page sentinels. We report our initial experimental results showing the effectiveness of the proposed solutions.

With WebCQ, our goal is to push more research, development, and understanding in the academic world, to improve the quality of web tracking services, and to investigate various systems issues in building large-scale change tracking engines on the Web.

Organization of the Paper
The rest of the paper proceeds as follows. We first describe the system architecture, the various WebCQ sentinel types, and the installation process of WebCQ sentinels. Then we discuss the main components of WebCQ, including strategies for detecting changes to arbitrary web pages, the methods for representing changes in web pages, and the WebCQ notification service. The mechanism and indexing structure for sentinel grouping is presented as an important optimization technique for efficient and scalable processing of large number of concurrent sentinels. We also comment on several research and engineering issues ranging from monitoring latency and notification latency, scalability, availability to applications and integration with other tools. The paper ends with an overview of the related work, a summary, and an outline of directions for future work.

2 WebCQ System Architecture

WebCQ is a server-based change detection and notification system for monitoring changes in arbitrary web pages. The system consists of five main components as follows (shown in Figure 1):

- a change detection robot that discovers and detects changes to arbitrary Web pages
- a proxy cache service that reduces the communication traffics to the original information provider on the remote server
- a trigger evaluation tool that filters only the changes that match certain thresholds
- a personalized change presentation tool that highlights changes between the web page last seen and the new version of the page
- a centralized change notification service that not only notifies users of changes to the Web pages

of interest but also provide a personalized view of how Web pages have changed and what fresh information should be delivered.

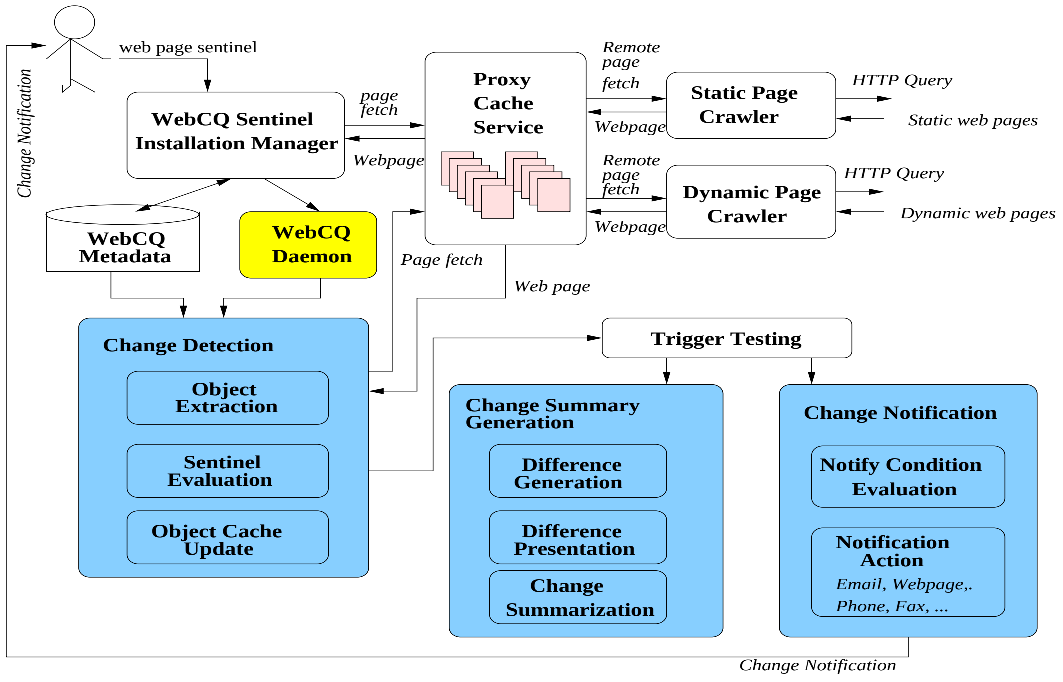


Figure 1: WebCQ System Architecture

In the first release of WebCQ, users register their web page monitoring requests with WebCQ using HTML forms. Typically a user enters a URL of interest (say the Internet movie database). The WebCQ installation manager will pass this URL to the proxy server. The proxy server will display the page in the lower frame of the WebCQ window as shown in Figure 2.

The user can click on any link of interest on the page until she reaches the exact page that she wishes to monitor (see Figure 2). Now the user may select the type of information content she wants WebCQ to monitor for her from the pull-down menu shown in Figure 2. We call each registered request a web page sentinel. Sentinels in WebCQ are modeled as continual queries [24] and are persistent objects. The sentinel installation is submitted to the sentinel installation manager once the user has entered her email address and her preferred monitoring frequency, notification frequency, and notification method. The sentinel installation manager first translates the sentinel request into a continual query expression [24], and then registers the continual query in the WebCQ metadata repository. The WebCQ daemon fires the change detection robot periodically to evaluate all installed sentinels. Once interesting changes are detected, the difference generation and summarization module will be fired to generate a detailed change notification report for each sentinel and a change summarization for each client. A notification will be fired to notify users of the interesting changes.

Users may monitor a web page for any change or specific changes in images, links, words, a chosen phrase, a chosen table, or a chosen list in the page. Furthermore, WebCQ allows users to monitor any

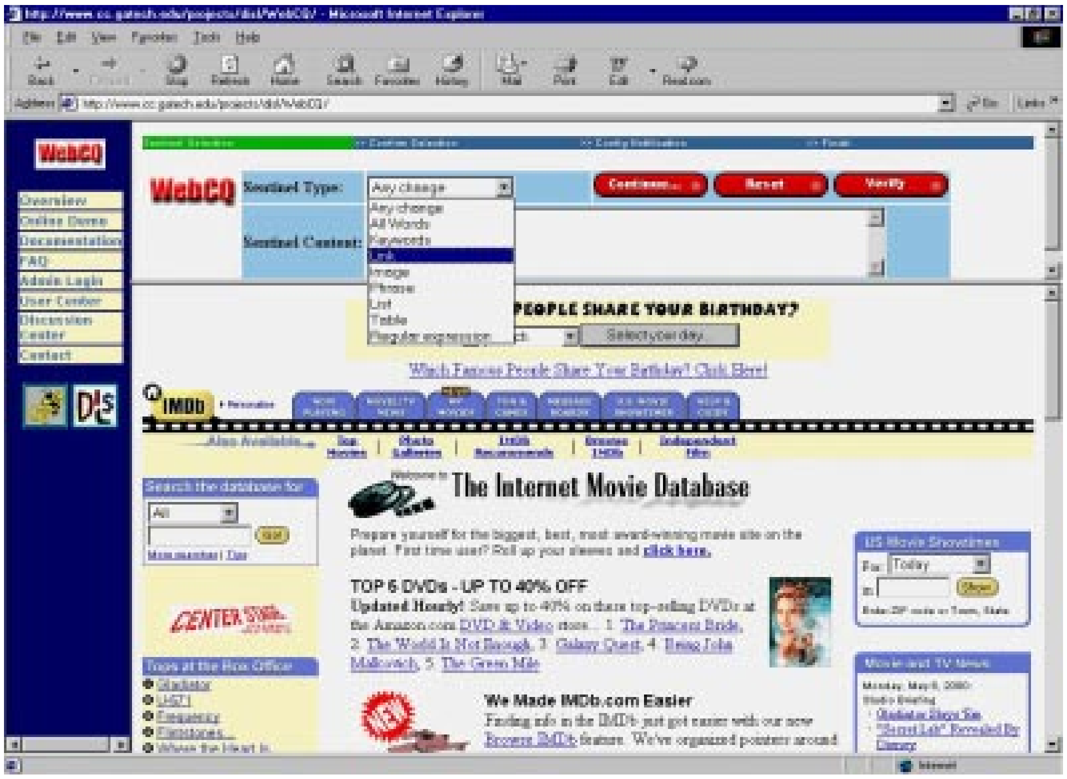


Figure 2: Installing a WebCQ sentinel using the proxy service

fragment in a Web page, which can be specified by a regular expression. Table 1 shows a list of basic sentinel types supported in WebCQ.

Users may register their update monitoring requests using basic sentinels or composite sentinels. A composite sentinel type is a composition of two or more sentinel types. A sentinel is either an instance of a basic sentinel type or an instance of a composite sentinel type. The operators used for sentinel composition are omitted here due to the space restriction. The syntax of a WebCQ sentinel is partially described as follows:

```
<WebCQ Sentinel> ::=
   CREATE Sentinel <sentinel-name> AS
       Sentinel Type <sentinel-type>
       Sentinel Target <sentinel target>
       Sentinel Object <object desc>
       [Trigger Condition <time interval>]
       [Notification Condition <time interval>]
       [Notification Method <method-desc>]
       [Start Condition <time point>]
       Stop Condition <time point>
<sentinel-name>::=<text string>
<sentinel-type>::= <any change> | <all links>
    | <all images> | <all words> | <table> | <list>
    | < phrase> | <regular expression> | keyword
<sentinel target>::=<url>
```

sentinel types	Synopsis	Sentinel Monitoring Method
Any Change	Any update on the page object	watch any change to the modification timestamp of the file, compare MD5 hash values
Link Change	Any change to links of the page	when new links added or old links removed
Image Change	Any change to images of the page	when new images added or old images removed
Words Change	Any change to words of the page	when new words added or existing words removed
Phrase Update	Any change to the selected text phrase	identify and extract the selected phrase; detect any change in the phrase
Table Change	Any change to the content of the table	identify and extract the selected table; detect any change to the table
List Change	Any change to the content of the list	identify and extract the selected list; detect any change to the list
Arbitrary text Change	Any change to the text fragment specified by a regular expression	identify and detect any change ; in the selected fragment
Keywords	The specified keywords appear in or disappear from the page	watch if the selected keywords disappear or appear

Table 1: A list of basic sentinel types in WebCQ

```
<object desc>::= <text string>
<time interval>::=<integer> {<minute> | <hour> | <day> | <week>}
<method-desc>::= <email-address> | <fax>
    | <personalized web-bulletin> | <phone>
    | <email-address> <personalized web-bulletin>
<time point>::=<month>‘‘-"<day>‘‘-"<year>‘‘-" <hour>‘‘-"<min>‘‘-"<TimeZone>
```

The trigger condition of a sentinel specifies how frequent the change detection robot should be fired to check whether any interesting changes have happened. The notification condition specifies the desired frequency for receiving change notification from the WebCQ system, and is designed to support situations where the desired notification condition is different from the trigger condition. It can be the same as the trigger condition or n times of the trigger frequency. For instance, one may want WebCQ to track changes daily but receive the notification report weekly. The optional clause `Notification Method` allows users to select a notification means from the set of methods supported by the system. In the first prototype of WebCQ, we only support two notification methods: email and personalized web bulletin. Due to the space limitation, we omit the complete syntax of the CQ specification language in this paper. Readers who are interested in more details may refer to [26]. A WebCQ sentinel example is given by Example 1.

Example 1 *Consider a WebCQ sentinel "track any change on the 'IMDb Movie of the Day' from the front page of the Internet Movie Database Web site". The installation the through WebCQ GUI is shown partially in Figure 4. The user may select regular expression as the sentinel type in the monitoring condition, namely "*`IMDb Movie of the Day` *.*?more.*?\)". WebCQ will automatically highlight the paragraph on IMDb Movie of the Day and capture and store this sentinel in the following WebCQ sentinel expression (in the form of a continual query [21, 24]):*

```
CREATE Sentinel movie_of_the_day AS
```

```
Sentinel Name ``IMDB movie of the day"
Sentinel Type  regular expression
Sentinel Target  http://www.imdb.com
Sentinel Object ``IMDB Movie of the Day.*?more.*?\)''
Trigger Condition  1 day
Notification Condition  1 day
Notification Method  email (john.doe@somedomain.com)
Start Condition July 1, 2002
Stop Condition June 30, 2003
```

The default duration for a continual query (CQ) is one year starting from the time of the installation. Our experience with the continual query system [24, 25] shows that the capability of supporting both trigger condition and notification condition is especially useful for the situations where a user prefers to receive a summarization of all the change detection reports of a sentinel periodically rather than receiving an email whenever the sentinel is evaluated and changes are found. Such sentinel-specific summarization is also useful when notification of changes to web pages is to be sent to an application program to trigger an action, rather than or in addition to sending it by email to a user. In WebCQ, the trigger condition and the notification condition are set to be the same by default. The notification model of the WebCQ system will be discussed further in Section 5. In the second release of the WebCQ system, we also allow triggers to be content-sensitive. For instance, users can specify a monitoring requesting for tracking changes on the Movie of the Day news from the Internet Movie database front page, with a trigger condition that requires to deliver the change only when more than 50% of the content in the Movie of the Day news has changed. Another example is *"notify me when Panic Room starring Jody Foster becomes the movie of the day"*.

Other important features of the WebCQ architecture include its server-based proxy cache service. The goal of introducing a proxy cache service is to reduce the cost of repeated connections to the remote information servers. The proxy service works as follows: Any remote page request will be sent to the nearest proxy server first. The proxy server only forwards the remote fetch request (such as HTTP Get or HTTP Post) to the corresponding information server if a copy of the requested page is not found in the local proxy cache or any of the relevant proxies. The static page crawler is used to fetch static pages, while the dynamic page crawler is designed for handling the remote fetch of dynamic pages, including search interface extraction and HTTP query composition. An obvious advantage of the proxy service is to enable the WebCQ system to share the cost of remote page fetch among all monitoring requests over the same web page. By reducing the frequency and the overhead of network connections to remote data servers, the unnecessary network connection cost is avoided, and the scalability of the system is enhanced.

In the subsequent sections we focus our discussions on the following three main components of the WebCQ system: change detection robot, difference generation and summarization, and change notification service.

3 Change Detection Robot

The change detection robot consists of three modules: object extraction, sentinel evaluation, and object cache update. For each sentinel, the change detection robot first fetches the web page being monitored by the sentinel through a page fetch call to the proxy server (recall Figure 1). Then it performs the following three tasks: *object extraction, sentinel evaluation, object cache update*, which will be discussed in the following sections.

3.1 Object Extraction

In order to detect changes on a Web page, we have to first identify the objects residing on the page. The *Object Extraction* task locates and extracts the objects being monitored from the page. In WebCQ, we provide three different methods to extract web page contents of interest. They are HTML-based data extraction, rule-based (using regular-expression) data extraction, and general syntax-based data extraction, as shown in Figure 3.

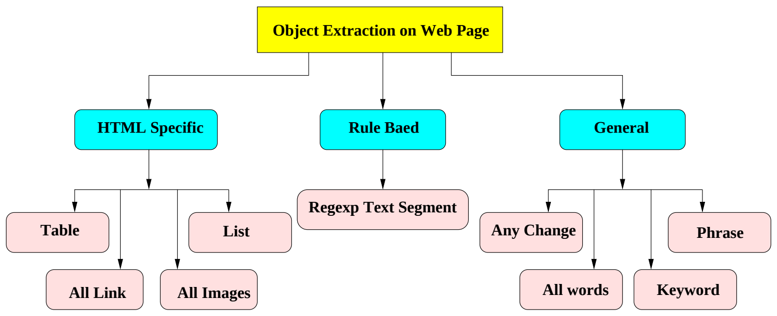


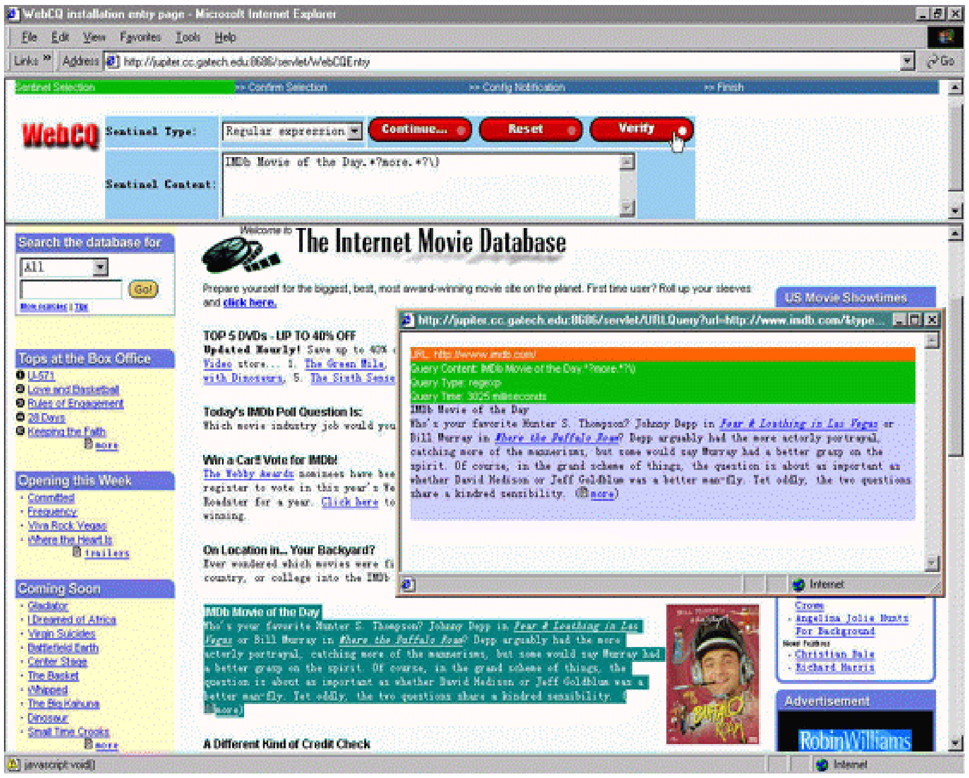
Figure 3: Object Extraction in WebCQ

- For HTML-specific data extractions, four different logical structures are extracted, including *Table, List, AllLink and AllImage.* A small token-based HTML parser is used to identify HTML tags and extract logical structures of interest from a given Web page.

- For rule-based data extraction, WebCQ identifies and extracts any text string specified by a Perl-like regular expression[1]. For instance, the regular expression "`<td><b>(Java.*?)</b></td>`" extracts a table cell containing a string starting with 'Java' in bold font. Recall Example 1 in Section 2, it is a regular-expression sentinel, which tracks changes to the "IMDB Movie of the Day" information on the front page of Internet Movie Database web site. The target object being monitored is shown in Figure 4 as the shaded area in the lower frame as well as in the pop-up window. The *Regular Expression* sentinel type is an important feature of WebCQ. An arbitrary text object in a Web page can be constructed using a *regular expression.* Almost all other data object types can be transformed into a regular expression type. It is well known that different

[1]See http://www.perldoc.com/perl5.6/pod/perlre.html for details about Perl regular expressions

regular expressions have different evaluation cost. In the current prototype of WebCQ system [41], we use the regular expression package from GNU (gnu.regexp). One of our future research efforts is to support the transformation of user-defined regular expressions into equivalent but more efficient ones.


WebCQ installation entry page - Microsoft Internet Explorer

Address: http://jupiter.cc.gatech.edu:8686/servlet/WebCQEntry

Sentinel Selection >> Confirm Selection >> Config Notification >> Finish

Sentinel Type: Regular expression

Sentinel Content: IMDb Movie of the Day.*?more.*?\)

http://jupiter.cc.gatech.edu:8686/servlet/URLQuery?url=http://www.imdb.com/&type...

URL: http://www.imdb.com/
Query Content: IMDb Movie of the Day.*?more.*?\)
Query Type: regexp
Query Time: 3025 milliseconds

IMDb Movie of the Day
Who's your favorite Hunter S. Thompson? Johnny Depp in Fear & Loathing in Las Vegas or Bill Murray in Where the Buffalo Roam? Depp arguably had the more actorly portrayal, catching more of the mannerisms, but some would say Murray had a better grasp on the spirit. Of course, in the grand scheme of things, the question is about as important as whether David Hedison or Jeff Goldblum was a better man-fly. Yet oddly, the two questions share a kindred sensibility. (more)

Figure 4: Object extraction during a sentinel installation

- For general syntax-based data extraction, in addition to sentinels of any change type, three types of logical objects are extracted: *Keyword, Phrase,* and *All words.* Standard Java String and Tokenizer packages are used in the WebCQ implementation. For any change sentinels, WebCQ first uses the last modification time stamp to determine if the page has changed at all. If yes, WebCQ will run a comparison of the MD5 [35] signature of the new and old Web page being monitored before computing the actual differences.

To make the paper self-contained, a brief description of the basic concepts used in this section is given in Appendix A.

3.2 Sentinel Evaluation

The Sentinel Evaluation module compares the object extracted from the current copy of the page with the previous cached copy of the object. The difference between the two copies of the object is computed and changes are detected. The sentinel change detection algorithm is described in pseudo code in Appendix B. Here we give an informal overview of the sentinel evaluation process.

- For detecting "any change" to a page, we use a set of progressive techniques as described below:

  First, we use the HTTP `HEAD` request to obtain the last-modified timestamp for all static pages. Occasionally, the last modification timestamp alone may not be enough. For example, a page whose last modification timestamp changes but no changes actually occur in its body may be flagged as changed. One may use a `HEAD` request combined with a checksum evaluation to handle such situations.

  In fact, several methods can be used in addition to the last modified timestamp to double-check if a page has changed. For example, once a `HEAD` request indicates a change, a simple solution is to use the file size in addition to the modification timestamp. This method will avoid those cases where the content of a page did not change though the timestamp is changed. However, it will miss the cases where a page has changed but the amount of text deleted equals the amount of changes inserted, leaving the file size unchanged. From HTTP 1.1 on, a "Content-MD5" header field is provided[17] as an MD5 digest (128 bits) of the document body. If this field is present, it can be used to compare with the previous stored MD5 digest of the page to detect if change happens to the document. However, at this point, we do not know what has been changed.

  A better approach is to use `HTTP GET` to retrieve the page and compute the difference. Obviously, generating the difference will give a more accurate detection result but it is also more expensive. In the situation where no last modification timestamp is provided such as the case of dynamic pages or a prior attempt that no timestamp is given, then HTTP `GET` is used to retrieve the body of the page via the proxy service and a checksum (such as MD5 digest) is computed.

- To detect changes to hypertext links or images in a page, the hypertext reference tag (e.g., `<A href=...>`) is used to identify all hypertext links in the page; and the image tag (`<IMG src=...>`) is used to identify all images in the page. For *All links, All images* and *All words*, they are all set-based sentinel types. We detect object insertions and deletions using set difference operations.

- For sentinels of types *Phrase, Table, List*, the task of identifying and extracting the correct object begin monitored in a page is more complicated. We introduce the concept of context bounding box. A content bounding box for an object being monitored in a web page is defined by the surrounding context of the object. This context-bounding box will be used to identify and extract the object in the subsequent copies of the page. There are several ways to define the bounding box context for an object. For example, we may define the bounding box context of an object by a given number of words before and after the boundary of the object. The context bounding box approach assumes that the selected text surrounding the object being monitored is relatively stable.

  The context-bounding box is particularly useful for handling duplicates of objects in a page, i.e., the objects that have the same contents within a page. It is understood that the chosen bounding box should be more stable than the object being monitored to ensure the precision of the object location and extraction. The quality of the monitoring to some extent depends on how we choose the bounding box for a given sentinel and whether and how often it may change. In the first prototype of WebCQ, we are using ten words before and ten words after the object to be extracted to define its bounding box. However, the selection criteria for bounding-boxes should be a configurable parameter. One can adjust it to use different selection criteria for defining the bounding box for different types of Web pages.

Figure 5 sketches the context-bounding box, and a set of examples, including a case where the bounding box is stable and representative cases in which the bounding box can be changed. We use B to denote the beginning of the bounding box for the object O and E the ending bounding box for O. An updated version of the object being monitored is in shaded area. We represent changes to the bounding boxes (either beginning or ending) as corner wedges.

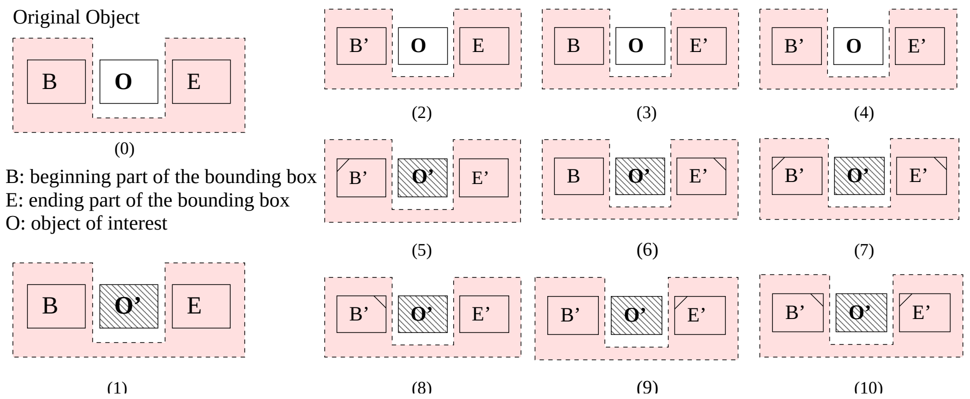


Figure 5: Context bounding box and types of changes

Case (1) in Figure 5 shows the stable situation where the object being monitored is changed and the bounding box is not. The sentinel change detection algorithm works the best in this type of situations. Cases (2), (3) and (4) are not considered as a change since there is no change to the object of interest. For changes similar to those shown in Cases (5), (6), and (7), we need to dynamically adjust the bounding box to precisely locate the object. In the first prototype of WebCQ, if less than 50% of the words used in a bounding box have changed, then we adjust the bounding box to the unchanged portion. For example, let B $= < w_1, w_2, ..., w_n >$ (n=10 by default), and $B^{'} = < w_1^{'}, w_2^{'}, ..., w_i^{'}, w_{i+1}, ..., w_n >$, i>1, be the changed version of B. That is, only the first i words in the beginning bounding box B has changed, leaving the part closer to the object being monitored O$^{'}$ intact. We can adjust the beginning bounding box to be of length i instead of 10 if $i < 5$ holds. Similarly, let E $= < w_1, w_2, ..., w_n >$, and E$^{'} = < w_1, ..., w_i, w_{i+1}^{'}, ..., w_n^{'} >$, $1 \leq i < n$, be the changed version of E. Only the last $n - i$ words in the ending bounding box E has changed, leaving the part closer to the object being monitored O$^{'}$ intact. We can adjust the ending bounding box to be of length i instead of 10 when the amount of changes is less than 50%, i.e., $i < 5$. A problem may arise when both beginning and ending boxes B($B^{'}$) or E($E^{'}$) are substrings of the object O$^{'}$, or vice versa. Therefore, in the current version of WebCQ, we do not let users monitor an object that is too small (e.g., less than 5 words for the *Phrase* sentinels).

Recall again Example 1 in Section 2, instead of using regular expression sentinel, an alternative way is to express the same tracking request using the phrase sentinel and set B=<"IMDb","movie","of", "the","day"> and E=<"more",")"> to identify the paragraph under "IMDb Movie of the day" as the phrase object to be monitored. This example will generate the same sentinel object as the one shown in Figure 4.

Finally, the algorithm will fail for the last three cases (i.e., (8), (9) and (10)) in which either beginning or ending bounding box is changed in the portion of the box that are close to the object being monitored.

Our experience has shown that for static web pages, using words instead of tags to define the bounding box context of an object often gives us more accurate results. However, for dynamic pages, using tags instead of words tends to provide results of higher accuracy. For a table or a list object, users may select a sentence or a keyword within the table or the list as the unique identifying text of the table or list. This mechanism also applies to phrase objects. Whenever users provide such identifying text, instead of using the bounding box context method, WebCQ will then use such identifying text to identify the object.

- For sentinels of type *Regular expression*, the object content is represented by a Perl-like regular expression. To detect a change in a Web page, we simply compare the new regular expression evaluation with the cached evaluation copy. The comparison is based on character string match.

3.3 Object Cache Update

We have discussed the first two steps in the change detection process of the WebCQ sentinels. Object Cache Update module is the last step. It is fired if a change is detected for a particular sentinel upon the completion of a sentinel evaluation.

More concretely, given a page sentinel, the change detection robot first fires the object extraction module that performs two tasks: (1) It fetches the current copy of the page through the proxy service. (2) Then it uses the bounding box of the sentinel object to locate and extract the phrase in the current copy of the page. Then the sentinel evaluation module is fired. It loads the cached copy of the sentinel object, which was extracted from the previous copy of the page, and compares the cached copy with the current copy of the sentinel object. If no difference is found, the change detection ends without taking any action for summarization or notification. An evaluation log is recorded. If a difference is found, the following three actions are performed:

- The change summary generation component is triggered first. The difference is computed between the two copies of **O**. A detailed change report is created and the personalized change summarization is updated.

- The object cache update module is fired to refresh the cache copy of the object **O** to the current (new) copy O'.
- The notification manager sends an email notification to the corresponding user with a brief summary of the changes and a link to the web page where a personalized change summarization report can be found.

4 Difference Generation and Summarization

Most of the tools that monitor changes to web pages have the capability of notifying users that something on a page has changed with the link to the new copy of the page. But few are able to include what and how the page has changed in the notification report. When the number of pages that a user is interested in tracking changes is large and the changes on the pages are subtle, it is likely that the user will not know what has changed by simply viewing the new copy of the page and comparing it with what the user has remembered when the page was last seen. Therefore, computing and showing the difference to a web page is a critical component of an information monitoring system from the usability perspective. In this section we address three issues related to representing and viewing changes: difference generation, difference representation, and change summarization.

4.1 Difference Generation

Computing changes to a web page consists of two tasks: obtaining both the old and the new version of the page and comparing the two versions to display the difference.

The first task is related to the archival of web pages. Most of content providers only provide access to the most current version of their web documents. Although some content providers may maintain a history of their documents, for instance using the Revision Control System (RCS), to our knowledge few provide world-wide access to their archives. Furthermore, archiving every page accessed demands careful consideration on technical issues such as storage, indexing structure, retrieval efficiency and legal issues such as copyright protection on archival [12]. In WebCQ, we deliberately choose not to archive every page we access. Instead, for every web page accessed by WebCQ we at most keep one past copy that will be used to compare with the current version of the page and compute how changes have been made. Saving an old version of a page is not significantly different from a proxy cache server keeping a copy of a page until the page reaches its expiration timestamp. In contrast, the object cache update module discussed earlier is designed for maintaining and refreshing the object cache used by both the sentinel evaluation module and the difference generation module (recall Figure 1).

The second task is related to the difference generation from two versions of a web page. One way to compute the difference between two versions of an HTML page is to use *HTMLDiff* developed by AT&T [12, 11]. Similar to the UNIX *diff* utility [19], *HTMLDiff* takes two versions of a page as an input and produces a new and merged HTML document. The new document highlights the differences between the two versions by flagging the inserted text with bold or colored face and deleted text with a horizontal line cross over. Changes to existing text are treated as deletions followed by insertions. As pointed out by Fred Douglis and his colleagues [12], every invocation of the *HTMLdiff* may potentially consume significant computation and memory resources. Such resource overheads will restrain the number of difference operations a server can perform at a time, limiting the scalability of the system. In WebCQ, most of the sentinels, except *any change* sentinels, are targeted at tracking changes to a page fragment rather than the entire page. A page fragment can be a specific HTML object (such as phrase, list, and table), an arbitrary text fragment, or a specific component of the page (such as links, images, and words). In such cases, a simplified difference generation algorithm should be used to reduce the overhead of the general *HTMLDiff*.

There are at least three basic mechanisms for computing the difference between two versions of an HTML fragment.

- **Difference is flagged only when content (raw text in between a pair of tags) changes.**
  This method views an HTML fragment simply as a sequence of words. Markups and extra whitespaces are ignored for the purpose of comparison. Therefore when a text paragraph comprised of five sentences is changed into a list of five items (each sentence starting with a <LI>), no difference is flagged because the content of the paragraph version matches exactly the content of the list version, although the presentation structure has changed.

- **Difference is flagged when either content or structure changes.**
  This method views an HTML fragment as a syntax parse tree with tags as internal nodes and raw text as leaf nodes. The subtree-equality comparison algorithm is used to compute the difference between two versions of a fragment. Obviously, in this case a subtree representing a paragraph of five sentences will be different from a subtree representing a list of five sentences. Thus, the example of changing a paragraph with a list will be flagged, even though it was merely a format change.

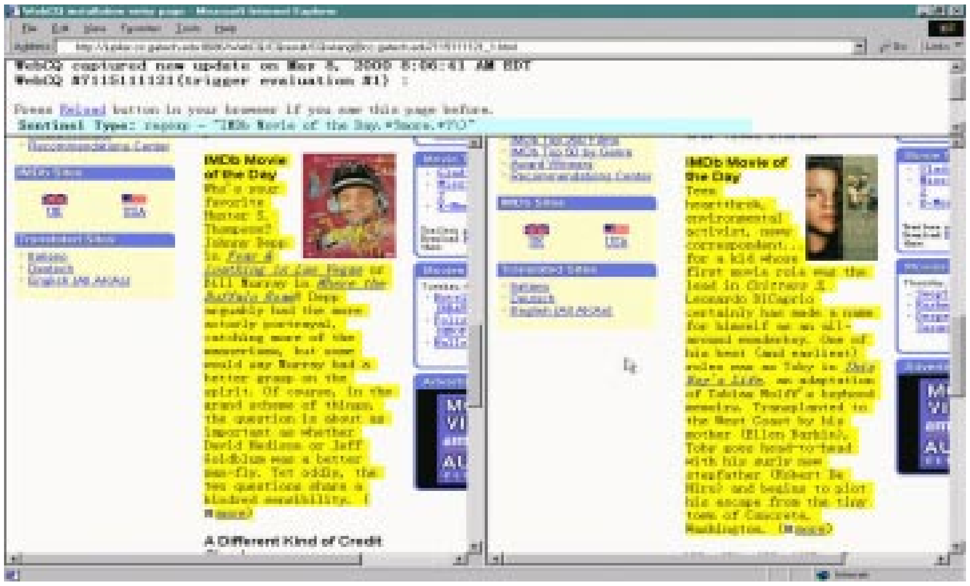
- **Different flags are used for content changes and structure changes.**
  This method can be implemented by applying both methods above. The application of the first method detects if the content has changed. The application of the second method tells if any format change has occurred. Thus, for the example of changing a paragraph with a list, the comparison will display no change to content but a change to formatting.

One problem with the second and the third method is that any small and trivial change to formatting, such as adding or removing a line-break tag `<br>` in the middle of a sentence or adding a paragraph begin tag `<P>` in between two sentences, will be flagged. After careful consideration, the first mechanism is used, in the current prototype of WebCQ, for computing the difference from two versions of an HTML page fragment.

An ongoing research effort is to design a structure-aware change detection and difference generation algorithm, called Sdiff. The technical detail and initial experimental results on Sdiff can be found in [34].

4.2 Difference Presentation

There are three popular ways to present the difference between two web documents [12, 2, 30]. The first approach is to *merge the two documents* by summarizing all of the common, new, and deleted materials in one document, as it is done in *HTMLDiff* [12] and UNIX *diff*. The advantage of this approach is to display the common parts of the two pages only once. There are also disadvantages of this approach. Merging two pages into one may introduce syntactic or semantic errors. It may also make the combined page difficult to read especially when there are a large number of changes.


WebCQ captured new update on May 5, 2000 5:06:41 AM EDT
WebCQ #7115111121(trigger evaluation #1) :

Press Reload button in your browser if you saw this page before.
Sentinel Type: regexp - "IMDb Movie of the Day.*more.*") "

IMDb Movie of the Day

IMDb Movie of the Day

A Different Kind of Credit

Figure 6: A difference presentation for the example sentinel on the IMDb front page

The second approach is to *display only the differences* but omit the common parts of the two pages. This approach is especially beneficial for the documents of large size and for the case where the two copies of a document have a lot in common. An obvious drawback of this approach is the loss of surrounding common context, resulting in sometimes confusing presentation of the difference.

The third approach is *side-by-side presentation* of the differences between two documents. A value-added feature to this approach is to highlight the fragments that are changed in both the old and the new copy of the page. This approach is considered the most pleasing way of displaying the differences between two documents. However, it is not an optimal approach for presenting the differences of very large documents, especially when the amount of differences is large as well.

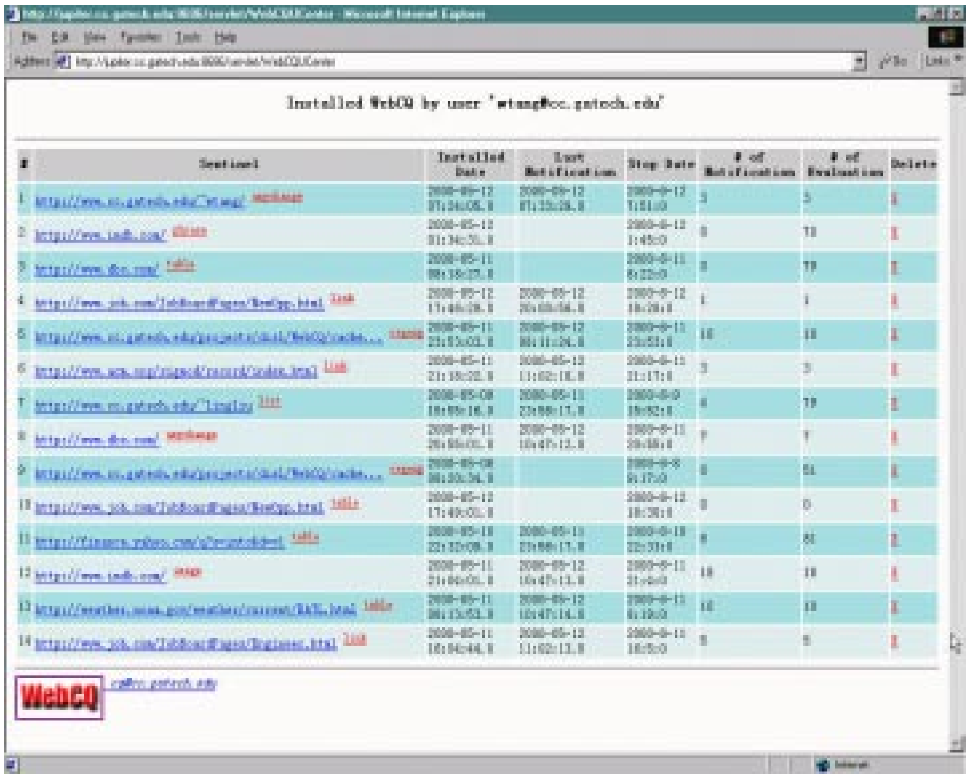
In WebCQ we use a hybrid approach that utilizes all three of the presentation schemes in selected combinations. For instance, we use the *Side-by-Side* for sentinels of phrase or regular expression type, and the combination of *Side-by-Side* with *Only Difference* to display differences for all-links, all-words, table, and list sentinels. We use the merge-two-documents approach for sentinels of image type. Figure 6 shows the difference presentation for the example sentinel on the Internet Movie Database front page given in Figure 4. The difference presentation module displays the new web page with the new text highlighted on the right window and the old copy of the web page with the deleted text highlighted in a different color on the left window. As one may have observed, in addition to the presentation of how the page has changed (i.e., the difference presentation), every detailed sentinel evaluation report in WebCQ also includes a brief summary of the sentinel, such as the sentinel type, the sentinel content, the title and the URL of the web page monitored by the sentinel, and a link to other sentinel specific information such as the last modification timestamp, the last evaluation timestamp, the number of evaluations since installation, and so on. Other example screenshots on difference presentation are available online at the WebCQ project page [41].

4.3 Change Summarization

Our experience with the change monitoring systems shows that one of the important usage improvement for change tracking and notification services on the Web is to provide at least one change summarization for each user, reporting the status of all sentinels installed by the user. Users will be given a choice of receiving a summarization notification for all sentinels they have registered with the system or one notification per sentinel. A change summarization report contains a list of summarization records. Each record presents a brief summary of the recent sentinel evaluation result, including:

- the URL of the web page being monitored,
- the type of change, such as insertion, deletion, update to the page, a new page, or a deleted page,
- the sentinel type,
- the last modification timestamp, showing when the page has changed,
- the last evaluation timestamp, displaying when the change is detected,
- the last notification timestamp, showing when the last notification was sent out,
- the timestamp when the page was last seen,
- the installation timestamp, showing when the sentinel was registered initially,
- the termination timestamp, showing when the monitoring sentinel will be terminated,
- the number of notifications since the installation,
- the number of evaluations since the installation, and
- the hypertext link to the detailed difference presentations of the sentinel.

Most of the information above is recorded and maintained by WebCQ during the sentinel processing. For all static pages, the HTTP `HEAD` request can be used to obtain the last-modified timestamp. However, if WebCQ has knowledge from a prior attempt that no timestamp is given, then HTTP `GET` is used to retrieve the body of the page and compute a simple checksum. Figure 7 shows an example change summarization.



Installed WebCQ by user 'wtang@cc.gatech.edu'

#	Sentinel	Installed Date	Last Notification	Stop Date	# of Notification	# of Evaluation	Delete
1	http://www.cc.gatech.edu/~wtang/ webchange	2000-05-12 07:34:06.0	2000-05-12 07:33:28.0	2000-6-12 7:51:0	3	3	✗
2	http://www.imdb.com/ [illegible]	2000-05-12 01:34:31.0		2000-6-12 1:45:0	0	73	✗
3	http://www.dbc.com/ table	2000-05-11 08:38:27.0		2000-6-11 8:22:0	0	79	✗
4	http://www.job.com/JobBoardPages/NewOpp.html link	2000-05-12 17:46:28.0	2000-05-12 20:08:56.0	2000-6-12 18:28:0	1	1	✗
5	http://www.cc.gatech.edu/projects/disl/WebCQ/cache... [illegible]	2000-05-11 23:53:02.0	2000-05-12 00:11:26.0	2000-6-11 23:53:0	18	18	✗
6	http://www.acm.org/sigmod/record/index.html link	2000-05-11 21:18:22.0	2000-05-12 11:02:18.0	2000-6-11 21:17:0	3	3	✗
7	http://www.cc.gatech.edu/~lingliu list	2000-05-08 18:09:16.0	2000-05-11 23:58:17.0	2000-6-8 18:52:0	6	79	✗
8	http://www.dbc.com/ webchange	2000-05-11 20:58:01.0	2000-05-12 10:47:12.0	2000-6-11 20:58:0	7	7	✗
9	http://www.cc.gatech.edu/projects/disl/WebCQ/cache... [illegible]	2000-05-08 08:20:34.0		2000-6-8 8:17:0	0	51	✗
10	http://www.job.com/JobBoardPages/NewOpp.html table	2000-05-12 17:40:01.0		2000-6-12 18:38:0	0	0	✗
11	http://finance.yahoo.com/[illegible] table	2000-05-10 22:32:08.0	2000-05-11 23:58:17.0	2000-6-10 22:33:0	6	81	✗
12	http://www.imdb.com/ [illegible]	2000-05-11 21:00:01.0	2000-05-12 10:47:13.0	2000-6-11 21:0:0	18	18	✗
13	http://weather.noaa.gov/weather/current/KATL.html table	2000-05-11 08:13:53.0	2000-05-12 10:47:14.0	2000-6-11 8:13:0	18	18	✗
14	http://www.job.com/JobBoardPages/Engineer.html link	2000-05-11 16:04:44.0	2000-05-12 11:02:13.0	2000-6-11 16:5:0	8	8	✗

WebCQ cq@cc.gatech.edu

Figure 7: A change summarization example

Change summarization listing all changed pages is particularly important when a user wants to track a large number of web pages and installs tens or hundreds of sentinels. This is simply because using individual email messages, each corresponding to one sentinel, would easily overwhelm the user. Such situations may be aggravated when multiple updates to a page were reported in a sequence of email notifications before a user could process the mails.

5 Change Notification Service

Change notification is a software facility that provides mechanisms for notification of information changes. Many notification services that provide notification of changes to Web pages have a few features in common and vary in several other aspects.

The functionality of a notification service is divided into three components: *what to notify* (what types of information should be delivered for notification), *when to notify* (when users should be notified of changes to the web pages they track), and *how to notify* (how timely delivery of notification can be guaranteed). Beyond this common underlying framework, most systems diverge, from both the architectural design to the technical solutions used to address these three basic questions.

In the rest of the section we review a list of important issues in designing and engineering a notification service and discuss the design decision and the mechanisms used in WebCQ notification service.

5.1 Reference Architecture

One of the design goals of a notification service is to make it reusable and configurable, so it can easily be incorporated into other system architectures.

In designing a notification service, the first design choice is to decide whether to build the notification service as a standalone service or as a component of the WebCQ information monitoring and tracking system. In the standalone architecture, the main components are comprised of information source, source change event observer, notification manager, and wrapper set. The wrapper set is bundled with the notification service. This architecture is considered "loosely coupled" with the source change event observer. The observer may be provided by, or combined with, the source and connect to the notification manager using the HTTP protocol. The notification service can also be designed as a component of a system. In this case, the notification service may cooperate with other system components in a tightly coupled fashion. The source observer is usually provided by the tracking system using a set of source wrappers or data extractors. For example, in WebCQ we choose to implement the notification service as a component of WebCQ so that the trigger facility can be reused to implement a richer set of notification conditions such as "every 5 times when the trigger condition becomes true". The WebCQ notification service can reside on the same server as the WebCQ engine or reside on a machine connected to the WebCQ server within a local area network. Compared with the "loosely coupled" notification service, a "tightly coupled" one has the advantage of less network communication overhead, but it loses some flexibility and harder to be implemented generically.

The main task for a notification service that cooperates closely with a change detection robot is to synchronize with the robot when changes to web pages are observed, and provide efficient notification of changes to appropriate users or user community. In addition, the notification service may enforce application-specific delivery constraints, such as delivery based on specific user-defined priorities, access controls, and security constraints.

5.2 Framework Design: Building Blocks

The basic building blocks for a notification service are the suite of mechanisms that identify when a notification should be sent, how a notification is sent, and what should be included in a notification.

When to Notify –

Eager notification, periodic notification, and on-demand notification are the three mechanisms we have investigated.

- *Eager notification* advocates that any interesting change once detected should be delivered to the client immediately. An obvious advantage of eager notification is to guarantee the minimum latency from the time changes are detected to the time the notification is sent out. Periodic notification and on-demand notification, on the contrary, are based on a deferred notification scheme.

- *Periodic notification* for changes have the advantage of ensuring that the user will have relatively up-to-date information about the pages being tracked, with a bounded freshness value. However, periodic notification requires network connectivity at the time the notification is sent out and for the duration of the update notification which is proportional to the number of pages to be monitored.
- *On-demand notification* promotes the idea of having users fire the notification request when they want to review the change tracking results. On-demand notification will not be effective if the number of web pages being monitored is large, as the user may become overwhelmed with a large number of changes.

In WebCQ, the notification service runs on the server side, thus on-demand approach is not considered. We choose the periodic notification over the eager notification because immediate notification may not scale well when a user registers a large number of sentinels for tracking changes to the web pages. The periodic interval is defined by the notification condition given by the user at the sentinel installation time.

How to Notify –

There are three issues involved with how to notify users of changes: notification initiation, notification mechanism, and change presentation.

- *How to initiate a notification*

  In WebCQ, the notification can be provided by either a server-initiated push delivery or a client initiated pull delivery.
- *How to carry out a notification*

  In the current WebCQ system, both email and web pages are used as mechanisms for server-pushed delivery. For client-pulled delivery, WebCQ currently only provides web pages with a search interface to allow users to query and view the change reports from anywhere in an ad-hoc style. Other means of notification include pager, fax, cell phone, or PDA. More notification methods will be provided for the WebCQ service.
- *How to present changes to users*

  Prioritization and summarization are the two major mechanisms we have investigated for ensuring an effective and pleasant presentation of change notification to the users. Prioritization allows the notification service to bring web pages of particular importance to the attention of a user at the first glance. Summarization allows the users to see an overview briefing of the current status of all web pages to which they track changes before going to the detailed change report of each sentinel.

We have investigated three different mechanisms for prioritizing a list of notifications on a per-user basis. The first one is to order a list of sentinels in a change summary report in a reverse chronological order of modification dates, with the most recently modified page on the top of the report. The second mechanism is to let users assign a priority number for each sentinel they install and then use the user-defined priority numbers to determine the order of sentinels in the change summarization report. The third approach is to let users prioritize the topics of interest and within a topic the web pages are organized in a reverse chronological order of modification dates.

Often users wish to know the amount of changes to the web pages before visiting the detailed difference presentation document. Examples of such types of summarization include the type(s) of changes found, the number of links added or deleted, the fraction of text that have been modified, the ratio and frequency of changes that happen to a web page, and so forth.

What to Notify –

We observed that many users prefer to have the WebCQ system track changes to the web pages of interest and notify them of the changes with some summarization in addition to individual change report per sentinel. In WebCQ, we provide support for the following four types of notification reports:

- Detailed sentinel evaluation report, displaying the differences side by side or in a merged document;
- A change summary report per user, displaying the list of sentinels installed by a user, each with a brief summary of the most recent change detection status. This approach is especially useful when the number of web pages being monitored per use is not small.
- A change summary report per user grouped by topic of the web pages being monitored. This approach scales much better than the previous one when the number of web pages monitored per user is getting large.
- A dedicated web notification site where a query interface is provided to allow users to search and find the change summary reports or the detailed change reports which match given conditions. This approach is particularly useful for busy users or users with Wireless devices and intermittent Internet connectivity.

The WebCQ notification service is utilized in at least three ways: First, we use the notification service to notify users of their change monitoring sentinel subscription, including both the subscription of the WebCQ system, the expiration of their installed sentinels, and the expiration of their WebCQ subscription. Second, we use the notification service to send the users the update alert and the change summarization of the web pages being monitored. Third, the WebCQ notification service is designed to allow an external event observer to be loosely coupled with the change notification. Such coupling enhances both the tracking capabilities of the WebCQ system as well as the update monitoring functionality. It allows us to provide the WebCQ users not only the service for alerting and notifying users of the changes to Web pages of interest, but also the capability of taking appropriate actions to react to the changes. In the current implementation of WebCQ, we require the users to explicitly register the event observation function to be called and the method of remote invocation when a notification service is subscribed separately from the WebCQ system.

6 Sentinel Grouping

In WebCQ, ideally there should be no restriction on how many Web page sentinels a user can create and how many users can register with the system. However, the system scalability and performance problems may arise when tens of thousands or millions of sentinels (continual queries) are running concurrently (500 users with an average of 20 sentinels per user will reach ten thousand CQs). Thus, one of the main challenges that WebCQ needs to address is the *scalability* of the sentinel processing strategy, namely how to guarantee the responsiveness of a Web monitoring service in the presence of a large number of concurrently running sentinels (continual queries).

Sentinel Grouping is an optimization technique developed in WebCQ. It promotes the reduction of repeated computation and processing by classifying sentinels into groups. Sentinels can be grouped together if they share the same target and have similar trigger conditions or their sentinel types and sentinel objects can be processed in one-pass. Consider a simple example. Assume we have n users who want to monitor new cancer trails over the National Cancer Institute (NCI) Web site. Each user is interested in monitoring the new cancer trails of m types of cancers separately. Thus we need to run $n * m$ Web page sentinels. Let T denote the time unit for the system-controlled polling interval (e.g., every 2 hours). Assume that k is the number of distinct types of cancers being monitored by the n users. If we group all $m * n$ installed page sentinels by the cancer type, then the number of concurrent trigger testing against the remote NCI Web site (www.nci.org) can be reduced from $m * n$ (one per sentinel) to k. Assuming a reasonable overlap of user interests in the types of cancers, such as breast cancer or prostate cancer, k is often significantly smaller than $m * n$, thus sentinel grouping will provide the WebCQ system a dramatic saving on both the network cost and the overall system overhead.

6.1 Grouping Strategy

The key idea behind the sentinel grouping is based on the general premise that a large number of Web page sentinels are often similar in terms of the target URLs, sentinel types, or sentinel objects. We distinguish three different classes of sentinels.

- The first class of sentinels are those that monitor the same Web page for any change or for a change on the same sentinel type and the same sentinel object. These sentinels are often captured in the same continual query (CQ) expression. We call this category of sentinels *the sentinels of the same CQ expression.*

- The second class of sentinels refers to those that monitor the same Web page (i.e., with the target URL) but monitor different fragments (i.e., different sentinel objects) in a page with the same sentinel type (e.g., table sentinel). These sentinels are captured in different CQ expressions. We call this category of sentinels *the sentinels of different monitoring objects*. For example, some students may be interested in monitoring the list of senior 4000-level courses in the course listing web site of College of Computing at Georgia Tech, whereas others may want to monitor 8000-level graduate courses in the same page.
- The third category of sentinels refers to those that may track changes in the same page but with different sentinel objects of interest and different sentinel types of interest. For example, some may want to monitor the CS4400 course home page for changes in the course schedule table, and others may be interested in monitoring the same page for changes in the paragraph on grading policy or the list of reference textbooks.

In this section we discuss the index structure for sentinel grouping in WebCQ and the implementation method based on the concept of sentinel signatures.

6.1.1 Index Structure for Sentinel Groups

In WebCQ the index structure we use to store sentinels is a hash table with the URL of the monitored page as the hash key. Each bucket of the hash table contains a list of groups, one group per URL. Sentinels that share the same target URL are hashed into the same bucket accordingly. Figure 8 shows a sketch of the data structure used for grouping sentinels of similar CQ expressions. Each group has a corresponding URL, an MD5 hash of the most recently cached version of the web page, and pointers to the nine sentinel types specified in Section 2. Each pointer points to a linked list of n_i constant tables for the corresponding sentinel object type i. A constant table contains a set of sentinel identifiers that share the same URL target, the same sentinel expression, and the same table fragment being monitored. For example, the pointer to the list sentinel type ST_5 may have a linked list of n_5 constant tables and each refers to one concrete `List` fragment being monitored in the page.

Whenever a new page sentinel is installed, the system will first capture it in a continual query expression. Then this sentinel will be hashed into the corresponding bucket, depending on its target URL and its sentinel type. A constant table is created for this sentinel with an entry added to record the sentinel identifier, the corresponding CQ expression, among others, for this newly installed Web page sentinel. If the target URL is new, or the bucket for this sentinel type has not been created, it will first create the MD5 hash signature of the page and then create the bucket for the sentinel type.

Concretely, sentinel groups in WebCQ are constructed as follows:

Step 1: The URL for each sentinel is hashed into a bucket. If the sentinel is the first entry in the bucket, a new group is created, and the sentinel is added to the group, as described in step 3.

Step 2: The page is then retrieved from the proxy and an MD5 hash is computed for the page signature.

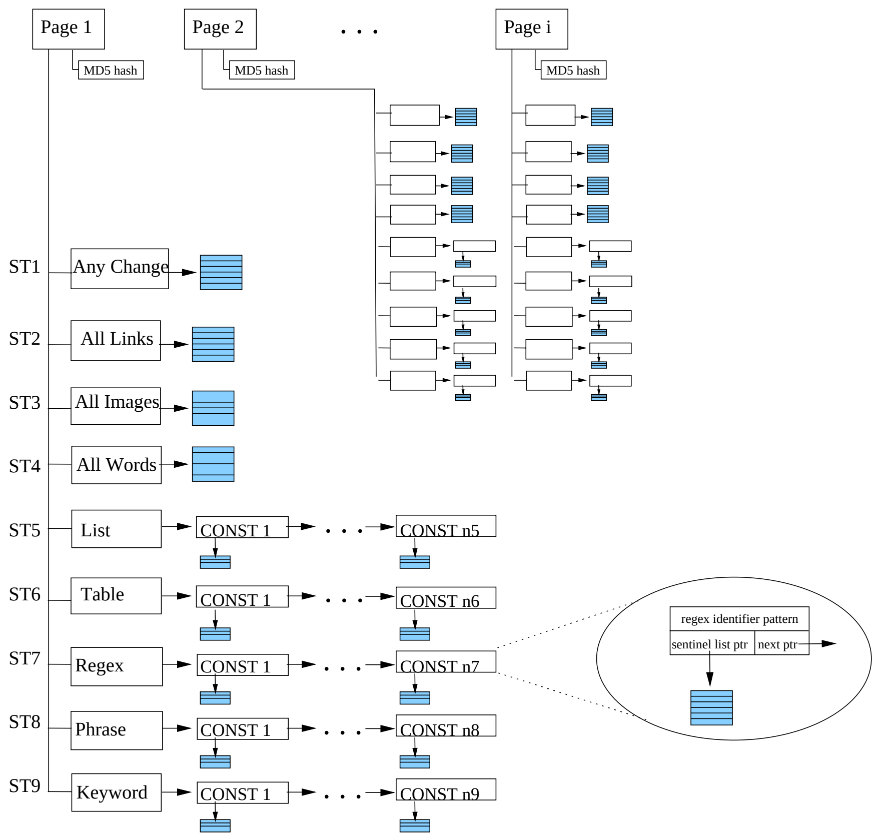


Figure 8: Grouping Structure for Web Page Sentinels sharing the same CQ expressions

Step 3: If there are other entries in the bucket, the sentinel is added to a pre-existing group that was already setup to monitor that URL.

- If the sentinel is an `anychange` sentinel, the sentinel ID is added to the `anychange` sentinel table in the index structure.
- If the sentinel is a table, list, paragraph, or regular expression change sentinel, the specific sentinel specification must be compared with each constant in the linked list of constants corresponding to that sentinel type. If a match is found, then the sentinel ID is added to the corresponding constant table of sentinels. If no match is found, then a new constant identifier is generated, and inserted into the linked list of constants for that sentinel type. A constant table is created accordingly, and the sentinel ID is added to this newly created constant table.

An obvious extension of the sentinel grouping strategy described in this section is to explore the alternatives for grouping sentinels. For example, This is especially desirable when a large number of CQs all have different trigger conditions but share some partial selection predicates. One idea is to choose the most selective predicate as the key predicate for grouping. We will discuss this topic in the next section.

6.1.2 Executing Sentinels Using the Group Index

Grouping sentinels over the same page can greatly reduce redundant operations. The process of executing grouped sentinels, managed by the WebCQ Daemon, operates in the following manner.

- **Step 1:** A URL is chosen from the list of all monitored URLs, and the current version of the page is requested from the proxy.
- **Step 2:** The sentinel group for the URL is retrieved from the metadata manager.
- **Step 3:** An MD5 hash is created for the page as the new page signature, as described in Section 6.2, and compared with the previous signature of the page. If the signatures match, no further processing is required for this group.
- **Step 4:** The previous version of the page is loaded from disk.
- **Step 5:** If there is any `anychange` sentinel (or all images, all links, all words sentinel) installed over this page, the old version and the current version of the page are compared to detect difference; if no change is found, then the processing for this group ends. Assuming a change is discovered, all `anychange` sentinels are notified.
- **Step 6:** Similarly, for the rest of sentinel types, check the linked constant list and for each constant table in the linked list, repeat the following process for each sentinel identifier:
  (1) load the object cache for the sentinel object; (2) locate the corresponding sentinel object in the current page, (3) compare the new version of the sentinel object with the cached version for difference, and if a change is found, call the trigger evaluation module or Change summary generation module, otherwise no change is found, and exit.

An independent garbage collecting process is provided to clean up the local cache, and remove versions of pages that are older than the current version.

6.2 Generating Page Signatures Using MD5

Change detection can be considered as pure overhead when comparing identical versions of a web page. Even in character-based difference algorithms, such as Unix `diff`, each character of the page must be compared for each new version of the page. Many HTML or XML difference algorithms require the Web document to be converted into a tree structure before applying the diff algorithm. When nothing has changed, the time spent to convert a page from text into an in-memory model (such as a DOM tree), to load the previous copy of the page into memory from disk, and to compare the new page with the cached copy, is completely wasted.

To eliminate the unnecessary local I/O and diff comparison between two identical versions of a page, a signature for each page can be kept in memory with the sentinel group for that page. Then, when a new version of the page is retrieved from the web, a new signature is computed and compared with the signature of the previous version of the page. Only if the signature has changed does the local version need to be loaded into memory to determine what has changed to the page.

Typically high quality signature algorithms, such as MD5, are somewhat time-consuming to compute. In order for signatures to provide a net benefit, it should be at least cheaper to compute a signature for a page than it is to load the page from disk into memory. The probability that the page has changed should also be lower than the ratio of time to compute the signature to the loading time for the page. Signature computation can be done in parallel on multiple processors or computed while other processes are blocked on I/O. In our initial experiments, computing an MD5 signature is less than one-third as expensive as loading a DOM tree of the page from disk, providing a net benefit to using the MD5 signature algorithm on pages that do not change on each request.

7 Experimental Evaluation

In this section we report four sets of experiments that evaluate the performance of the WebCQ change detection algorithms and sentinel grouping method. The first set of experiments compares the performance of different types of sentinels. The second set of experiments examines the costs associated with sentinel grouping. The third set of experiments demonstrates the benefits gained from sentinel grouping. The fourth set of experiments reports the performance comparison of the WebCQ original implementation with the WebCQ grouping implementation.

All experiments were run on a SunFire 280, dual 733 MHz processor server, with 8GB RAM, 72 GB disk on a RAID 5 controller, and gigabit local Ethernet. The software was implemented in Java and run with the J2SE 1.3.1 from Sun, using the server virtual machine. All experimental results discussed in this paper were averaged over more than ten execution runs over the chosen test data sets.

The set of data used for these experiments were pages gathered from the Yahoo! news portal and `my.yahoo.com` during December of 2001. These pages are chosen for our experimental evaluation as they represent relatively complex web pages, and they support a large variety of sentinel types, and are constantly changing. Page size in this set varies between 30KB and 60KB, averaging 47KB; the number of nodes in each page varies from 1400 nodes to 2171 nodes, averaging 1672 nodes.

7.1 Performance Comparison of Different Types of Sentinels

A major component of the WebCQ system is the algorithms used to detect changes in web pages by comparing the previous version of a page with the current version after both versions are loaded to memory. Figure 9 shows the execution time over the randomly generated data set of four different sentinel types: (1) any change sentinel over content using character-by-character comparison (any change sentinel for short), (2) link sentinel, (3) table sentinel, and (4) any change sentinel using MD5 signature (MD5-based any change sentinel). The any change sentinel monitors the entire content of a page. It

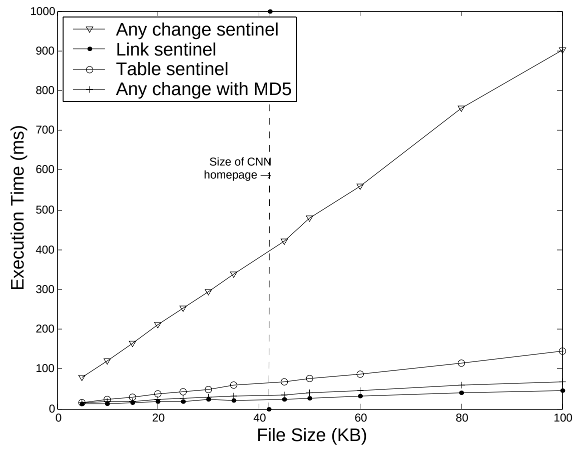


Figure 9: Processing time for detecting changes for different type of sentinels

parses the page character by character like the Unix `diff` utility and when a change is found, it returns the location of the object that has been changed. The link sentinel scans the page, locates all of the links, and checks to see if any have been added or removed from the page between versions. The table sentinel locates the table sentinel object in the web page using a regular expression, and compares this current version of the table sentinel object with the same table from the previous version of the page. If a change is found, it returns the location of the object that has been changed. The MD5-based *any change* sentinel computes hashes for the current version of the document, and compares it with the cached MD5 signature of the previous version of the page. Although technically there is a possibility for any hash function that two separate documents may have the same hash value (MD5 signature), it is well known that MD5 has extremely small probability for hash collision, especially for two versions of the same document. One benefit of the MD5-based *any change* sentinel over the original *any change* sentinel is that the signature of a page may be kept in memory, and the previous version of a page need not be loaded if the signature for the new version matches the signature of the previous version cached in memory. Figure 9 shows two interesting observations. First, some sentinels are inherently more costly than others, but the cost for all sentinel types increases uniformly with the size of the page being monitored. Second, the experimental results validate the intuitive result that the MD5-based *any change* sentinel performs significantly faster than the *any change* sentinel does.

7.2 Cost for Grouping Sentinels

While grouping sentinels together can save memory and execution time, there is some overhead for creating groups. Sentinel grouping is performed in three steps: creating a group, inserting new sentinels into a group, and loading and computing the MD5 hash for the group. Figure 10 and Figure 11 demonstrate the relative costs of grouping sentinels depending on the group size. The pages used for this experiment were based on the data collected from *Yahoo!*.

Figure 10 shows that there is a cost for grouping sentinels. When the group size is one, namely no sentinel is similar and thus can be grouped together, the cost of grouping increases linearly with respect to the number of sentinels. As the size of the group increases, the cost of grouping drops quickly. With group size of 10, the grouping cost is already considerably low. The cost of grouping sentinels decreases for larger group sizes because adding a sentinel to a group is the cheapest of the three steps, while the other two steps only need to be performed once per group. The cost of creating a group and computing the MD5 hash signature for the target page associated to the group can be amortized over the size of the group.

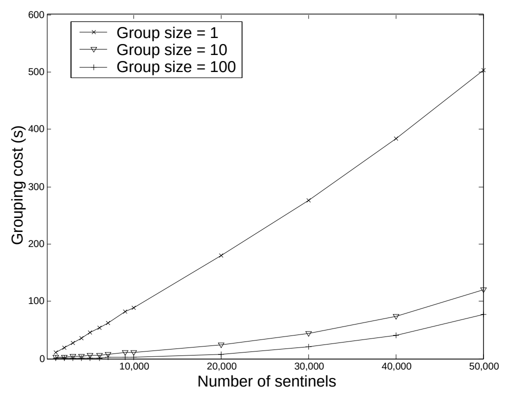


Figure 10: Grouping cost for varying group sizes

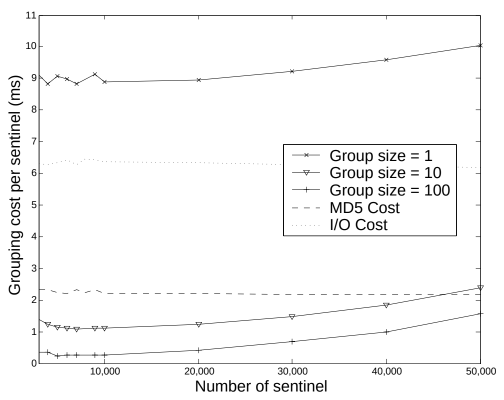


Figure 11: Grouping cost per sentinel

Figure 11 shows the grouping cost per sentinel. There are five lines in Figure 11: three lines listing the average cost of grouping sentinels into groups of different sizes, and two lines showing the average cost of the local I/O required to retrieve a document and the cost to compute an MD5 signature for a document. As expected, the average time required to group sentinels hold steady as the total number of sentinels increase. Group time for a particular sentinel depends on the average number of sentinels per group rather than the total number of sentinels in the entire system. This behavior is desirable for any system that must scale to millions of sentinels.

7.3 Performance Gains for Sentinel Grouping

The third set of experiments measure the benefits of sentinel grouping. These experiments were run over pages from the data set gathered from *Yahoo!*. Figure 12 shows the cost of executing any change sentinels without any form of grouping. It is clear from this experiment that the I/O time required to load the two versions of a Web document to be compared is very high compared to the overall execution time.

While I/O is obviously the major factor in sentinel evaluation, in the next experiment we want to show that sentinel grouping may significantly reduce the costs in evaluating sentinels for grouping with both uniform distribution and Zipf-distribution. It is expected that Web-page sentinels be distributed in a Zipf-like pattern, in which a few pages have many sentinels whereas most pages have only a few

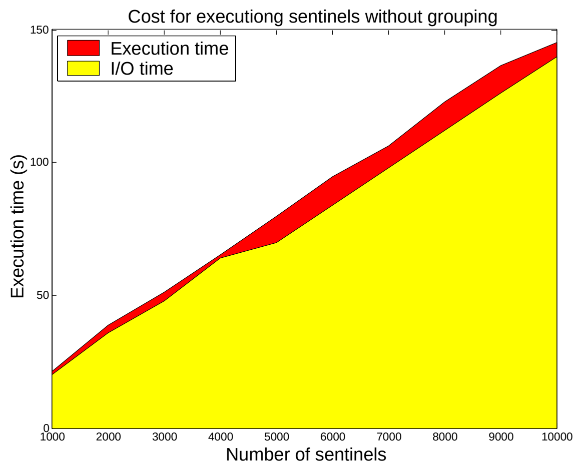


Figure 12: Execution time without grouping

sentinels installed over them. Zipf-like distributions are expressed by the equation Ω/i^{α}, describing the popularity of page i, where Ω is a normalizing constant, i is the i^{th} most popular page, and the exponent α describes the slope of the curve. This is typically the distribution seen by general proxy servers [7]. Figure 13 compares grouping effects for Zipf-like distributions of sentinels on web pages with no grouping and grouping where the group size (gs) is ten.

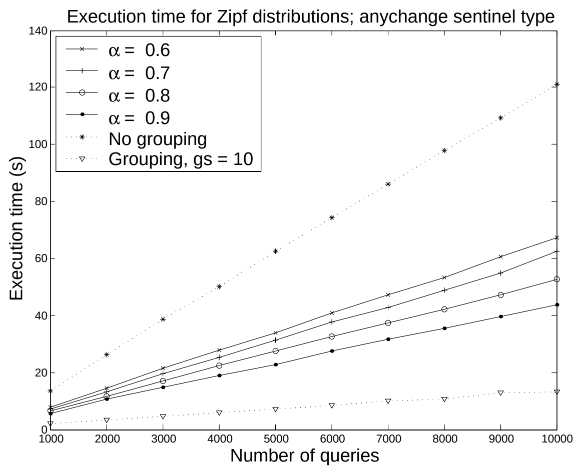


Figure 13: Grouping with Zipf and uniform distributions

Figure 14 shows the total time required to execute several thousand triggers with no grouping or grouping with 5, 10, and 100 sentinels per group under uniform distribution. For 5,000 sentinels, the total execution time with a group size of 5 is 14.2 seconds, while it is 7 seconds for a group size of 10 and 1.3 seconds for a group size of 100 sentinels. Without grouping, however, it takes over 62 seconds to process the same number of sentinels.

Figure 15 shows the throughput for various levels of grouping. For 5,000 sentinels, without grouping

WebCQ can process 80 sentinels per second. With grouping, the WebCQ system achieves throughput rate of 351 sentinels per second when group size is 5, over 4 times more sentinels can be processed per second. Similarly, the throughput rates are 708 sentinels per second for group size of 10, and 3,698 sentinels per second for group size of 100, which are almost 9 times improvement for group size 10 and over 46 times improvement for group size of 100.

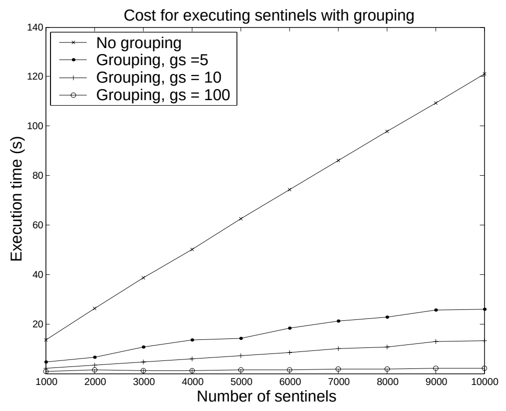


Figure 14: Execution time for grouped sentinels

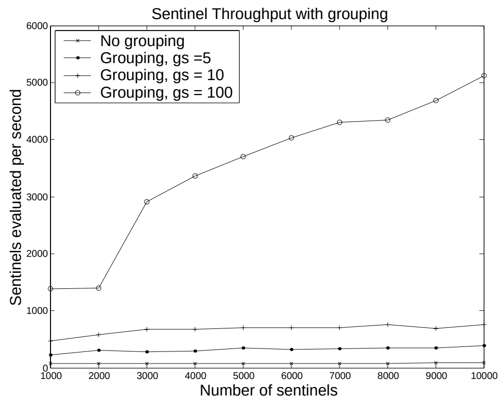


Figure 15: Throughput for grouped sentinels

7.4 Comparing WebCQ with Grouping to WebCQ Original

Work on sentinel grouping in WebCQ began due to several limiting factors in the first implementation of WebCQ. The original WebCQ was a novel implementation of a change detection system for web pages. It incorporated several ideas from our earlier work on the CQ project [26], applying the concept of continual queries to data available over the web.

However, the first implementation of WebCQ consumed too many resources in managing and executing sentinels. The system relied on the Oracle 9i database system to manage all of the metadata, including user profiles, sentinels, and the object cache. Each sentinel execution required multiple accesses to the database. Execution of individual sentinels relied on external tools that were not optimized for large scale processing. To alleviate the problem, in WebCQ grouping implementation we moved the management of most information out of the database and into main memory. The historical data cache was moved to the file system where we also keep extensive logs to guard against system failures. Furthermore, we implemented our own change detection and management tools directly in the system architecture, providing a seamless intraprocess workflow for all operations. The combined improvements in sentinel algorithms, I/O sensitive operations, and sentinel grouping have contributed to achieving one to two orders of magnitude improvement over our initial version, as documented in Figure 16.

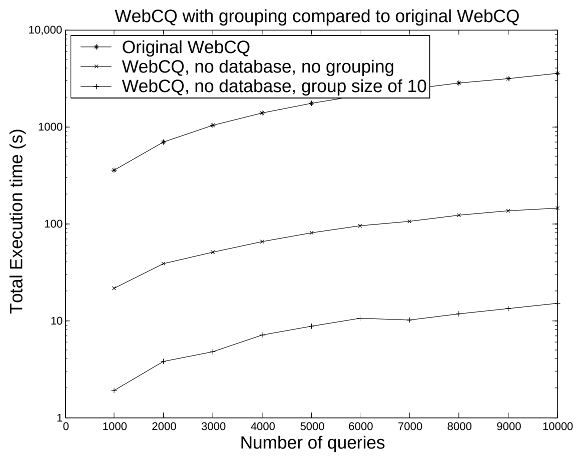


Figure 16: Comparison of WebCQ with grouping vs. the original version of WebCQ

8 Discussion

We have presented the design and development of the WebCQ system, including the sentinel grouping technique for scalable processing of Web page sentinels. We also reported our initial experimental results showing the benefits and the effectiveness of the WebCQ approach to large-scale information monitoring on the Web. In this section we discuss a number of important ongoing issues and our plan for extensions along several dimensions.

8.1 Monitoring and Notification Latency

Latency is an important factor in WebCQ information monitoring and delivery service. In WebCQ we use the concepts of change detection interval and notification interval to model the overall monitoring latency. Typically, a *source-specific change detector* (whether it is source-initiated or polling-based) sets the sampling latency, bounding the delay from source event occurrence until change events are detected. We refer to such delay as the *change detection interval* or the *sampling latency*. This "change detection interval" determines the minimum resolution of event frequency. The notification manager controls the latency from change detection until notification. We refer to such latency as the *notification latency* or *notification interval*, which determines the maximum resolution of notification frequency.

Achieving low sampling latency (less than notification latency) may require source-initiated change detection. Our experience with tracking changes in Web pages shows that it is hard to decide what polling intervals are the most appropriate or efficient. Inappropriate setting of polling interval is one reason polling-based change detection systems can be inefficient. But when source-initiation is unavailable, which is the case for most web sources, the polling approach is the only choice.

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

Second, change notifications can be sent directly from a source to the end-receivers, or through an intermediate server using either "end-to-end" delivery or "store-and-forward". In very large-scale environments such as the Internet, the use of intermediate proxies is common.

Third, in terms of notification delivery constraints, the *Notification Service* is responsible for ensuring certain guarantees before delivering a notification to the recipients. For example, whether the notification supports real time delivery or not, or whether to resend when messages are lost. Currently, we use a "no retry" policy in the WebCQ system for simplicity. Performance improvements on reducing both the sampling and the notification latency are under way. In addition, security constraints are also an important responsibility of the notification service. For example, in case of a firewall, the service needs to have appropriate knowledge of how to pass messages through the firewall, for example via proxies.

Finally, to ensure correct notification, the notification manager maintains a subscription list, which is an editable and auditable "first-class" object. An initial verification is performed at the time of a notification service subscription. For example, in WebCQ, the email addresses are checked upon user registration in order to guarantee that the correct email addresses are used for the users to receive notification. We are interested in incorporating more advanced quality of service (QoS) properties into the notification service of the WebCQ system.

8.2 Scalability

According to B.C. Neuman [31], "the scale of a system has three dimensions: numerical, geographical, and administrative. The numerical dimension consists of the number of users of the system and the number of objects and services encompassed; the geographical dimension consists of the distance over which the system is scattered; the administrative dimension consists of the number of organizations that exert control over pieces of the system."

It is widely recognized that the scale can affect many components of the system: naming, authentication, authorization, accounting, communication and the use of remote resources. Scale also affects reliability: when a system scales numerically, the likelihood that some host will be down increases; when a system scales geographically, the likelihood that some hosts fail to communicate with others increases. Scale can affect performance as well, in terms of system load and communication latency.

General solutions to scalability fall into four categories: replication, distribution, caching [31], and optimization. They can be both hardware-related and software-related. We considered scalability when we began to build the WebCQ system:

Replication : WebCQ can replicate the meta database (user info + registered WebCQ) to improve the response time and availability of the whole system (under development). The placement of replicas is an ongoing research project.

Distribution : WebCQ can direct different user requests to different servers. One example is to distribute using a hashing function on user's email addresses. Another example is to distribute according to different user request domain, e.g., stock watch, book watch, and flight price watch. Some software components can also be spread across multiple servers, such as email services (under development).

Caching : WebCQ system uses cache proxy architecture to increase scalability. When the number of users and number of user requests increase, it is possible that different requests have interests in the same Web page. Web caching has been used for quite some time in general Web server and browser architectures. In WebCQ, we cache the accessed Web pages and store them locally for later use. We use a policy file to control the frequency of refreshing the pages. The default is one day. When we serve a new user request, we first look locally in the cache for the page, if it is within the refresh threshold, we retrieve it from the cache, otherwise, the proxy server fetches the new content from the source site. From our experience in building the Continual Query system, we assume local access is always faster than communicating with remote servers.

Optimization : WebCQ develops the sentinel grouping techniques that optimize the concurrent processing of large number of sentinels by grouping sentinels of similar structure together to reduce the duplicated computation. Our initial experimental results (recall Section 7) demonstrate the effectiveness of sentinel grouping for scalable processing of sentinels.

8.3 Availability

The WebCQ system is a Web-based middleware system. It involves Web servers, proxy servers, and other software components. The whole system can be viewed as a cluster of software components. The technical challenge and the attempts made in WebCQ for ensuring availability can be summarized as follows:

- No bottlenecks to scaling: new servers can be added and configured dynamically; requests can be distributed to multiple software components.
- No single points of failure that could impact availability: requests must automatically failover to working components (by replication).
- Transparency to applications and application developers: the system software takes care of the replica management and consistency control.
- Single-system image to system administrators: the cluster is managed as a single logical resource.
- Hardware and operating system independence: WebCQ uses Java to achieve maximum software portability and independence

The WebCQ system has a presentation front end (user interface) and a back end (server end). For the front end, different mechanisms are used to increase the scalability and availability. For example, one commonly used approach to direct user request to different servers is "DNS Round Robin" between the Web clients and the Web servers. This provides simple form of load balancing and failover schemes. Another clustering technique is for dynamically generated pages that go between the Web server and the Java Servlet engine. A Java servlet is capable of keeping a user session or a persistent database connection (to avoid database connection setup cost). Application states can be stored in the server meta database (which can be replicated). However, there are many open issues in this endeavor. One of our ongoing research efforts is to investigate the state of art in availability research and develop a general solution for improving the availability of information monitoring systems such as WebCQ.

9 Related Work

There has been considerable research done on data update monitoring in databases. Powerful database techniques such as active databases and materialized views have been studied extensively. These techniques have been proposed primarily for "data-centric" environments, where data is well organized and centrally controlled in a database with a close-world assumption. When applied to an open information universe as the Internet, these assumptions no longer hold, and some of the techniques do not easily extend to scale up to the distributed interoperable environment.

Active Databases
Event-Condition-Action (ECA) systems are rule-based programs in which an event triggers the testing of a condition, which in turn (if true) triggers an action. Most of the active database systems [43] provide facilities that allow users to specify, in the form of ECA rules, actions to be performed following changes of database state. Some popular active database research prototypes include Ariel [15], Postgres [39], and Starburst [14]. These systems provide powerful rules and allow general events, conditions, and actions, and therefore are more difficult to provide efficient implementation. Despite the conceptual generality, rules have been so far supported in a fairly restricted form in practical systems (e.g., by built-in triggers in relational database management systems such as Oracle, Sybase, and Informix). Active queries, introduced in Alert [37], are more sophisticated than database triggers, since they can be defined on multiple tables, on views, and can be nested within other active queries. However, active queries rely heavily on a number of extensions specific to the IBM Starburst DBMS [14].

Compared to the state-of-art of research in active databases, the WebCQ system differs in three ways: First, the WebCQ system targets at monitoring and tracking changes to arbitrary web pages. Second, WebCQ monitors data provided by the content providers on remote servers, and WebCQ monitoring and tracking service requires neither the control over the data it monitors nor the structural information about the data it is tracking. Whereas active databases can only monitor structured data that reside in a database. Third, the WebCQ system provides efficient and scalable proxy service as well as grouping techniques for trigger processing, and it can handle a large number of concurrently running sentinels on a large number of web pages.

Web-based Push-enabled Systems

There are several systems developed for monitoring source data changes. One type of systems is the extension of Web search engines or search software by monitoring URL changes and notifying the users whenever the URLs of the data sources of interest have changed. A representative system is the NetMind (`http://www.netmind.com/`, formerly known as URL-minder), which provides keyword-based and phase-based change detection and notification service over Web pages. Compared with WebCQ system, NetMind does not provide fine grain of specification on triggering conditions. The content-based condition is still keyword-based. They do not support up-to-minute update monitoring either. The second type of monitoring systems is the application-specific change notification systems such as E*Trade alert facility, Amazon.com new book notification service, and so on. The most representative one of this type is the Stanford news service SIFT [45], which filters and notifies Internet news of interest through user preferences on news items and news groups. The problem with these systems is that they are tailored to a particular (type of) data source or application, and consequently do not have the generality and extensibility of WebCQ. The third type of projects is the change detection over wrapped Web pages, such as the $C3$ [9] project at Stanford and OpenCQ [24]. C3 develops a query subscription mechanism that allows users to subscribe the data sources they are interested in detection of changes as well as query over the change databases. The C3 system periodically goes to the subscribed web sites, fetches the pages, applies the HTML-based `diff` functions to derive the types of changes, and archives the changes in the change databases. The OpenCQ system is specialized in tracking changes over structured information. Therefore, wrappers [22, 23] are needed to transform Web pages into structured data format and then install continual queries over the structured data. In contrast, WebCQ is designed to monitor and tracking arbitrary web pages directly. No structured formats are required. The advantage is obvious with respect to the system scalability and effectiveness. A disadvantage compared with OpenCQ is that it does not support semantically-enhanced and fine-grained information monitoring, such as notify me when both IBM stock price and Microsoft stock price drop by 10% within one week.

Other Related Research

There are two other related lines of research. The first is query and trigger optimization by grouping. The second is continual query systems for data streams.

TriggerMan [16] proposed to group a set of triggers for optimizing trigger processing over relational data. Conquer [26] and NiagaraCQ [10] extended the idea of trigger grouping. Conquer allows grouping at trigger level, query level, and notification level. NiagaraCQ evaluated query grouping through simulation. These systems work with structured or semi-structured data such as relational databases and XML data. In contrast, WebCQ can work with arbitrary Web pages (e.g., HTML and XML) and the grouping optimization does not require semantics of the data content to be monitored. For example, WebCQ can monitor a change in a paragraph on a Web page without knowing the meaning of the paragraph.

Recently, there have been some research efforts on continuous query systems over data streams, including the Stream project in Stanford University [5], Aurora project in Brown University and MIT [46], and research projects in UC Berkeley [27, 28]. Most of these projects have explored optimization strategies such as grouping, multi-query optimization, and caching over data streams. The current WebCQ system provides grouping optimization and caching at Web page level (proxy cache) as well as sentinel object level. Continual queries over data streams is an ongoing research effort in the WebCQ development.

10 Conclusion

We have presented WebCQ, a prototype of a large-scale Web information monitoring system, with an emphasis on general issues in designing and engineering a large-scale information change monitoring system on the Web. Features of WebCQ include the capabilities for monitoring and tracking various types of changes to static and dynamic web page changes, personalized delivery of page change notifications, and personalized summarization of web page changes. One of the main design goals of WebCQ is the scalability of processing a large number of Web page sentinels. We develop a sentinel grouping strategy to allow sentinels of similar structure to be grouped together. Sentinels of the same group can be processed together at low cost by reducing the amount of repeated computation. Therefore, grouping leads to highly scalable processing of Web page sentinels.

Our work on the WebCQ system continues. Several issues are being explored, including experiments to evaluate the performance of alternative architectures and algorithms, and the performance improvements by incorporating research results in indexing techniques into the WebCQ change detection and change notification. We are also investigating the scalability enhancement of the WebCQ system along three dimensions: replication, distribution, and caching.

Acknowledgement

We would like to thank the entire CQ team, both past and current members, for the many interesting discussions and the development endeavor. The authors are partially supported by NSF CCR-9988452, DARPA PECS Program, DoE SciDAC program, and Yamacraw mission, State of Georgia.

References

[1] Webwhacker. http://www.webwhacker.com.

[2] NetMind. http://www.netmind.com.

[3] TracerLock. http://www.tracerlock.com.

[4] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[5] S. Babu and J. Widom. Continuous Queries over Data Streams. In *ACM SIGMOD Record*, September 2001.

[6] BotSpot. http://bots.internet.com.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *Proceedings of IEEE Infocom '99*, pages 126–134, March 1999.

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

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

[10] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2000.

[11] F. Douglis, T. Ball, Y.Chen, and E. Koutsofios. WebGuide: Querying and Navigating Changes in Web Repositories. *Proceedings of 1996 USENIX Technical Conference*, January 1996, pp1335-1344.

[12] F. Douglis, T. Ball, Y.Chen, and E. Koutsofios. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web, 1 (1), pp.27-44*, January 1998.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and e. T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol - HTTP/1.1. January 1997.

[14] 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.

[15] E. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of ACM SIGMOD Conference*, 1992.

[16] E. Hanson, C. Carnes, L. Huang, M. Konyala, and L. Noronha. Scalable trigger processing. In *Proceedings of the International Conference on Data Engineering*, 1999.

[17] Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[18] ICQ. http://www.icq.com.

[19] J. W. Hunt and M. D. Mcllroy. An algorithm for efficient file comparison. *Technical Report Computer Science TR#41, Bell Laboratories, Murray Hill, N.J.*, 1995.

[20] R. Ladin, B. Liskov, L.Shrira, and S. Ghemawat. Programming high availability using lazy replication. *ACM Transactions on Computer Systems, 10,4, pp360-391*, Nov. 1992.

[21] 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.

[22] L. Liu, C. Pu, and W. Han. XWrap: An XML-enabled Wrapper Construction System for Web Information Sources. *Proceedings of the International Conference on Data Engineering*, 2000.

[23] L. Liu, C. Pu, and W. Han. An XML-Enabled Data Extraction Tool for Web Sources. *International Journal of Information Systems, Special Issue on Data Extraction, Cleaning, and Reconciliation*, 2001.

[24] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering, Special Issue on Web Technology*, 1999.

[25] L. Liu, C. Pu, and W. Tang. WebCQ: Detecting and Delivering Information Changes on the Web. *Proceedings of the International Conference on Information and Knowledge Management*, November 2000.

[26] L. Liu, C. Pu, W. Tang, and W. Han. Conquer: A Continual Query System for Update Monitoring in the WWW. *International Journal of Computer Systems, Science, and Engineering. Special Issue on Web Semantics*, 1999.

[27] S. Madden and M. J. Franklin. Fjording the stream: an architecture for queries over streaming sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[28] S. R. Madden, M. A. Shaw, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[29] 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.

[30] Mortice Kern Systems (MKS). Web integrity. http://www.mks.com/solutions/ebms/.

[31] B. C. Neuman. Scale in Distributed Systems. *IEEE Computer Society Press*, 1994.

[32] M. Newbery. Kapipo. http://www.vuw.ac.nz/ newbery/katipo.html.

[33] Pointcast. http://www.pointcast.com.

[34] D. Rocco, D. Buttler, and L. Liu. Sdiff. *Technical Report, Georgia Tech, College of Computing*, February 2002.

[35] Ronald L. Rivest. Rfc 1321: The md5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[36] B. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer, Vol.23, pp.9-21*, May 1990.

[37] 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.

[38] SmartBookmarks. http://www.firstfloor.com/.

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

[40] Webclipping.com. http://www.webclipping.com.

[41] WebCQ (online demo). http://disl.cc.gatech.edu/WebCQ.

[42] WebSprite. http://www.websprite.com.

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

[44] WWWFtech. http://www.wwwftech.com.

[45] 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.

[46] S. Zdonik, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and D. Carney. Monitoring streams - a new class of data management applications In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, 2002.

A WebCQ related concepts

The following explains some basic concepts used in WebCQ object extraction.

1. **Document**

   A *Document* is referring to the Web page itself.

2. **Link**

   A *Static Web page* is a Web page that does not incur server-side computing. A static Web page does not change between subsequent page fetching requests unless the owner of the page manually modifies it.

   A *Dynamic Web page* is a Web page that involves server-side computing for content generation. A server-side program is invoked to create the content of the page whenever a page request is received at the Web server. Example of dynamic Web pages include JSP (Java Server Page), ASP (Active Server Page), SSI (Server-Side Includes) pages, and pages generated by CGI (Common Gateway Interface) programs or Java servlets.

   A *Link* object can be classified into the following categories:

   - Any URL[2] specified in an HTML tag that has "href" parameter, but not a URL of "mailto://" or "javascript://" types.
   - Any HTML form action, i.e., the value specified for the "action" parameter inside <FORM> tag.
   - Any URL that is the content for "value" parameter within any HTML tag, e.g., "<OPTION VALUE=http://www.cc.gatech.edu>". Some Web pages use this kind of links inside a drop-down selection menu.

3. **Image**

   An *Image* object is identified by the "<IMG>" HTML tag.

4. **Table**

   A *Table* data object is recognized by the "<TABLE>" and "< /TABLE>" HTML tags.

5. **List**

   A *List* data object is identified by "<UL>"-"< /UL>", "<OL>"-"< /OL>", or "<DL>"-"< /DL>" tags. *All Words* object represent all the "words" on the page.

6. **Word**

[2]Right now, we recognize these URL protocols: http, ftp, file, gopher, mailto, mid, afs, cid, news, nntp, prospero, telnet, rlogin, tn3270, and wais. See ftp://ftp.isi.edu/in-notes/rfc1738.txt for more details.

An ASCII character string[3] starting with alphabetic character ('a'-'z', 'A'-'Z') or a numerical value ('0'-'9') without any whitespace (horizontal tab - %x09, space - %x20, line feed - %x0A, carriage return - %x0D). For example, "WebCQ" is considered a word, as well as "2000-7-6,".

7. **Phrase**

   Phrase data object is defined as a set of words concatenated by zero or more white spaces, which does not contain HTML tags.

8. **Keyword**

   Keyword data object is consisted of one or multiple words which does not contain HTML tags.

9. **Regexp text segment**

   A *Regexp text segment* is a segment of HTML document source represented using a regular expression, e.g., "Daily stock quotes(.*?)IBM(.*?)Volume". For each extracted object (data object), we have an associated sentinel type for the detection of changes to the particular object.

[3]Right now, we do not consider extended ASCII.

B WebCQ Sentinel Change Detection Algorithm

```
Document $D_{new}$, $D_{cache}$;      // the current and cached Web pages
int $\epsilon$;                       // a threshold number (> 0), default 1
Case Sentinel Type of:
    "Any change":
            if (timestamp($D_{new}$) <> timestamp($D_{cache}$)) then {
                if (|sizeOf($D_{new}$) - sizeOf($D_{cache}$)| > $\epsilon$)
                    then **detected "document changed"**;
                    else if (MD5($D_{cache}$) <> MD5($D_{new}$) then **detected "document changed"**;
            }
            return;
    "All links"
            if ({links in $D_{new}$} MINUS {links in $D_{cache}$} <> $\phi$)
                then **detected "new links added"**;
            if ({links in $D_{cache}$} MINUS {links in $D_{new}$} <> $\phi$)
                then **detected "old links removed"**;
            return;
    "All images"
            if ({images in $D_{new}$} MINUS {images in $D_{cache}$} <> $\phi$)
                then **detected "new images added"**;
            if ({images in $D_{cache}$} MINUS {images in $D_{new}$} <> $\phi$)
                then **detected "old images removed"**;
            return;
    "Phrase": // for simplicity, we do not consider duplications of phrase match
            String p = Phrase_sentinel_content; //attribute of sentinel object
            String B = $Beginning_part_of_bounding_box; //initialized from cache
            String E = $Ending_part_of_bounding_box; //initialized from cache
            if (B $\in$ $D_{new}$ *and* E $\in$ $D_{new}$)
                then {// bounding boxes are not changed
                    //extract(D,B,E,ifTag) is a function to get the text region inside a document D given
                    //bounding box bounds B and E, boolean "ifTag" controls whether to include tags or not
                    String $O_p$ = *extract*($D_{new}$, B, E, true);
                    if ($O_p$ <> $O_{p_{cache}}$) then **detected "phrase change type (1) "**;
                } else { // bounding boxes are changed
                    //expand(D,p) is a function to get the text region in a document D if its "non-tag"
                    // representation is p
                    String $O'_p$ = expand($D_{new}$, p);
                    if (E $\overline{\in}$ $D_{new}$ && B $\overline{\in}$ $D_{new}$) { //both B and E changed
                        if ($O'_p$ <> $O_{p_{cache}}$) then
                            B' = adjust(B); E' = adjust(E);
                            if (extract($O'_p$,B',E',false) <> extract($O_{p_{cache}}$,B',E',false))
                                then **detected "phrase change type (7)"**;
                    } else if (E $\overline{\in}$ $D_{new}$) then { // only E changed
                        if ($O'_p$ <> $O_{p_{cache}}$) then
                            E' = adjust(E);
                            if (extract($O'_p$,$\phi$,$\phi$,false) <> extract($O_{p_{cache}}$,$\phi$,$\phi$,false))
                                then **detected "phrase change type (6)"**;
                    } else if (B $\overline{\in}$ $D_{new}$) { // only B changed
                        if ($O'_p$ <> $O_{p_{cache}}$) then
                            B' = adjust(B);
                            if (extract($O'_p$,$\phi$,$\phi$,false) <> extract($O_{p_{cache}}$,$\phi$,$\phi$,false))
                                then **detected "phrase change type (5)"**;
                    }
                }
            return;
```

Figure 17: WebCQ Sentinel Change Detection Algorithm

```
"Table":
        String t = Table_sentinel_content; //attribute of sentinel object
        String B = $Beginning_part_of_bounding_box; //initialized from cache
        String E = $Ending_part_of_bounding_box; //initialized from cache
        String $O_t$ = extract($D_{new}$, B, E, true);
        if ($O_t$ <> null and toString($table_{cache}$) <> $O_t$))
            then detected "table change";
            //* we can further detect detailed table cell changes
            else if ($O_t$ == null)
                then
                //* it could be bounding box change or table disappearance, we need to distinguish
        return;
"List":
        String l = List_sentinel_content; //attribute of sentinel object
        String B = $Beginning_part_of_bounding_box; //initialized from cache
        String E = $Ending_part_of_bounding_box; //initialized from cache
        String $O_l$ = extract($D_{new}$, B, E, true);
        if ($O_l$ <> null and toString($list_{cache}$) <> $O_l$))
            then detected "list change";
            // we can further detect detailed list item changes
            else if ($O_l$ == null)
                then
                //* it could be bounding box change or list disappearance, we need to distinguish
        return;
"All words"
        if ({words in $D_{new}$} MINUS {words in $D_{cache}$} <> $\phi$)
            then detected "new words added";
        if ({words in $D_{cache}$} MINUS {words in $D_{new}$} <> $\phi$)
            then detected "old words removed";
        return;
"Key words": // case-insensitive and boolean "AND" match
        $\kappa$ = {keyword list};
        flag = $F_{cache}$; // previous keyword match result
        match = false;
        foreach kw $\in$ $\kappa$ do {
            if ($\exists$ w $\in$ {All words} s.t. UPPERCASE(kw) is_substring_of UPPERCASE(w))
                then match = true;
                else { match = false; break; }
        if (flag <> match) then {
            if (flag == false)
                then detected "keyword appeared";
                else detected "keyword disappeared";
        }
        $F_{cache}$ = match;
        return;
"Regular Expression":
        String regtext = eval($regexp);
        if (regtext <> $regtext_{cache}$) then
            detected "content update to regexp segment";
        return;
```

Figure 18: WebCQ Sentinel Change Detection Algorithm (continued)