Managing Versions of Web Documents in a Transaction-time Web Server

Curtis Dyreson, Hui-Ling Lin, and Yingxia Wang
Washington State University
School of Electrical Engineering and Computer Science
Pullman, WA, USA
+1 509 335 0903
{cdyreson,ywang,hlin3}@eecs.wsu.edu

ABSTRACT

This paper presents a *transaction-time* HTTP server, called ττApache that supports *document versioning*. A document often consists of a main file formatted in HTML or XML and several included files such as images and stylesheets. A change to any of the files associated with a document creates a new version of that document. To construct a document version history, snapshots of the document's files are obtained over time. *Transaction times* are associated with each file version to record the version's lifetime. The transaction time is the system time of the edit that created the version. Accounting for transaction time is essential to supporting audit queries that delve into past document versions and differential queries that pinpoint differences between two versions. ττApache performs automatic versioning when a document is *read* thereby removing the burden of versioning from document authors. Since some versions may be created but never read, ττApache distinguishes between known and assumed versions of a document. ττApache has a simple query language to retrieve desired versions. A browser can request a specific version, or the entire history of a document. Queries can also rewrite links and references to point to current or past versions. Over time, the version history of a document continually grows. To free space, some versions can be *vacuumed*. Vacuuming a version however changes the semantics of requests for that version. This paper presents several policies for vacuuming versions and strategies for accounting for vacuumed versions in queries.

Categories and Subject Descriptors

H.3.5 [Online Information Services]: Web-based services—*HTTP server;* H.2.4 [Database Management Miscellaneous] Temporal—*Document versioning, transaction-time databases*

General Terms

Design, Performance.

Keywords

Observant system, versioning, transaction time

1. INTRODUCTION

The World-wide Web is the largest, most frequently used, text-based information resource. The web currently has several million servers providing access to several billion documents. The web is also dynamically changing. Hundreds of thousands of documents are added, moved, updated, and deleted daily. Recent studies have investigated the lifetime of documents [2,4,19]. Cho and Garcia-Molina suggest that documents at `.com` sites change rapidly and have short lifetimes, while those in `.edu` and `.gov` domains change slowly and live longer. In the study, more than 40% of documents in the `.com` domain changed every day, but more than 50% of documents in `.edu` and `.gov` domains were unchanged for at least four months. Many of the documents available on the web conform to the HyperText Markup Language (HTML), but in the near future, the Extensible Markup Language (XML) [27] is expected to gain in importance as a mark-up language for web documents.

The database research community has been active in applying database concepts and techniques to the web [10]. Over the past two decades there has been a substantial amount of research on extending databases to support time [23,25]. This research has, in part, developed the field of *transaction-time databases* [15,17,21]. Transaction time is the time when a particular fact is stored in a database and considered current, i.e., the time between when it is inserted and deleted (an update is modeled as a deletion followed by an insertion). Very briefly, a transaction-time database stores all of the past states of a database and allows queries, called transaction timeslice, to retrieve any desired past state. Transaction-time databases are useful when the history of a database is needed, for instance when performing audits in financial or legal applications. They also support unlimited undo or rollback on (committed) data.

In this paper we apply concepts and strategies from transaction-time databases to the web. Transaction time is a problematic concept for the web because *there are few update transactions*. Browsers and other consumers of web data have read access to data, but rarely can insert, update, or delete data. Updates to web data are irregular, ad-hoc, and hidden from readers of that data. To remain current with a constantly evolving document, the document must be re-read.

We call such a reader an *observant system*. An observant system is a system that can observe documents but (generally) cannot modify them. A web browser is an observant system. It reads documents from the web but cannot update those documents (however, a browser can submit information to a server for update). A web server (an HTTP server) is also an observant system. A web server responds to an HTTP GET by reading a file from local storage, but it is usually uninvolved in an update of that file. Observant systems are common on the web because they facilitate the fast, easy, and cheap publication of data: data is published by placing it at a location where it can be observed.

Although an observant system is uninvolved in an update, it can detect that an update has occurred during a read by comparing the current version with the last observed version of the same

WWW 2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

document. A difference denotes that an update occurred sometime since the previous read.

A *transaction-time web server* is an observant system that archives document versions during HTTP requests to create a complete history of the documents at a website [8]. The server also processes *transaction-time queries* to fetch requested versions from the archive. Internal to the server are an *archive* to store past versions and a *history table* to record information about the versions.

This paper describes an extension of the Apache web server, called ττApache (Transaction-Time Apache), that provides transaction-time support. ττApache makes the following contributions.

No additional work for document authors — The extended server automatically archives document versions during HTTP GETs. Unlike other archives that store versions off-line or using robots, we believe that the server itself can efficiently create and manage versions during resource GETs.

HTTP-compatible queries — A simple "URL munging" scheme to retrieve versions and version histories [8]. The scheme is very inelegant, but is fully backwards-compatible with existing servers, browsers, and standards. An Apache server can seamlessly migrate to a ττApache server at any time without affecting anything else on the web. Elsewhere we have proposed better ways to express transaction-time queries, in for example XPath [9], but more elegant solutions require making more substantive changes [16].

Link rewriting — Server-side rewrites of links and references to other documents to point to archived documents [8]. This enables a user to "time-travel." In time-travel the user defines their perspective as of some past (or future) time and surfs the web (of ττApache servers) as it existed at that time.

Assumed versioning — Support for *known* vs. *assumed* versions. Some versions of the document may be unobserved versions because they were not requested from the server during their lifetime. If the document has not been modified since the last read, then it is *known* that the current version is under observation. But if a document has been modified since the last read, then the evolution of the document is unknown between the read time of the previous observed version and the modification time of the current observed version. One or more unobserved, transitory versions may have existed. Hence the previously read version is *assumed* but not known to have existed until the current version's modification time.

Vacuuming — The ability to expire documents from the archive and maintain version history for files that are moved or change names.

Efficient performance — We give empirical measurements of the extended functionality. We show that the server extensions increase disk I/O in some HTTP sessions, which results in a slightly slower average turnaround time. But the additional cost would not adversely impact the performance of most servers.

All of the new functionalities are designed to be backwards compatible with existing protocols (e.g., HTTP) and standards (e.g., HTML), so a site can become a vacuum-enhanced, transaction-time web server at any time. This paper presents a logical model for the design and URI-compatible syntax for supporting the new functionality.

1.1 Related Work

In many situations "old" documents are still of use. Currently, the de facto method for storing old documents is an *archive*. An archive is a warehouse for deleted or modified documents. When a document[1] is modified, it is moved either manually or automatically (often by a robot) into the archive. Each archive has a specific interface to find an archived document, usually in a few mouse clicks. At many sites, especially news-related sites, a search engine-like retrieval mechanism is also available. Archives can be site-specific or built for a number of sites, e.g., the Internet Archive [12]. Unlike the Internet Archive, ττApache archives only the documents that it serves.

One problem with some archives is that the retrieval interface is *not standardized* but instead varies widely from site to site. This is problematic because when an old document is retrieved, the (external) links on that document point to current information. Furthermore, it is often the case that an archived document cannot be displayed the same as when it was created because it includes files such as inline images and links to external documents that have subsequently been archived.

The Internet Archive uses the WayBack Machine to elegantly, and correctly support transaction timeslice. When an archived page is retrieved a JavaScript program is appended to the page to redirect hyperlinks on the page to archived documents in the Internet Archive as of the time the page existed. This allows the user to surf the web, as it once existed, or at least the portion of the web that is stored in the Internet Archive as of that past time.

iPROXY is a closely related system [20]. iPROXY is a personal proxy server. One of the services it provides is archiving of the documents it downloads. Hence a client can set up an iPROXY server to create a *proxy-side* archive. The documents in the proxy-side archive may originate at many different servers. ττApache, in contrast, maintains a *server-side* archive. It archives only documents that it serves. One advantage of a server-side archive is that all the clients share it. This does not preclude documents in a server-side archive from being additionally cached in proxy- or client-side archives. Like ττApache, iPROXY uses URL-munging to support transaction timeslice, but without a link-rewriting component.

ττApache provides much finer-grained versioning than iPROXY or the Internet Archive. ττApache is a *per-request* archiver. In contrast, the Internet Archive is a periodic archiver. The Internet Archive robot only periodically visits a document. iPROXY is an *on-demand* archiver. In on-demand archiving a document is archived as the result of a specific user request. There are other systems that do *author-requested* archiving, for instance with a cgi-bin script [7].

Neither the Internet Archive nor iPROXY create document version histories. So neither supports *next* or *previous* version queries nor distinguishes between known and assumed versions. ττApache is also the only archiver to support vacuuming, which allows users to control the growth of the archive.

In the context of transaction-time database, a semantic foundation for vacuuming has been presented [22]. A vacuuming specification is proposed that consists of a *removal specification part* and a *keep specification part* that overrides the removal part. Vacuuming impacts both database queries and updates. ττApache

[1] We will use the terms 'page', 'resource', and 'document' interchangeably in this paper.

supports only a removal specification, and correctly supports vacuuming for both web queries (HTTP requests) and updates (document edits).

Concurrent Versions System (CVS) is a widely-used version control system for developers to maintain their source code [6]. CVS stores the version history of files in a *repository*, which is built as a directory tree structure corresponding to the directories in a *working directory* outside of the repository. The history of each file keeps all versions of that file in the RCS file format that only stores difference between versions [24]. ττApache stores entire versions rather than the difference between versions. Versions are committed to CVS via explicit command line options. Besides version tracking, CVS provides functionalities such as browsing histories, removing and renaming files and directories.

The final related system is Xyleme [26]. Xyleme is a warehouse for XML data. XML documents are periodically pulled from the web and incorporated into the warehouse. Version information, or rather, differences between versions of a document are detected, stored, and can be queried. Efficient techniques for isolating changes between versions have been developed [5]. Unlike Xyleme, ττApache is a very primitive versioner. ττApache does not compute changes between versions since the versioning is done in the inner-loop of the server, potentially on each request. Hence we need to keep the cost of versioning at a minimum. In this paper we empirically demonstrate that our extensions to Apache have little impact on server performance for real-world conditions.

1.2 Motivating Functionality

A transaction-time web server provides HTTP-compatible queries allowing online users to time-travel among different versions of resource. ττApache supports version *time-slice* and version *history* queries. A timeslice query retrieves the version of a document as of a given time. A *previous* (*next*) version query retrieves the requested version relative to the current version (what is considered "current" depends on previous timeslice queries). A *history* query returns the list of versions in a history.

We implemented a simple scheme for specifying such queries: they are appended after a '?' to a URL. The advantage of this scheme is that requests to non-transaction-time servers will function exactly as before since the query portion is ignored in HTTP requests for static resources (queries for dynamic resources are not-included in this strategy). We chose to use "URL munging" because it requires no changes to existing browsers, servers, or HTTP. A site can choose to use a ττApache web server without adversely impacting current functionality.[2]

The following examples illustrate the strategy.

1) Retrieve the current version of `sports.html`.

```
sports.html?now
```

One could also specify that the links in the retrieved version be rewritten to point to the current version. A comma character separates the time-slice specification from the link-rewriting specification. In the following query, both specifications are "`now`".

```
sports.html?now,now
```

However, since the default fetch and rewrite are `now` the following URL has the same effect, as both of the URLs given above.

```
sports.html
```

2) Resurrect the previous version of `sports.html` as though it were the current version.

```
sports.html?pre
```

A non-transaction-time web server ignores the transaction time part of the URL (for a static document) so it would fetch the current document. A transaction-time web server will fetch the predecessor, but will not restructure links in the predecessor. The version two versions ago can also be requested by appending a second `pre` step to a `pre` time-slice as illustrated below (a period character is used append a step).

```
sports.html?pre.pre
```

As can the version prior to 26-Sep-2003.

```
sports.html?26-Sep-2003.pre
```

3) Retrieve the version as of `26-Sep-2003`, and time-travel on links as of that time.

```
sports.html?26-Sep-2003,26-Sep-2003
```

The links are rewritten to time-travel within the selected timeslice. If the version on the requested date is assumed, a 404 error is generated.[3] In previous and next version queries the link rewriting time can be set to the time of the selected version.

```
sports.html?next.next,timeOf
```

4) Retrieve a list of the changes made in 2003 to `sports.html`.

```
sports.html?history(1-Jan-2003,
                    31-Dec-2003)
```

The time interval of the history query is set to the entire year. The query will return a page with a list of links to known and assumed versions (the formatting of the page is part of the server configuration).

5) By default only the *known* versions of the document are used. The *`assumed`* keyword can be added to include the assumed versions in the fetch (or rewrite) (Section 2.2 discusses the differences between known and assumed versions). Fetch the version prior to `26-Sep-2003`, even if that date specifies an assumed version.

```
sports.html?assumed.26-Sep-2003.pre
```

6) Remove all the versions within the first year of the server's history (i.e., vacuum with a fixed time-window [`begin` to `begin+365`] where `begin` is the start time).

```
sports.html?
    vacuum(t-window,begin,begin+365)
```

The vacuuming works only on the file `sports.html`. The archived versions of *included files* (e.g., images) are not affected. A vacuuming specification can be set only by an authorized user.

[2] URL munging is an inelegant solution, but can be implemented in existing browsers, servers, and relevant W3C recommendations (namely it fits within the HTTP protocol and URI scheme). In future, we anticipate that web technology will permit cleaner expression of transaction-time queries using content negotiation, XLink, or XPointer.

[3] The default strategy can be modified in the server's configuration file to use assumed versions or to return the latest known version prior to an assumed version.

7) Vacuum every other version of all files in the "`sports`" directory. This specification uses a version window, which starts at version 1, and applies to every 2nd version.

```
sports/?vacuum(v-window,1,2)
```

8) A query for the seventh version of sports.html will result in a 404 HTTP_NOT_FOUND error due to the previous vacuuming specification. The web master prefers a more elegant interaction with users of the site. She would like the server to *repair the query* and return the previous document version as a reply for the user's request for a vacuumed version. She can achieve this as follows.

```
sports.html?vacuum(repair=past)
```

9) If the history of the document is no longer needed, it can be obliterated, removing it from the archive.

```
sports.html?obliterate
```

1.3 Paper outline

This paper describes an efficient archive with a standard, simple mechanism for retrieving past versions of documents. Apache archives documents during resource *reads* as described further in Section 2. Section 3 gives a syntax for transaction-time queries and describes the functionality of the server. We implemented the archive as an extension of the Apache web server, as discussed in Section 4. Creating document versions is potentially expensive since it is done in the "inner loop" of the server, and adds processing to *every* request. We empirically measure the cost of the extra processing in Section 5.

2. TIME MODEL

An observant system only occasionally observes a document. Each observation yields information about the document, as it exists at a single point in time, which we will call the *read time*. The observation also yields meta-data about the document. In the HTTP 1.1 protocol an important piece of meta-data is the *modification time* of a file. The modification time is the time when the file was last modified.

The read and modification times are kinds of transaction time. Research in temporal databases has identified two primary, distinct time dimensions: *valid time* and *transaction time* [13]. Valid time is the real-world time of a datum, whereas transaction time is the system time when that datum exists on the system. A file modification time is a transaction time. In this paper, the transaction-time domain is a set of instants,

T = {`beginning`, …, *now*, …, `forever`}.

The variable, *now*, represents the ever-changing current time [3]. In contrast to traditional temporal database research, the transaction-time domain ends at forever thus permitting future transaction times. This enables document authors to set future expiration times for documents and to schedule documents for future release. Example times in this paper will be represented using Gregorian calendar conventions in the granularity of days, so each instant in the transaction-time domain corresponds to a day.

2.1 File versions

We will call each observed modification a new *version* of the file. The read time and modification time can be used to infer knowledge about unobserved states of the file. If the file has not been modified since the last read, then it is known that the current observed version is the version in existence since the modification time. We will refer to this as the *known* lifetime or state of a version. Each subsequent read on an unmodified version will push the read time later. If the document has been modified since the last read, then the evolution of the file is unknown between the read time of the previous observed version and the modification time of the current observed version. We will refer to the unobserved lifetime of a version as its *assumed* lifetime, because the observant system can only assume that the version existed. A file version history is defined as follows.

Definition 1. [File version history] The version history of a file, denoted $V(f)$, is represented as a sequence of ordered triples, $(m_1, r_1, s_1), \ldots, (m_n, r_n, s_n)$, where m_i is the time of the file modification that created version i, r_i is the time at which that file was last observed, and s_i is the HTTP status (OK, missing, or protected). We assume that the version must have been observed at least once, so $m_i \le r_i$ for every version i. ■

A file version is created when a file is first read at time t. The file's modification time m_1 is retrieved and the first version's information is (m_1, t, s_1). Each subsequent read, at time r_1, increases the last read time to (m_1, r_1, s_1). Finally, during some read, at time r_2, it is observed that the file has been modified at time m_2. A new version is created with the information (m_2, r_2, s_2). The closed interval of time $[m_1, r_1]$ is the known lifetime of version 1 and the open interval of time (r_1, m_2) is the assumed lifetime.

An observant system should include the history of observed HTTP errors. Hence we record when a version is missing (a 404 error) or protected from observation (a 403 error).

2.2 Document versions

A document version history is derived from a set of file version histories. A document consists of a *main* file and a set of included files such as figures, inline image and stylesheets, which we will call the *included files*. Over time, the main file may change or any of the included files. Furthermore, the members in the set of included files may vary. A new document version is created by a change in either the main file or any of the included files.

Definition 2. [Document version history] The version history of a web document is represented as a sequence of ordered tuples, $(k_1, e_1, a_1, F_1, s_1), \ldots, (k_n, e_n, a_n, F_n, s_n)$ where $[k_i, e_i)$ is the known lifetime of version i and $[e_i, a_i)$ is the assumed lifetime. The version must have some lifetime so either $k_i < e_i$ or $e_i < a_i$. If $e_i = a_i$ then the version has only a known lifetime, or if $k_i = e_i$ then the version has only an assumed lifetime. F_i is the set of included files for version i, and s_i is the HTTP status of the main file. ■

We first give an example of constructing a document version history, and then discuss the details. The known and assumed lifetimes for a document version are derived from the times of the file versions. Figure 1 shows the document version history of a `sports page` document. The document consists of a main file, `sports.html`, and three included images, `a.gif`, `b.gif`, and `c.gif`. The lifetime of each file version starts with the time

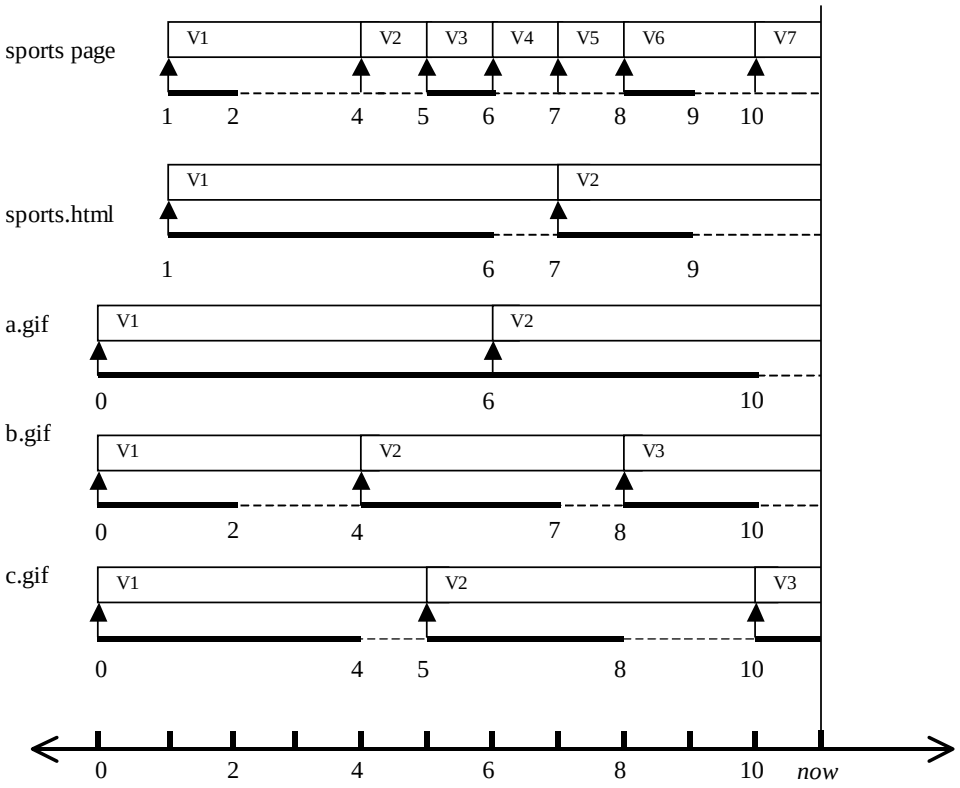

Figure 1 Constructing a Document Version History

that the version was created (a file modification time, as indicated by an arrow). The lifetime extends until the version was next modified. During its lifetime the version was possibly read several times. The time of the last read terminates the known time of that version (indicated by the solid line below the version). The assumed version of the file begins after the last read since the file was unobserved from that time until it was next modified (indicated by the dashed line below the version). For instance, the file `sports.html` has versions: V1 and V2. V1 was created at time 1 and last read at time 6. At time 9 the file was read again. An edit had taken place at 7, so V1's lifetime was terminated and V2's lifetime begun. Between 6 and 7, V1 was not observed and so its existence can only be assumed. It may have been deleted and recreated or edited several times. The document versions are derived from the underlying file versions. The overall lifetime and status of the document is the same as that of the main file. Each file version creates a new sports page version. The sports page version has a known lifetime when all of the file version's lifetimes are known but an assumed lifetime when even one file version is assumed.

The remainder of this section describes the construction process in detail.

Definition 3. [Document version] Assume that a document, D has a main file, M. For each version of M, let m_i be the modification time of the version, s_i be the status, and F_i be the set of included files. Then the set of document modification times is given below.

$$T_{mod} = \{m \mid \exists i[(m_i, r_i, s_i) \in V(M) \text{ and } (m,r,s) \in V(f) \text{ and } m_i \le m < m_{i+1} \text{ and } f \in F_i] \text{ or } (m, r, s) \in V(M)\}$$

T_{mod} is the set of modification times for the main file together with the set of modification times for included file versions that have the same lifetime as some main file version. Let

$$Sort(T_{mod}) = (m_1, m_2, \ldots, m_j)$$

be a sorted list of modification times such that $m_i < m_{i+1}$. A version $v_i = (k_i, e_i, a_i, F_i, s_i)$ of D is constructed as follows.

- $k_i = m_i$
- $a_i = m_{i+1}$
- s_i is the status of M as of time k_i
- F_i is the set of included files in M as of time k_i
- $e_i = r$ where r earliest time, $m_i \le e_i \le m_{i+1}$, for which some file version (included or main) is assumed. ■

3. FUNCTIONALITY

This section discusses, from a user perspective, the functionality that ΓΓApache adds to the Apache web server. First, a transaction-time server supports additional kinds of queries, called *transaction-time queries*. There are transaction-time queries to retrieve specific versions as well as entire version histories. Over time the collection of archived versions will continue to grow. But versions can be vacuumed from the archive to recover space. We present several vacuuming strategies. It is also possible to hide documents from ΓΓApache, and forward version histories when documents are forwarded.

3.1 Queries

In this section we present a syntax for transaction-time queries (to conserve space we have omitted the syntax of vacuuming specifications). Transaction-time queries consist of two parts: a *fetch* and *rewrite*. The fetch part retrieves a particular version or version history. The rewrite part specifies the formatting of hyperlinks in the requested document. The BNF for a query is given in Table 1. Table 2 lists the meanings of the keywords in the BNF. The default for an empty fetch and rewrite is *now*. Therefore, the default response of a transaction-time web server is the same as that of a normal web server, i.e., return the current version and do not rewrite links. The queries use only the known versions by default, but the keyword `assumed` means that a user wants to include the assumed versions as well.

Table 1 BNF for queries

```
<ττQuery>   ::= <fetch> [,<rewrite>]
<fetch>     ::= <timeslice> | <history>
<timeslice> ::= ε
                | [assumed.]
                  (
                    pre
                    | next
                    |<time> [.<timeslice>]
                  )
<history>   ::= [<timeslice>.]
                history [(<time>,<time>)]
<rewrite>   ::= ε
                | [assumed.]
                  (
                    < time>
                    | pre
                    | next
                    | timeOf [.<rewrite>]
                  )
<time>      ::= <day>-<month>-<year>
                  [/<hour>:<min>:<second>]
                | now | beginning | forever
<day>       ::= 01 | 02 | … | 31
<month>     ::= Jan | Feb | … | Dec
<year>      ::= 0000 | 0001 | … | 9999
<hour>      ::= 00 | 01 | … | 23
<minute>    ::= 00 | 01 | … | 59
<second>    ::= 00 | 01 | … | 59
```

Table 2 Meaning of terms in a timeslice query

Term	Description
ε (empty)	Request for the current version
`pre`	Request for the previous version
`next`	Request for the next version
`now`	Request for the current version
time literal	Request version that existed at the specified time
`timeOf`	Restructure hyperlinks with the time which is the same as that in the transaction-time query
`assumed`	Use known and assumed versions.

3.2 Vacuuming

A primary concern in running and maintaining ττApache is that the archive grows in size over time until it eventually exceeds the storage capacity, since every modification of a main file or an included file is stored as a new file version in the archive. To restrict the archive's growth, we can store the differences between versions (thereby reducing the size of individual versions) or eliminate versions. There are three primary options. One is to store the difference between file versions with the difference computed by comparing each file version with its predecessor version. The second also stores the version differences, but the difference is computed by comparing each file version with the original file version. The third method is to eliminate versions, with the archive storing the complete file versions.

A comparison of the three methods is shown in Table 3. In the table, method (a) stores only the difference between successive versions; therefore it makes the best usage of storage. Method (c) stores entire file versions, thus occupying the most storage. Method (b) is in between in the storage consumption. It stores the difference between the version and the initial version (along with a complete initial version). When processing a time-slice query, method (c) can retrieve the queried version fastest; method (b) would have to reconstruct the requested version from the initial file version and the difference with the requested version; method (a) has to reconstruct the queried version by accumulating all the differences from the beginning, thus more time is needed. The reconstruction in method (a) needs every file version so a file version cannot be eliminated directly, however, a version can be eliminated indirectly after first determining the difference from the previous version to the next version (skipping the current version). While in method (b) and (c), every stored difference or file version is independent (or at most dependent on the initial version), hence most of the versions can be eliminated. So method (a) is not amenable to vacuuming. ττApache implements storage method (c), which provides faster query responsiveness than (b).

Table 3 A comparison of three storage control methods

Measured Aspect	Store Version Difference (Reference Chosen)		Store Entire Version (c)
	Previous Version (a)	Initial Version (b)	
Storage utilization	Smallest	Medium	Largest
Time-slice efficiency	Slowest	Medium	Fastest
Can eliminate?	Indirect	Yes	Yes

The archive keeps file versions for all documents at the website, but each document author should be able to express their preference for which file versions are to be vacuumed. For instance, one author may want to remove versions older than two years, while another would like to keep only versions that are significantly different. Hence, each deletion should be predefined under some *policy*. File versions are deleted according to *vacuuming policies* specified by document authors or a site administrator.

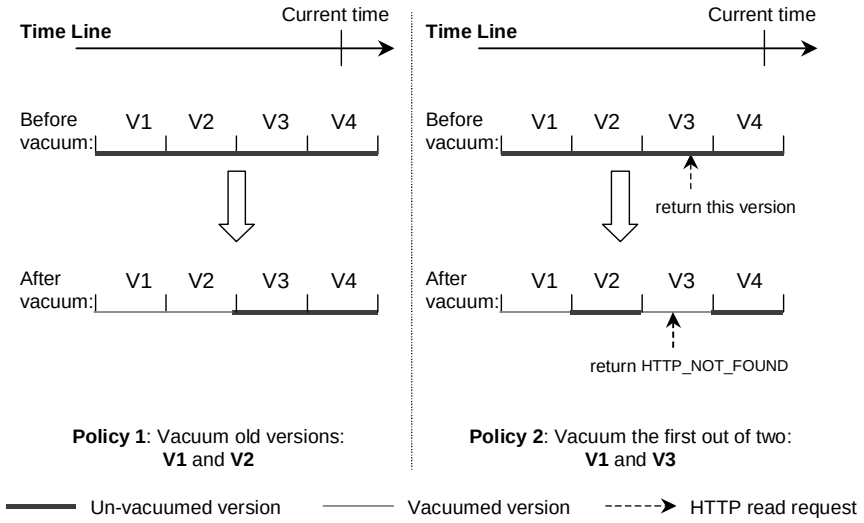

Figure 2 Vacuuming Example

Vacuuming, in a transaction-time database, means to physically delete records of past states [13]. A transaction-time database maintains past database states, thus making it possible to access any past state. If no information is physically deleted, a transaction-time database will grow forever, and will eventually outgrow the storage capacity. Vacuuming is a way to remove unwanted data when more space is needed for other data. The TSQL2 temporal query language offers a basic vacuuming functionality: when a particular date is specified, only data that is prior to the date should be physically deleted [14].

In a transaction-time web server that archives document versions, vacuuming works to restrict the growth of the archive. Compared with the data stored in a transaction-time database, the archive size increases much faster since web documents are much larger than database records. Each modified or deleted document, e.g. HTML file or image file, is stored in the archive. In general, a web server has a large number of web documents. Vacuuming is applied to reduce the size of the fast-growing archive. Old or seldom needed document versions or incremental document versions having small changes are vacuumed under specified vacuuming policies.

For example, suppose the file `sports.html` has four file versions in the archive: V1, V2, V3, and V4. If we apply the policy that older versions are no longer needed (e.g., V1 and V2 are "older" versions), then they are vacuumed from the archive. Policy 1 in the left half of Figure 2 shows the older versions being vacuumed. However, Policy 2 in the right half of Figure 2 shows the policy of vacuuming every other version; therefore, V1 and V3 will be vacuumed. A query for V3 will result in an HTTP_NOT_FOUND error (or the query can be repaired to the next or previous version).

Vacuuming policies are specified in the server configuration file on a per-directory basis, or can be dynamically specified by authorized users.

- *periodic* sieve - This policy vacuums every n^{th} version, e.g., a 1,2-sieve will vacuum every other version.
- *version-window* sieve – This policy vacuums a moving window of versions, e.g., vacuum all versions older than the fifth previous version, in other words, keep only the most recent five versions.
- *time-window* sieve – This policy vacuums versions within a fixed time window (e.g., Jan-1-2001 – Dec-31-2001) or a moving time window (e.g., now – 1 year).
- *percent-difference* sieve - This policy vacuums every version that is less than *n*% different from the previous version. The idea is to keep only versions that have "significantly" changed. The diff utility is used to compute the size of the difference.
- *composite* sieve – The previous sieves can be combined, e.g., in the previous year (a time-window sieve) vacuum every other version (a periodic sieve).

No matter which vacuuming policy is used, notification should be given to a user who queries a vacuumed version. The query-repair strategy is specified along with a vacuuming strategy on a per-directory basis.

- Redirect to previous version – A timeslice on a vacuumed version is redirected to the closest, previous, non-vacuumed version.
- Redirect to next version – Similar to previous, but the query is redirected to the closest, next, non-vacuumed version.
- Return vacuumed icon – An appropriate vacuum "icon" is returned by the server. For example, an request for a vacuumed HTML page could return a page reporting that the "Page has been vacuumed" along with a link to the previous and next non-vacuumed versions.

3.3 Additional functionality

Not all documents should be versioned in the archive. The server configuration includes options to specify the default policy for the entire site (archive everything or archive nothing). Individual directory trees and documents can be excluded (or included) in the archiving by editing the `transactionTime.cnf` file appropriately. Furthermore, an archived document (or directory tree) can be permanently removed from the archive by executing an *obliteration* specification.

Files are sometimes moved on disk to new locations. When a file is moved, the history of that file, unfortunately, does not move with it, since the file move is performed by the operating system,

not the web server. ΓΓApache has a forwarding query to let authors advise ΓΓApache that a file has been moved and the history of that file should be logically extended to the new location.

4. SERVER ARCHITECTURE

Figure 3 depicts a transaction-time server architecture. It works in much the same way as a normal server, but has some extended functionality to archive files during an HTTP GET and to respond to ΓΓQueries.

A client requests a document by passing a URL to ΓΓApache. Just like a normal server, the URL is converted to an absolute path to a file located somewhere on the server's file system, and the requested file is read from disk and sent to the client. Unlike a normal server, ΓΓApache maintains the file version history in the *history table*. The history table is a database of file modification and read time information. There are three possible actions when a file is read.

1) No history exists - A new tuple is created to store the information relevant to the first version.
2) The history is out-of-date - A new version has been created since the previous GET. If the latest tuple in the history is earlier than the modification time of the file, then the current version's lifetime is terminated and a new tuple for the new version is added to the history table. ΓΓApache also creates an archived version of the file as shown as gray block arrow in Figure 3.
3) The history is current - If the latest tuple stores the same file modification time as the current file, then no changes have been made since the previous GET however, the read time in the tuple is updated to the current time.

To process a transaction-time query, the history table is also consulted. First, ΓΓApache retrieves the requested file from the disk. It has to parse the file to determine which other files are part of the document (i.e., image files and stylesheet files). Currently it parses only HTML files. Once it determines the set of files for the document, the server then retrieves all of the tuples relating to all of the files and constructs the document version history as outlined in Section 2. Next the query itself is evaluated with respect to the created document version history. The requested version is retrieved from the archive and links are rewritten as specified by the rewrite part of the transaction-time query. Finally, the server sends the requested document to the client.

4.1 Apache

We implemented a transaction-time web server as an extension of the Apache web server. The Apache web server was chosen because it is widely-used, open source, free, and extensible. According to Netcraft, a British market research company, Apache servers now run on over 50 percent of the Internet's web server [18]. Our new features: the document history table and ΓΓQuery parsing and evaluation are implemented in the server's inner loop, which is the code to process incoming requests. The Apache web server uses a pre-forking model. It forks child processes to do the actual work of accepting incoming requests. A child process serves multiple connections in its lifetime. The parent process forks a new child process or kills an old one based on the load of the server. The advantage of this pre-forking model is robustness. If a child crashes the server can keep going.

4.2 Using BerkeleyDB for the history table

We chose BerkeleyDB for management of the history table. BerkeleyDB is an open source, database library that provides a simple function-call API for data access and management. BerkeleyDB is robust and fast. Since multiple processes are accessing the database (requests are handled in Apache by child processes) we implemented all database operations within transactions. We used BerkeleyDB's lock manager to support transaction aborts, commits, and concurrency control. BerkeleyDB uses deadlock detection and recovery in the lock manager rather than deadlock avoidance so we additionally checked all transactions to recover from potential deadlocks.

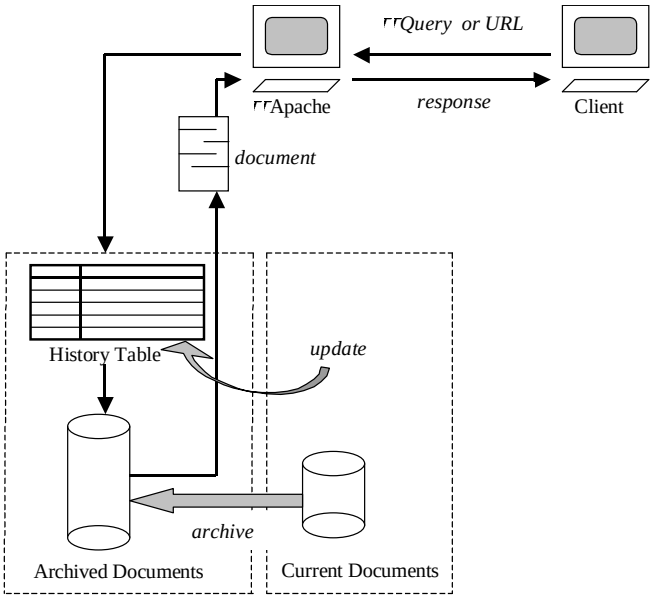

Figure 3 A ΓΓServer Model

5. EMPIRICAL ANALYSIS

ΓΓApache is implemented as an extended Apache web server. The extension supports user requests for specific document versions and histories. This section describes how the additional functionality impacts performance. Performance is an important concern for most web users [11]. There is always an overhead on performance when additional functionality is added to an application. In this section we empirically measure the overhead in a series of experiments. We discuss the goals and designs of the experiments in Section 5.1 and Section 5.2, respectively. The results are presented in Section 5.3.

5.1 Factors in measuring overhead

The goal of the experiments is to determine the amount of overhead imposed by the transaction-time functionality. The overhead could manifest itself in several, related ways. First, a longer turnaround time is expected. The turnaround time is the interval of time, at the client side, between the submission of an HTTP request and its completion. Second, disk I/O will be more frequent. ΓΓApache has to additionally manage the history table and to create archival copies of files. The increased frequency of disk I/O will result in a longer turnaround time. Third, memory usage might increase. Memory is consumed by the file system cache, virtual memory, the kernel, and server processes. Since

rrApache reads and writes more files than the normal Apache server, the file system cache will be more heavily used. As memory usage increases, swapping of virtual memory will also increase. So, the amount of memory used will (indirectly) effect the turnaround time. Fourth, CPU efficiency could decrease. If a process is waiting for disk I/O, the CPU will be idling for that process. A low CPU efficiency is an indicator of longer turnaround time.

One factor that influences the amount of overhead is the *rate at which files are updated*. On a request to a file that has been updated, rrApache has to update the history table and archive a file. This effectively doubles or triples the disk I/O compared with the normal Apache web server. Hence it will be important to test a range of file update rates to determine how the update rate changes the overhead.

The *size of a requested file* is also an important factor in determining the amount of overhead because a larger file takes more time to backup. In addition, the size of a main resource file is related to the performance of creating a version history since it is necessary to parse a file to figure out how many inline images are included before generating a list of all possible versions (a cost that is similar to allowing sever-side includes). We will experiment with different file sizes to determine how the file size impacts the cost.

The final factor is the *percentage of rrQueries* in all requests. We anticipate that transaction-time queries will have a greater overhead because several database reads may occur during a request. We will test several ratios of rrQueries to normal requests.

5.2 Design of the experiments

We designed a series of experiments to independently test how the page update rate, the page size, and the percentage of rrQueries affect the overhead. The experiments are "peak load tests" that artificially maximize the load on the server by inundating it with requests. This strategy is the same as that used by the Apache Benchmark tool [1]. We spawned five concurrent users to make requests. However, in contrast to other tools, we tried to force requests to go to the disk to read files rather than to find files in the system's file cache. If all the files are cached in memory, rrApache will be almost as fast as Apache. So we want to force rrApache to read and write files from disk and minimize the effects of file caching by the system. The system we are using to run the tests has a single disk with a random block I/O time of approximately 10 milliseconds, consequently, if all of our requests force at least one disk read, at best we can process about 100 requests per second.

Overall, every test for a single data point within an experiment consists of the following steps.

1. **Start rrApache from an initially empty state –** We create a new, empty database and a new, empty archive to ensure that all tests start in an identical, empty state.
2. **Pre-fetch 3000 documents -** Before applying the actual tests, we pre-fetch some of the available documents (there are 3000 documents at the test site). The purpose of this step is to populate the file system cache thereby reaching the memory condition of a normally active web server before starting the first run.
3. **Perform multiple runs -** We perform five runs for each data point. Each run randomly makes 3000 requests. The data collection methods vary. For timings, we record the turnaround time from the beginning of a run to the end of that run, and then take the average over all the runs. For disk and memory usage statistics, we aggregate the measurements, collected using `vmstat`, over all the runs.
4. **Shut down rrApache -** We shut down rrApache when the last request of the last run is completed. rrApache will be restarted (Step1) for a new test.

The general experiment is tailored to test three specific factors, file update rate, file size, and percentage of rrQueries. One factor is the rate at which files are updated. An update is implemented by "touching" a file just prior to requesting it. We decided to test file update rates of 0%, 1%, 2%, 5%, 7%, 10%, 15%, 50% and 100%. The percentage is the probability that a page will be updated. For instance, in a 2% test, there is a .02 probability that the test code will update the requested file. We measured the overhead for each update rate using the experimental procedure outlined above. Another factor is the file size. We performed experiments with large files, 60KBs, and small ones, 1KB. A third factor is the percentage of rrQueries. We decided to test four percentages of rrQueries, 1%, 5%, 20% and 80%. For example, a test with 1% rrQueries means that there are 99% normal Apache requests (non-transaction-time queries) and 1% rrQueries. The rrQuery applied in a test covers four kinds, `pre`, `26-Sep-2003`, `pre.pre`, and `history`. The rrQuery was picked randomly from these four up to the percentage of rrQueries.

5.3 Experiment Environment

We conducted the experiments on a Pentium PC (Dell Precision 340). It has an Intel® Pentium® 4 CPU 1700MHz, 256MB RAM and 37.2GB disk space. The PC runs Linux RedHat 7.2. We installed Apache v1.3.19 and BerkeleyDB v4.0.14 for testing. We isolated the machine for testing. Only the test program and normal background processes are running during the testing period.

5.4 Results

Figure 4 shows the effect of the file size on turnaround times at nine update rates, 0%, 1%, 2%, 5%, 7%, 10%, 15% and 100%. Four columns marked as Apache_1KB, rrApache_1KB, Apache_60KB and rrApache_60KB are compared for each update rate. The label such as 'Apache_1KB' indicates an experiment involving 1KB files with the normal Apache web server. In the small file (1KB) experiments the performance of Apache and rrApache are very close up to an update rate of 15%. However, in the large file (60KB) experiments the performance of Apache and rrApache are similar only at update rates below 5%. The larger file size slows down rrApache when the update rate is above 5%, and the times increases substantially when the update rate is over 50% (half of the requests create archived files). At 100% update (every request archives a file) the turnaround time of rrApache_60KB is approximately twice that of the Apache_60KB. We should note that we anticipate that most servers would realistically have update rates of less than 1% (one in every 100 requests is for a new version).

Recall that these are "peak load tests" of 3000 requests. So, at 0% update rate, there are approximately 100 and 52 requests per second handled for a 1-kilobyte page and 60-kilobyte page, respectively. From 0% up to about 5%, there is effectively no difference between rrApache and Apache. At update rates of 50% or above rrApache is twice as slow for large files. So the

cost of data management in TTApache (updating a history table and archiving) is within 10% of Apache when the update rate is less than 5%. At a 100% update rate, Apache handles about 73 and 42 requests per second for a 1-kilobyte page and 60-kilobyte page, respectively. Whereas, TTApache handles about 50 and 20 requests per second for a 1-kilobyte page and 60-kilobyte page, respectively. If an average frequency of a site is below 20 requests per second, TTApache can perform the additional functionality of the data management without sacrificing its performance even though large pages are requested, and even though the update rate is 100%, that is, files are modified on every request.

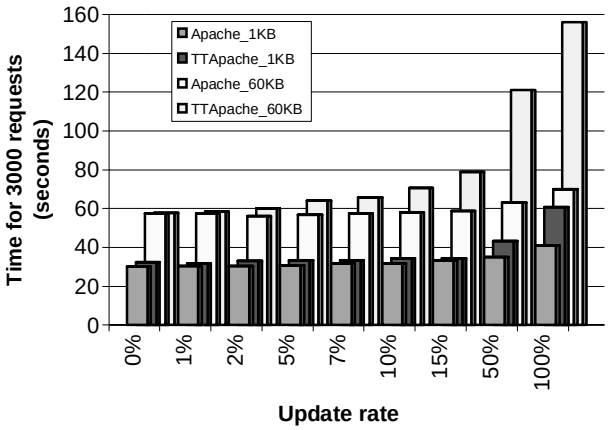

Figure 4 The effect of file size at different update rates

To understand the difference in turnaround time, we measured the amount of disk I/O. TTApache does more disk I/O than Apache. Depending on the update rate, the amount of disk I/O could triple. Consider an update rate of 100%. Each HTTP GET in Apache will trigger a file read (assuming that the server does not implement an internal cache). In TTApache, there may be an additional file write to create the archived copy. Also, the history table will have to be updated triggering another disk write operation. The additional disk I/O may further reduce the effectiveness of the file system cache since more files are being read and written. Figure 5 shows the cumulative disk I/O for different update rates for 1KB files. The region labeled 'apache' represents Apache. The other regions show the cumulative disk I/O for TTApache at different update rates. Surprisingly, TTApache at a 0% update rate has twice as much disk I/O as Apache. This is due to the additional database management system reads and writes and the lesser effectiveness of the disk cache. At a 100% update rate, the disk I/O use is ten times as great, rather than three times as we might expect. This is an artifact of the file system cache. Apache is doing fewer disk writes than TTApache, so files remain in the file system cache longer. Figure 6 shows the disk I/O for larger files. Since the file size is now much larger than the size of a database read or write, Apache and TTApache at low update rates perform about the same amount of I/O. But at 100%, TTApache performs triple the I/O as expected. Note that the disk I/O activity in the 60KB experiment is many times greater than the 1KB experiment. Both experiments show the cumulative disk I/O is proportional to the update rate. Since the update rate will usually be quite low (below 2%) as a rule of thumb, TTApache will increase disk I/O by 5% to 25%.

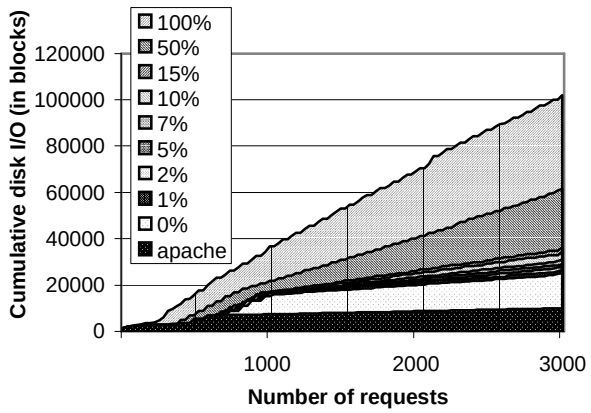

Figure 5 Cumulative disk I/O for several update rates (1 KB)

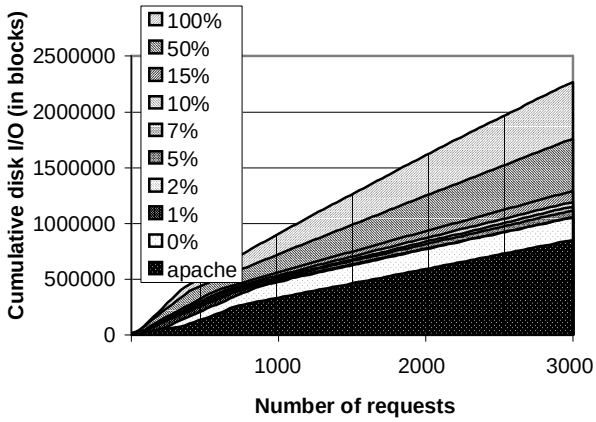

Figure 6 Cumulative disk I/O for several update rates (60 KB)

The last measurement concerns the overhead in processing transaction-time queries. These queries will be less efficient since they incur some additional processing and disk reads. Figure 7 shows different ratios of TTQueries (mix rates) on turnaround times for the nine update rates in a 3-D plot.

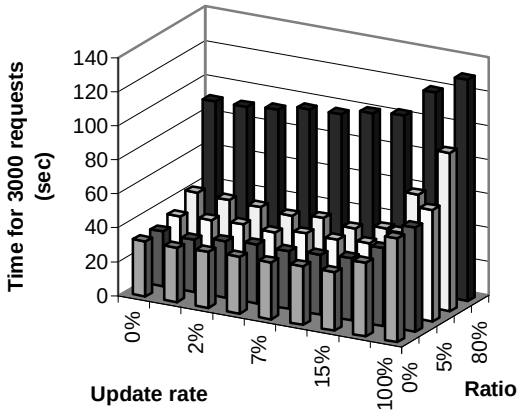

Figure 7 Measuring the cost of transaction-time queries

There are two trends to observe. First, within a mix rate the turnaround time increases as the update rate increases as we would expect from the previous experiments. Second, the

turnaround time slows as the percentage of rrQueries increases. At an 80% mix rate the turnaround times are two to three times that of the Apache web server. In practice, we anticipate that mix rates of below 10% will be common. We do not expect most queries to be transaction-time queries; if they are, this performance penalty may be significant.

6. CONCLUSIONS

A transaction-time web server provides transaction-time related services. In this paper we sketched a strategy to add transaction-time functionality to the Apache web server. We call the extended server rrApache. There are five benefits to the extension. First, rrApache performs automatic versioning when a document is read. This removes the burden of versioning from document authors. Second, rrApache allows users to request a desired document version or a document history in a URL. This increases the usability of archived documents. Additionally, users are able to restructure links in the requested document. Third, rrApache can distinguish between known and assumed versions of a document. Fourth, the size of the archive can be controlled by vacuuming versions that are not needed. Fifth, the cost of the additional functionality is modest under expected usage conditions. We built and tested an extended Apache web server that uses BerkeleyDB to manage version information. There is always an overhead on performance when additional functionality is added to an application. Our experiments showed that there is some additional cost with rrApache, but that for low update rates (which we believe is the scenario common to most web sites), the overhead is inconsequential.

REFERENCES

[1] Apache HTTP Server Version 1.3. Manual Page: ab. www.apache.org/docs/programs/ab.html. Current as of 7/8/2002.

[2] Brewington, B. and G. Cybenko. "How Dynamic is the Web?" in *Proceedings of the 9th Intl. WWW Conference*, Amsterdam, Netherlands, May 2000, pp. 257-276.

[3] Clifford, J, Dyreson, C., Isakowitz, T., Jensen, C. and R. T. Snodgrass. On the Semantics of Now in Temporal Databases. *ACM Transactions on Database Systems*, **22**(2), June 1997, pp. 215-254.

[4] Cho, J. and H. Garcia-Molina, "The Evolution of the Web and Implications for an Incremental Crawler", In *Proceedings of VLDB Conference*, Cairo, Egypt, Sep. 2000, pp. 200-209.

[5] Cobena, C., Abiteboul, S., and A. Marian, "Detecting Changes in XML Documents," in *Proceedings of ICDE Conference*, Heidelberg, Germany, April 2002, pp. 41-52.

[6] Concurrent Versions System, http://www.cvshome.org. Current as of April, 2003.

[7] F. Douglis, "Server-side tracking of new documents," in *Proceedings of the 1st International Workshop on Web Site Evolution*, Atlanta, GA, 1999.

[8] C. E. Dyreson, "Towards a Temporal World-wide Web: A Transaction-time Server," in *Proceedings of the Australian Database Conference*, Gold Coast, Australia, January 2001, pp. 169-175.

[9] C. E. Dyreson, "Observing Transaction-time Semantics with rrXPath", In *WISE2001*, Kyoto, Japan, December 2001, pp. 193-202.

[10] Florescu, D., Levy, A. and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[11] Graphic, Visualization, and Usability Center. GVU's Tenth WWW User Survey (October 1998). http://www.cc.gatech.edu/gvu/user_surveys/survey-1998-10/graphs/use/q11.htm.

[12] The Internet Archive. www.archive.org. Current as of February, 2003.

[13] Jensen, C. and C. Dyreson (eds.). A Consensus Glossary of Temporal Database Concepts - February 1998 Version, pages 367–405. Springer-Verlag, 1998.

[14] C. Jensen, C. S. "Vacuuming," in *The TSQL2 Temporal Query Language*, R. T. Snodgrass, editor, Chapter 23, pp. 451-462. Kluwer Academic Publishers, 1995.

[15] Lomet, D. and B. Salzberg. "Transaction-Time Databases," in *Temporal Databases: Theory, Design, and Implementation*. Chapter 16, pp. 388–417. Benjamin/Cummings, 1993.

[16] Marian, A., Abiteboul, S., Cobena, G. and L. Mignet, "Change-Centric Management of Versions in an XML Warehouse". in *Proceedings of the Very Large Data Bases (VLDB) Conference*, (Roma, Italy, 2001), Morgan Kaufmann, 581-590.

[17] McKenzie, E. and R. T. Snodgrass, "Extending the Relational Algebra to Support Transaction Time". In *Proceedings of SIGMOD Conference*, San Francisco, CA, May 1987, pp. 467–478.

[18] Netcraft Web Server Survey. www.netcraft.com/survey/. Current as of May, 2002.

[19] Padmanabhan, V. N. and L. Qiu, "The Content and Access Dynamics of a Busy Web Site: Findings and Implications". In *Proceedings of SIGCOMM*, Stockholm, August 2000.

[20] Rao, H., Chen, Y., and M. Chen, "A Proxy-Based Web Archiving Service", *Middleware Symposium*, Portland, OR, July, 2000. www.research.att.com-/~iproxy/archive, Current as of February, 2003.

[21] Roddick, J. and R. T. Snodgrass. "Transaction Time Support," in *The TSQL2 Temporal Query Language*, R. T. Snodgrass, editor, Chapter 17, pp. 319–325. Kluwer Academic Publishers, 1995.

[22] Skyt, J., C. S. Jensen, and L. Mark. "A Foundation for Vacuuming Temporal Databases," *Transactions on Data and Knowledge Engineering*, **44**(1), December 2002, pp. 1-29.

[23] Soo, M. D. Bibliography on Temporal Databases. *SIGMOD Record*, **20**(1):14–23, Mar. 1991.

[24] W. F. Tichy, RCS–A System for Version Control, *Software–Practice & Experience*, **15**(7), July 1985, 637-654.

[25] Tsotras, A. K. Temporal Database Bibliography Up-date. *SIGMOD Record*, **25**(1):41–63, March 1996.

[26] Xyleme, enabling intelligent access to XML content. www.xyleme.com. Current as of February, 2003.

[27] W3C-XML, 2002. Extensible Markup Language (XML) 1.0 (Second Edition). www.w3.org/TR/REC-xml. Current as of 7/8/2002.