A VACUUMING-ENHANCED TRANSACTION-TIME WEB SERVER

By

Yingxia Wang

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

May 2003
To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of Yingxia Wang find it satisfactory and recommend that it be accepted.

______________________________
Chair

______________________________
Acknowledgement

Most of all, I would like to express my sincere appreciation to my advisor, Dr. Curtis Dyreson, for his invaluable guidance, encouragement, and support throughout this research.

I would like to thank my committee member Dr. Zhe Dang and Dr. Carl Hauser. Their comments are precious for improvement of the thesis. I also would like to thank Dr. Ali Saberi and Dr. Shira Broschat. Without their encouragement and care, this thesis wouldn’t be accomplished.

I am grateful to my friends, Cheng Li, Tengshun Peng, Da Meng, and Lihua Ran, for their advice and help during my course study.

I thank my parents and my brother for their love and support all the time.
A VACUUMING-ENHANCED TRANSACTION-TIME WEB SERVER

Abstract

by Yingxia Wang, M.S.
Washington State University
May 2003

Chair: Curtis Dyreson

A transaction-time web server archive document versions during HTTP requests to create a complete history of the documents at a website. The server also processes transaction-time queries (TT 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. Both the archive and the history table will grow over time as new versions are created, and eventually exceed the storage capacity.

This thesis investigates a technique called vacuuming to control the size of the archive. Vacuuming removes selected versions from an archive. A vacuuming policy identifies which versions to vacuum. This thesis develops a wide range of vacuuming policies. Unfortunately, vacuuming controls the growth of the archive by sacrificing the completeness of the server’s history. Some TT Queries can no longer be satisfied, so query repair strategies are presented to tune those queries.

Like the archive, the history table also grows over time. Two techniques for recouping space in the history table, coalescing and obliteration, are introduced. This thesis also explores resource migration, which migrates the history of a resource when the resource moves to a new location.
All of the new functionalities were 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 thesis presents a logical model for the design and URI-compatible syntax for supporting the new functionality.

Transaction-time support, vacuuming, query repair, coalescing, obliteration, and resource migration were implemented as a modular extension of Apache v2.0. This thesis describes porting the implementation of transaction-time support in \( \text{TT} \)Apache, which was based on Apache v1.3, to \( \nu\text{TT} \)Apache (vacuum-enhanced, transaction-time Apache), based on Apache v2.0, as well as implementing the new functionality. Since the Apache internals changed significantly in v2.0, the port required a substantial redesign of the transaction-time support. The thesis presents extensive empirical measurements on the performance overhead of the transaction-time and vacuuming functionality in \( \nu\text{TT} \)Apache. The empirical work suggests that the additional functionality has a marginal impact in most server environments.
# Table of Contents

Acknowledgement ............................................................................................................. iii

Abstract .............................................................................................................................. iv

Table of Contents ............................................................................................................... vi

List of Tables ..................................................................................................................... ix

List of Figures ................................................................................................................... x

1 Introduction ................................................................................................................. 1

1.1 Motivation ........................................................................................................... 5

1.1.1 Control Archive Size ................................................................................... 5

1.1.2 Shorten History Table ................................................................................. 7

1.1.3 Flexible Management .................................................................................. 7

1.2 Vacuuming .......................................................................................................... 8

1.2.1 Queries on Vacuumed Versions ................................................................ 10

1.2.2 Reduces Storage Cost ............................................................................... 10

1.2.3 Query Repair ............................................................................................. 11

1.3 Related Work .................................................................................................... 12

1.4 Outline of Thesis ............................................................................................... 15

2 Motivation ................................................................................................................. 16

2.1 Motivation Example .......................................................................................... 16

2.2 Resource Management Examples ..................................................................... 17

2.3 TQuery Examples .............................................................................................. 21

3 A Logical Model for Vacuuming .............................................................................. 23

3.1 Vacuuming Policies ............................................................................................ 23
3.1.1 Periodic Sieve ................................................................. 24
3.1.2 Percent-Different Sieve .................................................. 25
3.1.3 Version-Window Sieve .................................................... 28
3.1.4 Time-Window Sieve ....................................................... 29
3.1.5 Composite Sieve ........................................................... 31
3.2 A Formal Model for Vacuuming ........................................... 32
3.3 Coalesce ............................................................................. 37
3.4 A Formal Model for Coalesce ............................................. 39
3.5 Obliteration ........................................................................ 40
3.6 A Formal Model for Obliteration ....................................... 42
3.7 Repairing Queries for Vacuumed Documents ....................... 42
   3.7.1 A warning that shows the requested version has been vacuumed. .... 43
   3.7.2 A warning that shows this request is affected by vacuuming, followed by a
       repairing query result. ........................................................................ 43
   3.7.3 No explanatory result information is given. ............................. 45
3.8 A Formal Model for Query Repair ....................................... 45
3.9 Syntax ............................................................................... 50
4 Implementing Vacuuming in Apache ..................................... 54
   4.1 Port to Apache v2.0 ......................................................... 54
      4.1.1 New Features of Apache v2.0 ......................................... 54
      4.1.2 Upgrade ..................................................................... 57
   4.2 Data Representation ........................................................ 59
      4.2.1 Various Version States ................................................. 59
      4.2.2 Policy Table ............................................................ 60

vii
List of Tables

Table 1-1 Comparison of Three Storage Control Methods .................................................. 6
Table 3-1 Difference Table for Earliest-30%-Different Sieve Example ............................ 26
Table 3-2 Difference Table for Latest-30%-Different Sieve Example ............................... 27
Table 3-3 BNF for Vacuuming .......................................................................................... 51
Table 3-4 Notation for Vacuuming Policies ....................................................................... 52
Table 3-5 Notation for Option Terms ............................................................................... 52
Table 3-6 BNF for Enhanced 77Query ............................................................................. 53
Table 3-7 Notation for Version Category .......................................................................... 53
Table 4-1 Event Handlers for 77Apache ........................................................................... 58
Table 4-2 File Version Status ........................................................................................... 60
Table 4-3 Policies for Each File ......................................................................................... 62
List of Figures

Figure 1-1 77Apache Model ........................................................................................................ 3
Figure 1-2 Vacuuming Example ................................................................................................. 9
Figure 2-1 Motivation Example ................................................................................................. 17
Figure 3-1 Periodic Sieve Example ........................................................................................... 24
Figure 3-2 Earliest-30%-Different Sieve Example ...................................................................... 26
Figure 3-3 Latest-30%-Different Sieve Example ......................................................................... 27
Figure 3-4 Sliding Version-Window Sieve Example ................................................................. 29
Figure 3-5 Time-Window Sieve Example .................................................................................. 30
Figure 3-6 Static Time-Window with Periodic Sieve Example .................................................. 31
Figure 3-7 Query Vacuumed Versions Example ...................................................................... 38
Figure 3-8 Coalesce Example .................................................................................................. 39
Figure 3-9 Obliteration Example ............................................................................................... 41
Figure 3-10 Example for Repairing Redirect .......................................................................... 45
Figure 4-1 Overview of the APR Layer .................................................................................... 55
Figure 4-2 Bucket-Brigade View of a Document ................................................................. 56
Figure 4-3 Filter Chain in Apache v2.0 .................................................................................... 56
Figure 4-4 Core and Modules in Apache v2.0 ........................................................................ 57
Figure 4-5 History Schema ..................................................................................................... 59
Figure 4-6 File Version Status Transition .............................................................................. 61
Figure 4-7 Policy Schema ...................................................................................................... 61
Figure 4-8 Policy Tree Structure ............................................................................................ 62
Figure 4-9 System Architecture Figure .................................................................................. 65
Figure 5-1 The effect of file sizes with different update rates................................. 72
Figure 5-2 The effect of file sizes on TTApache.......................................................... 72
Figure 5-3 Virtual Memory used for different update rates (1KB).............................. 74
Figure 5-4 Virtual Memory used for different update rates (60KB)............................ 74
Figure 5-5 Idle memory for different update rates (1KB).......................................... 75
Figure 5-6 Idle memory for different update rates (60KB)....................................... 76
Figure 5-7 Buffer used for different update rates (1KB)........................................... 77
Figure 5-8 Buffer used for different update rates (60KB)......................................... 77
Figure 5-9 Cache used for different update rates (1KB)............................................ 78
Figure 5-10 Cache used for different update rates (60KB)....................................... 79
Figure 5-11 Cumulative disk I/O for different update rates (1KB).............................. 80
Figure 5-12 Cumulative disk I/O for different update rates (60KB)............................ 80
Figure 5-13 Cumulative swap I/O for different update rates (1KB)............................ 81
Figure 5-14 Cumulative swap I/O for different update rates (60KB)........................... 81
Figure 5-15 CPU used for different update rates (1KB)............................................. 83
Figure 5-16 CPU used for different update rates (60KB).......................................... 83
Figure 5-17 The effect of the ratio of TTQueries (1KB)............................................ 84
Figure 5-18 The effect of vacuuming with different update rates and file sizes............ 86
Figure 5-19 The effect of different vacuuming policies with different update rates and file sizes......................................................... 87
Figure 5-20 The effect of repairing with different update rates and file sizes.............. 88
1 Introduction

Since its emergence a decade ago, the World Wide Web has grown from a small research project into a vast repository of information and a popular medium for communication. The web is actually a large collection of heterogeneous documents that are created, maintained, and modified by independent authors. The web is increasingly being used in all aspects of the society; for example, customers use the web to search for shopping goods that match their interest, researchers take the web as a digital library to search for and locate materials in their area, and organizations use the web to expand their business and interact with their customers.

A web server works as a communication media that makes documents accessible to online web users. The server maps an incoming HyperText Transfer Protocol (HTTP) request to a document and returns the document to the user.

Web documents, unlike standard text collections, can contain both multimedia (images, etc.) and connections to other documents (through hyperlinks). A document is often composed of several files, typically a main file in HyperText Markup Language (HTML) and included files.

The interlinking nature of web documents leads to some concerns, when considering their dynamically evolving nature. Web documents change over time. Therefore, broken links occur when links to documents outlive those documents, e.g., when the referenced documents are moved. The Graphic, Visualization, and Usability Center’s 10th survey shows the broken link problem as the third most frequent web problem [Gvu98]. In order to reduce the number of broken links, web search engines nowadays are designed to
periodically update the links they maintain. However, documents evolve in an unpredictable frequency. It could occur right after the search engine refreshes.

Although web documents evolve frequently, the old or outdated information is still of reference or analysis value. Mechanisms are desirable that can offer a channel to trace back to the information. A web archive, an integrated back-end information container in web search engine, e.g., www.google.com, gives a cached snapshot of the document.

In the database research field, a transaction-time database extends the general database schema to keep track of all past states of a database [LS93]. Transaction time of a particular fact is the time between when the fact is inserted into the database and when it is deleted from the database (an update is looked at as a deletion operation followed by an insertion operation). The history of a database is useful for real world database applications that require accountability and traceability. Both time-slice (queries for data as of a past state) and rollback to an old database state are supported in a transaction-time database.

A transaction-time web server stores the history of a web server. In the context of the web, there are no update transactions, since browsers or other online users of the web only have read access to the data. They are rarely permitted to insert, or update, or even delete data. Updates to the data are irregular and hidden from the online users. So, detecting the updates occurs only when the updated data are re-read by the users, at which time a comparison can be made with the previous version of the same data. A difference means that an update occurred sometime since the previous read. Transaction
time for the data starts from the system time of a modification that creates a version of the data, and ends at the time of the next modification of the data.

A transaction-time web server provides version time-slice, version difference, and version restoration services. A transaction-time web server allows queries for old versions of a document, called version time-slice [Dyr01]. Users can specify a timestamp to point out which version they desire. The server retrieves the matched version and returns it to the users. With all versions stored, the server can compute the difference between two versions of a document, called version difference. Furthermore, the history of stored versions makes the server capable of tracing back to an old state, termed version restoration.

![Figure 1-1 77Apache Model](image)

77Apache is an extension of the Apache web server that provides transaction-time functionality [Lin02]. Figure 1-1 shows the high level architecture of the server.
Apache stores resource versions in an archive, with a database (history table) recording the version history information. A resource, identified by a URI, consists of a main file formatted in HTML or XML, together with several included files. A modification to either the main file or any of the included files for the resource will generate a new version of that resource. Apache provides HTTP-compatible queries allowing online users to travel among different versions of resource. For example, a time-slice query lets the user specify the version by a particular timestamp, e.g. “http://URL?Feb-14-2003”; a single step query of “http://URL?now”, “http://URL?pre”, or “http://URL?next” lets the user retrieve the current version, previous version, and next version of the requested resource, respectively; multi-step queries of format “http://URL?pre.pre” can locate the resource by concatenating operations on the relationships among versions; and version history queries like “http://URL?history”, and “http://URL?history(Feb-14-2003,Feb-25-2003)” give a summary report of all versions of that resource in the specified time range if available. The transaction-time query also has a link-rewriting scheme that enables online users to rewrite the hyperlinks in a document to include transaction-time queries.

In this thesis, we enhance a transaction-time web server with vacuuming, which includes vacuuming policies and resource managements. Transaction-time server archives documents by automatic versioning. As documents evolve, archive grows forever. Vacuuming is introduced to restrict the archive size.
1.1 Motivation

While a transaction-time server provides additional functionality it also increases the demand for system resources. In 77Apache, the ever-growing archive size will lead to a resource capacity problem. We discuss this concern in Section 1.1.1. The history table (a database internal to 77Apache) also continuously grows over time, which increases the cost of processing transaction-time queries. We explore this problem in Section 1.1.2. The archive itself is a collection of information, which requires maintenance. We discuss the desirability of flexibly managing the archive in Section 1.1.3.

1.1.1 Control Archive Size

A primary concern in running and maintaining 77Apache 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 use differences or eliminate versions. There are generally three methods. One is to store 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 that 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. The comparison of those methods is shown in Table 1-1.

In the table, method (a) stores the difference between two consecutive versions, which is the smallest one, therefore it makes the best usage of storage. Method (c) stores the whole file versions, thus occupying the largest storage capacity. Method (b) is in
between in the storage consumption. When processing time-slice, method (c) can retrieve
the queried version fastest; method (b) should reconstruct the queried version according
to the original file version and the difference; method (a) has to reconstruct the queried
version by accumulating all the differences ever since the beginning, thus more time is
needed. Obviously, the reconstruction in method (a) is transitive in file version history,
which indicates each stored difference is indispensable and cannot be eliminated or
hidden from others. While in method (b) and (c), every stored difference or file version is
independent of each other, hence can be eliminated or hidden from others.

<table>
<thead>
<tr>
<th>Measured Aspects</th>
<th>Store Version Difference (Reference Chosen)</th>
<th>Store Entire Version (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computed Version (a)</td>
<td>Stored Version (b)</td>
</tr>
<tr>
<td>Storage utilization</td>
<td>Smallest</td>
<td>Medium</td>
</tr>
<tr>
<td>Time-slice efficiency</td>
<td>Slowest</td>
<td>Medium</td>
</tr>
<tr>
<td>Ability of elimination</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Ability of hiding</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1-1 Comparison of Three Storage Control Methods

In Apache, we adopt method (c) to build the archive. The archive stores each
complete file version. File versions are eliminated to reduce the archive size.

The archive keeps file versions for all documents under the website. Documents
however are normally owned by different individuals, and each may have his preference
on which file versions to be eliminated. For instance, one may want his resource versions
older than two years to be deleted, while another person would like to delete resource
versions older than three years. Hence, each deletion should be predefined under some
policy. File versions are deleted according to policies specified by document authors or a
site administrator.
1.1.2 Shorten History Table

The history table of Apache also has an unrestricted growth. In Apache, the history table is an indexing mechanism that associates a file version to a file in the archive. Every file version adds one tuple to the history table. As the versions of each file increase, the history table also grows over time.

As long as some versions are no longer desired or used, their corresponding tuples in the history table are useless, or even harmful to the performance of processing transaction-time queries and other HTTP requests. Versioning (constructing resource versions from the main file versions and the included file versions) is performed as a nested loop, which has to walk through each version tuple in a file’s lifetime in the history table. Useless versions’ entries increase the cost in processing.

1.1.3 Flexible Management

Document migration leads to an outdated history for a document, thus requiring “history migration”. The archive is the core element in supporting versioned documents. In the real world, the website structure is dynamic. Documents are grouped and organized under the specified website’s document root. Designers can adjust the structure, such as to rearrange the organization of documents, or to upgrade the website architecture. The migration of documents makes the URL no longer workable, since the URL is paired with the location of the document. The changes of document location result in orphaned URLs because the association between a document and a URL is lost. However, the association between a URL and a file version in the archive still exists and is useful for individuals interested in the history information of the document. A mechanism should be
provided to make the URL reusable for the documents possessive in new location. That is, to recover the connection between documents and their archived version files via URL.

As mentioned before, file versions could be removed either because they are outdated or because of a user’s desire. From the perspective of resource owners, for example, they would like to choose file versions from a particular time range and mark them all as “discarded” versions; they would also like to designate those versions of little difference to be “discarded” versions. But, it is a boring work for any individual to repeat choosing manually. They would like the discarding task to be carried out automatically. Hence, it is desirable to let Apache manage the archived versions.

1.2 Vacuuming

Vacuuming, in a transaction-time database, means to physically delete records of past states [Jen95]. 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, which 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 [Jen95].

In a transaction-time web server [Dyr01] 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 on the time line from oldest to newest. 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 Figure 1-2 shows the older versions being vacuumed. However, policy 2 of Figure 1-2 shows the policy of vacuuming the first version of every two consecutive versions; therefore, V1 and V3 will be vacuumed.

Figure 1-2 Vacuuming Example
1.2.1 Queries on Vacuumed Versions

A transaction-time web server without vacuuming retains all its previous status in the archive. If the document is requested when the current time is $t$, the same document is returned for a time-slice query at time $t$. A request for the document version as of time $t$, and the request issued at some time later that asks for this document version will both give the same result. We call this *faithful property* of the archive. With this property holding, if a request for a past document version is not found, it means that there was never a version in the archive.

With vacuuming introduced into transaction-time web server, the faithful property of archive is compromised. A query for a vacuumed version cannot retrieve a version after it has been vacuumed.

An example explains how the faithful property is violated. In the example, policy 2 of Figure 1-2, a web server with vacuuming gives an HTTP_NOT_FOUND as response to the HTTP request for V3 of “sports.html”. However, a transaction-time web server is able to find this version V3 in its archive and return it to the browser in the absence of vacuuming.

1.2.2 Reduces Storage Cost

A crucial concern in maintaining an archive is its ever-growing requirement for storage space, as both the archive and the history table are increasing to occupy the storage.
With vacuuming introduced into Apache, some file versions can be eliminated from the archive thereby freeing space in secondary storage. The archive can thus be restricted in its growth. There are several kinds of vacuuming policies that will be discussed in Chapter 3. File versions in the archive are selected and then vacuumed. The selection criteria are either based on the file content or its lifetime. File versions with small changes or older file versions are usually selected for vacuuming.

Vacuuming reduces the size of the archive, but not the history table. With multiple vacuumed versions recorded in the history table, we can eliminate those tuples to shrink the size of history table, termed as coalesce. Coalesce combines temporally consecutive vacuumed versions into a single obliterated version. Moreover, obliteration eliminates all versions in the archive for the specified file, followed by coalescing its versions in the history table.

1.2.3 Query Repair

Repairing compensates the potential loss of the faithful property because of vacuuming. Query repair remedies the misinterpretation of a request, to rebuild broken links and decrease the page-missing rate for vacuumed versions. As a result, the user request can always get an approximate response from the web server, even though the file or the included file no longer exists. With repairing, a vacuuming-enhanced web server can closely satisfy the faithful property in some sense by assisting the online user to interpret and understand the query result.

There are four ways to repair a query. The default strategy is to do nothing. More actively, information can be returned to the user notifying that the requested information
has been vacuumed. Other strategies are to direct the query to the nearest “valid” version, either a previous or next version.

1.3 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 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 [Int03]. Unlike the Internet Archive, Apache 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 supports 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 [Rao’00]. 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. Apache, 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 Apache, iPROXY uses URL-munging to support transaction timeslice, but without a link-rewriting component.

Apache provides much finer-grained versioning than iPROXY or the Internet Archive. Apache 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 [Dou99].

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. Finally, Apache is the only archiver to support vacuuming, which allows users to control the growth of the archive.
The final related system is Xyleme [Xyl03]. 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 [Cob’02]. Unlike Xyleme, Apache is a very primitive versioner. Apache does not compute changes between versions, rather we compute only a measure of the size of the change 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.

Concurrent Versions System (CVS) is a dominant version control system for developers to maintain their source codes [Cvs03]. 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 [Tic85]. Apache 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.

In the context of transaction-time database, a semantic foundation for vacuuming has been presented [Sky’02]. This work studies the semantics and syntaxes of vacuuming specifications, which is composed of a removal specification part and a keep
specification part that overrides the removal part. Apache supports only a removal specification. Further, the detection and evaluation of vacuum-affected queries are explored, along with correct options to those queries. Due to the vacuuming specification, modification (update or delete) of transaction-time database is constrained. It makes no sense to insert a removal specification part that conflicts with an existing keep specification part. Vacuuming specifications are stored as temporal relations in database, thus making it possible to vacuum them as well.

1.4 Outline of Thesis

The remaining contents of this thesis are structured as follows.

Chapter 2 gives examples to motivate how Apache v2.0 offers resource management utilities. Chapter 3 presents the logical model of vacuuming, including vacuuming policies, repairing strategies, and obliterations. Concept definitions and syntaxes are also given to formalize the ideas and functionalities. Chapter 4 introduces implementation details. The technique for porting Apache to Apache v2.0 is presented first, followed by the new algorithms for the vacuuming. Experiments are given in Chapter 5. Conclusions and future work are covered in Chapter 6.
2 Motivation

77Apache v2.0 is designed to support resource management. It enables resource owners to designate policies for managing documents. Those policies collaborate to control the growth of the archive by discarding useless, outdated, and redundant file versions. Also, a resource can be moved to a new location while keeping the correct association with its archived versions. 77Queries against vacuumed versions result in a “Page Not Found” response. 77Apache v2.0 will repair those queries to minimize the page-missing rate. In this chapter, we give a motivating example to illustrate the additional functionalities of 77Apache v2.0. Section 2.1 briefly introduces the example. Section 2.2 explains the resource management examples. Section 2.3 describes the repaired 77Query examples.

2.1 Motivation Example

To demonstrate the functionality, we introduce a concrete example. We assume a virtual online shopping website, referred by the following URL:

http://www.shop.com/

The organization of this website is depicted in Figure 2-1. The “Home Directory” is located by the URL given above. Under this directory, there is an “index.html” which is the entry page for this website. It introduces this website and shows any news for this website, such as each upgrade of this website, each new merchant’s attendance as an image, and each new policy for online shopping. The “images” directory has all image files collected. The “goods” directory is the core component: it includes the “guide.html” file that gives the guidelines for a shopping tour. The “in-stock”
subdirectory contains (1) the information for all in-stock goods, e.g., a complete list including their quantities and unit prices, (2) an on sale list including the discount and the duration of the sale, and (3) a complete sale record for those goods. The “out-of-stock” subdirectory contains a list of the currently out-of-stock goods. The “merchants” directory maintains the trading information with each merchant supplier. The “customers” directory maintains the information of each registered customer, and their corresponding orders.

A web master manages resources under this website. Customers or other users browse documents in the website.

![Site Map](http://www.shop.com)

Figure 2-1 Motivation Example

### 2.2 Resource Management Examples

We give examples to show how Apache v2.0 helps the web master manage and maintain this website.
This website is updated everyday with the latest information. In particular, the quantity of each item for sale varies as online transactions with customers and trades with merchants are completed. The unit prices for some goods are changing due to various reasons, e.g., holiday sale, clearance sale. Some hot-sell goods may be out of stock if the supplier cannot fill an order for new stock in time. Each day, online users search for desired merchandise, especially sale items. Customer orders are entered in “order.html”.

We assume this shopping website has a high number of accesses per day, and achieves outstanding sale records.

Apache adds an archive to store the past states of this website. That is, every version of a document is saved in the archive. Over time, the archive grows without limit and gets larger and larger. After two years, the web master decides that some of the archived versions for “/goods/in-stock/list.html” is useless. He wants to discard the first year’s versions of that file. With Apache v2.0, he can easily reach this goal by issuing a vacuuming operation.

**Example 1:** Vacuum files within a fixed time-window


The term “vacuum” represents a vacuuming request, with the “t-window” specifying the time-window. The next two arguments show the boundary of the window. “begin” means the time of the first file version for “list.html”, while the “begin+365” stands for one year from the “begin”. The vacuuming works only on
the main file of this document “list.html”. The archived versions of included files are not affected.

The web master realizes that he can notify the web server to automatically vacuum some versions of the files located under the “/goods/” directory, hence relieving him from periodically executing the vacuuming operation. For example, he wants to store only every other version of all the documents under that directory.

**Example 2:** Vacuum the first one for every two versions in the “goods” directory

- http://www.shop.com/goods/?vacuum(v-window,1,2)

The term “v-window” designates a version-window, which has two arguments “1” and “2”, meaning that vacuuming the first one version for every two versions. After several weeks, however, the web master no longer wants the vacuuming to work on the file “/goods/in-stock/sale-record.html”. He hopes to keep all the versions in the archive, since that information is useful for later reference and commercial statistics. So, he issues the following request to disable the vacuuming on that file for the future.

**Example 3:** Disable future vacuuming on a specified resource


The “includeparent=no” term prevents the setting of “v-window,1,2” on directory “/goods/” (imposed in Example 2) from working on the “/goods/in-stock/sale-record.html” file. Since the file versions for “/goods/in-
stock/list.html” now archive every other version, if an online user issues a query for the vacuumed version, he will get an HTTP_NOT_FOUND error page. The web master prefers a more elegant interaction with his customers. He would like to give the previous valid document version as a reply for the user’s request. He can achieve this as follows.

**Example 4**: Repair a request for a vacuumed version to the previous “good” version


The history of the document “/goods/in-stock/list.html” has been vacuumed for its first year. The web master doesn’t want online users to know about the vacuuming. He just hopes to keep the online users aware that the year’s history is eliminated. He can issue a coalescing request as follows.

**Example 5**: Coalesce vacuumed versions


It would be a repetitious work to issue such a command for each document under the “/goods/” directory. The web master can simplify his work by coalesce the directory.

**Example 6**: Coalesce on directory and all subdirectories

- http://www.shop.com/goods/?coalesce(recursive)

One day, the e-shop rearranges the business trading relationship with the merchant suppliers. After the negotiation, the merchandise offered online are all in stock. Hence,
the web master desires to eliminate and discard all the past history information of “/goods/out-of-stock/list.html”. He can send the obliteratation request as follows.

**Example 7: Obliterate history**


At the end of year, this e-shop creates statistics of annual sales in order to plan the next year’s service for the customers and merchants. Before that, the web master wants to reorganize this website. For instance, he will move the “/goods/in-stock/sale-record.html” to another location, “/statistics/sale-record.html”. But he still wants the historical versions accessible for the new URI. He can achieve this by forwarding the “history” as follows.

**Example 8: Forward the history**


### 2.3 77Query Examples

With 77Apache v2.0’s additional features, a 77Query could involve a vacuumed version. As to the examples in the previous section, queries for “/goods/in-stock/onsale.html” is an example. Online users can now actively tune the query result by repairing a query.

**Example 1: Repair a query to the next “good” version**
This query with “\texttt{pre,pre}” tries to retrieve the previous version (specified by the first “\texttt{pre}”) of the “\texttt{onsale.html}” with the embedded hyperlinks also rewritten to point to a previous version (specified by the second “\texttt{pre}”). If the previous version is not a valid version, the web server will turn to the current version and return it as if it is a valid version (also with links rewritten), as specified by “\texttt{repair=future}”.

In \texttt{Apache v2.0}, a file version can have a variety of statuses, including “known”, “assumed”, “vacuumed”, and “obliterated”. A history query will show a full list of all versions for the requested URI. Online users can restrict a history query to versions with particular statuses.

**Example 2**: History query for known and assumed versions


Terms “\texttt{k}” and “\texttt{a}” represent “known” and “assumed”, respectively, which points out the statuses for file versions. This query only retrieves known and assumed versions for the document.
3 A Logical Model for Vacuuming

Apache v2.0 imposes vacuuming policies on versions in the archive, as well as providing the repairing mechanisms for the vacuuming affected Query. Moreover, coalesce and obliteration can be applied to eliminate tuples in the history table thereby reducing its size. We discuss the logical model of vacuuming in this chapter. Two aspects need to be investigated when designing the vacuuming solution. One is to design the vacuuming policies – selecting which version(s) in the archive should be vacuumed, and the other is to design the repairing strategy. We present different designs for vacuuming polices in Section 3.1 and the corresponding formal model in Section 3.2. Coalesce is discussed in Section 3.3, with its formal model described in Section 3.4. Obliteration is described in Section 3.5 followed by its formal model in Section 3.6. The design and formal model for repairing strategies are given in Section 3.7 and Section 3.8 respectively. As a summary, we give the syntax description for all functionalities in Section 3.9.

3.1 Vacuuming Policies

We design vacuuming policies on the basis of the properties of the archive, and we will analyze the storage saving efficiency for each policy. One feature of the archive can be found is that changes among successive versions will generally be small. The first category of policies focuses on document content. We describe the details for each of them in Section 3.1.1 and Section 3.1.2. Another feature of the archive is that older versions are less accessed, compared with the new versions of the file. There can be several policies under the consideration of time period. We represent those policies in
Section 3.1.3 and Section 3.1.4. As a combined consideration of those two properties, we show the composite policies in Section 3.1.5.

Before digging into vacuuming policies, we observe a rule that the current file version should never be vacuumed.

### 3.1.1 Periodic Sieve

The periodic sieve policy involves version numbers. It has two parameters: *frequency* and *periodicity*. With particular values of the two parameters $f$ (frequency) and $p$ (periodicity) specified, it vacuums the first $f$ versions for every $p$ versions of the file in the archive. For example, Figure 3-1 shows how the periodic sieve with frequency 2 and periodicity 3 affects the version history. File versions V1, V2, V4, V5 are vacuumed. File version V7 is not vacuumed, although it conforms to the vacuuming policy, because it is the current version.

![Figure 3-1 Periodic Sieve Example](image-url)
For all of the vacuuming policies, we assume \( a \) to be the size of the history of the file before vacuuming, that is the number of versions for that file. And we assume that each file version is uniformly, that is, of the same size and duration.

For the periodic sieve policy, the size of vacuumed history will decrease from \( a \) to approximately \((1 - \frac{1}{p})a\) .

Since the file version number for each file version is unique and vacuuming does not affect the version number, the effect of vacuuming can be overlapped. Figure 3-2 shows the example.

![Figure 3-2 Effect of Overlapped Vacuuming](image)

**3.1.2 Percent-Different Sieve**

The percent-different sieve policy focuses on the percentage of change among file versions. With the percentage specified, \( n\% \), it vacuums versions that are less than or equal to \( n\) percent different from their previous version. The percentage comparison is made on two non-vacuumed versions, which may not be adjacent to each other. In
general, if the difference is uniformly distributed, then the size of a file history will shrink to \((1 - \%_{100})a\).

Since the vacuuming direction could be either forward or backward along the timeline of the version history and thus leads to different comparison results, we explore two cases for this policy.

![Diagram](image)

**Figure 3-3 Earliest-30%-Different Sieve Example**

<table>
<thead>
<tr>
<th>Two Versions Pair</th>
<th>Difference Value(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V1, V2)</td>
<td>20</td>
</tr>
<tr>
<td>(V1, V3)</td>
<td>31</td>
</tr>
<tr>
<td>(V3, V4)</td>
<td>40</td>
</tr>
<tr>
<td>(V4, V5)</td>
<td>32</td>
</tr>
<tr>
<td>(V5, V6)</td>
<td>28</td>
</tr>
<tr>
<td>(V5, V7)</td>
<td>50</td>
</tr>
</tbody>
</table>

*Table 3-1 Difference Table for Earliest-30%-Different Sieve Example*

The earliest-percent-different sieve works forward on the history versions beginning from the earliest. We give a concrete example in Figure 3-3 to demonstrate how this policy operates on a history. The difference percentage between two versions is listed in Table 3-1. Assume they are denoted as \(\text{diff}(m, n)\), with the symbol ‘%’ ignored. File
version V2 is vacuumed since \( \text{diff}(V1, V2) = 20\% \), which is less than 30\%. File version V3 is compared with V1 instead of V2 because V2 is now a “vacuumed” version. \( \text{diff}(V1, V3) \) is more than 30\%, therefore V3 is kept. The vacuuming decision can be inferred similarly for other version pairs.

![Time Line Diagram](image)

**Figure 3-4 Latest-30\%-Different Sieve Example**

<table>
<thead>
<tr>
<th>Two Versions Pair</th>
<th>Difference Value(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V1, V2)</td>
<td>20</td>
</tr>
<tr>
<td>(V2, V3)</td>
<td>22</td>
</tr>
<tr>
<td>(V3, V4)</td>
<td>40</td>
</tr>
<tr>
<td>(V4, V5)</td>
<td>32</td>
</tr>
<tr>
<td>(V5, V6)</td>
<td>28</td>
</tr>
</tbody>
</table>

The latest-percent-different sieve works backward on the history beginning from the current version. According to the “never vacuum the current file version” rule, this policy just skips checking the current file version. Considering the previous example before vacuuming, this policy of the same percentage 30\% performing on the version history is shown in Figure 3-4, along with the difference percentage data listed in Table 3-2.
Compared with the earliest-percent-different sieve, file version V3 is vacuumed in this policy as the diff(V2,V3) is less than 30%.

### 3.1.3 Version-Window Sieve

The version-window sieve sets up a version window, modeled as $[\text{from}, \text{to}]$, where $\text{from}$ and $\text{to}$ represent particular version numbers. A file version with the version number inside the window will be vacuumed. After vacuuming, the size of history versions in archive is approximately decreased to $(1-((\text{to-}\text{from}+1)/\text{current}))a$, where “current” stands for the current file version number.

Since current is evaluated to different values at different times, the version-window can be either static or sliding.

A static version-window sieve has fixed version numbers for both $\text{from}$ and $\text{to}$. Hence, it forms a fixed window on the version history.

A sliding version-window sieve has current as the $\text{to}$ value, the window is semi-aligned with the most recent file version. From the point view of time line (earliest to latest, left to right), this window is right aligned. The window slides when a new version is generated. Only the latest versions are kept, older versions are vacuumed.

Figure 3-5 gives an example showing how the sliding version-window sieve policy works, with the window of $[0, \text{current}-5]$. It vacuums file versions other than the latest 5 versions. After file version V8 is created, file version V3 has to be vacuumed since it is outside the range of the latest five file versions.
3.1.4 Time-Window Sieve

The time-window sieve policy is similar to the version-window sieve policy, except that the window is specified on time rather than on version number. The policy sets up a time window to vacuum versions that are inside the window. Assume the time window has a model of \([\text{from}, \text{to}]\), where \text{from} and \text{to} represent the beginning and end of the time window, respectively. We set the decision criteria on each file version’s modification time. We say this file version is inside a time window range if its modification time is inside the time window. Figure 3-6 gives an example showing this policy. File versions V2, V3 and V4 are vacuumed because they are inside the window. File version V1 is not vacuumed since its modification time is outside the window.
Assume that new versions are created at a uniform rate, then the final archive size is reduced to \(1-((\text{to}}-\text{from})/(\text{now}}-\text{begin}))a\), where \text{“now}” represents the current time, and \text{“begin}” represents the time when the file version history starts.

Due to the properties of time, there are two possible cases for time window: static time-window and sliding time-window.

A static time-window sieve has a \([\text{from}, \text{to}]\) window specified as an absolute time range. The window itself is hence fixed in time. For example, both \text{from} and \text{to} are timestamp literals, e.g., 02-Feb-2003, or 02-Feb-2003/15:30:00.

A sliding time-window sieve uses a now-relative time \([\text{Clin} \, 95]\). A now-relative time is of the format \text{now or now\,interval}, where \text{interval} is some duration of time. The now-relative cut-off point is evaluated to the timestamp when this policy is taking action, thus will be different values at different evaluation time. Hence, the window will slide along the time line as the time advances.
3.1.5 Composite Sieve

As a combined consideration of content and time properties for file versions, a composite policy is designed to have both play a role in vacuuming. Under this scenario, we can specify a time-window or version-window, within which the periodic sieve or percent-different sieve can be applied to the file versions. Hence, there are eight possible combinations: static time(version)-window with periodic sieve, static time(version)-window with percent-different sieve, sliding time(version)-window with periodic sieve, and sliding time(version)-window with percent-different Sieve.

Figure 3-7 gives an example of static time-window sieve combined with periodic sieve. File versions V2, V3, V4, and V5 are inside the time window, so they are imposed under the periodic sieve with frequency 1 and periodicity 2. In the end, V2 and V3 are vacuumed.
In the real world, it is common for people specifying things based on the concept of “week”, “month”, or “year”, etc. We can introduce a third dimension, *time-periodicity*, to represent such a scenario. As long as the history versions are separated into time segments in the unit of *time-periodicity*, we can thus apply each of above vacuuming policies on the basis of segments. For example, to describe the desire of “for each week, vacuuming all versions except the last one”, we can at first specify the *time-periodicity* as “week”, and then specify a sliding version-window sieve of *[begin, current-1]*. Note that, “current” and “begin” are thus evaluated inside the time range of each period segment.

### 3.2 A Formal Model for Vacuuming

This section gives a formal description of vacuuming that has been explored in the previous section. Vacuuming is imposed on the file versions in the archive. It travels through the history information of this file and makes a vacuuming decision. Once some file version is vacuumed from the archive, its status information in the history table will be updated accordingly. We first give the definition of a file version history, then define vacuuming predicate. Since several vacuuming policies can work on one file simultaneously, file versions are vacuumed as long as they conform to any of those policies. We define the vacuuming function in the end to formalize this issue.

**Definition 1** [File Version History]

Let $T = \{t_1, t_2, \ldots, t_n\}$ be a non-empty set of times with $t_1 < t_2 < \ldots < t_n$. Let $T_{\text{now}} = T \cup \{\text{now-relative time}\}$, where “*now-relative time*” is a variable that will be evaluated using the current time. For example, “now-5 days” will be evaluated to the timestamp of
5 days ago. Since $T$ is defined in $<$ ordering, we call the earliest time $t_1$ by $t_{\text{begin}}$ and the latest time $t_n$ by $t_{\text{end}}$.

A file version is a quintuple $(n, d, m, r, s)$ where

$n \in N$ is the sequence number of this file version, where $N$ is the set of natural numbers,

$d \in D = \{1, 2, 3, \ldots, 100\}$ to represent the file difference percentage,

$m \in T$ is the modification time of this file version,

$r \in T$ is the time when this file version was last observed, and

$s \in S$, where $S = \{\text{‘non-existent’}, \text{‘permission-denied’}, \text{‘valid’}, \text{‘vacuumed’}, \text{‘obliterated’}\}$ to represent the status of the file version.

File version history $H$ is a sequence of file versions,

$$(n_1, d_1, m_1, r_1, s_1), \ldots, (n_k, d_k, m_k, r_k, s_k),$$

where the sequence numbers in $H$ are strictly increasing, i.e., $n_i < n_j$ for every $i < j$. We assume that each version in $H$ must have been observed at least once; i.e., $m_i \leq r_i$ for every $i$, and $r_i \leq m_{i+1}$ for $i \leq k-1$.

Let $H$ be fixed, we define $\text{Dom}_{\text{seq}} = \{n_1, n_2, \ldots, n_k\}$, $\text{Dom}_{\text{diff}} = D$, $\text{Dom}_{\text{mod}} = T$, $\text{Dom}_{\text{read}} = T$, $\text{Dom}_{\text{stat}} = S$. Additionally, we define a useful notation, $\text{Dom}_{\text{UV}}^{\text{stat}}$, to be the un-
vaccum subset of $\text{Dom}_\text{stat}$, i.e., $\text{Dom}^{\text{UV}} = \{\text{'non-existent'}, \text{'permission-denied'}, \text{'valid'}\}$.

For any file version $v \in H$, we write $v.n$, $v.d$, $v.m$, $v.r$, and $v.s$ to refer to each attribute of the version.

Viewing that the decision of whether vacuuming a file version or not is based on the attributes of the file version, we define a vacuuming predicate to determine if the file version is to be vacuumed.

**Definition 2** [Vacuuming Predicate]

A vacuuming predicate, $P_{\text{vac}}(v, B, \text{diff}, t_1, t_2)$, is defined as follows:

\[ v.n \in B \land v.d \leq \text{diff} \land v.m \in [t_1, t_2], \]

where $v \in H$, $B \subseteq \text{Dom}_\text{seq}$, $\text{diff} \in \text{Dom}_\text{diff}$, and $t_1, t_2 \in T_\text{now}$. File version $v$ is **vacuumed** if and only if $P_{\text{vac}}(v, B, \text{diff}, t_1, t_2)$ is true.

To exemplify the predicate, we give specific examples below. In our vacuuming policies, we have:

i. Periodic sieve or version-window sieve

Notice that the two policies choose a sub-collection of file version numbers, denoted as $B$, from the entire collection of file versions. The selected versions will be vacuumed. In such case, the vacuuming predicate is $P_{\text{vac}}(v, B, 100, t_{\text{begin}}, t_{\text{end}})$. 
For periodic sieve, it will vacuum the first $x$ versions out of every $y$ versions. Thus,

$$B = \{ n \mid n \in \text{Dom}_{\text{seq}} \land (n \mod y < x) \}$$

For version-window sieve, it will vacuum versions inside the version window $[q_{\text{from}}, q_{\text{to}}]$, where $q_{\text{from}}$ and $q_{\text{to}}$ represents the boundaries of the window. That is, $q_{\text{from}}$, $q_{\text{to}} \in \text{Dom}_{\text{seq}}$. Thus,

$$B = \{ n \mid n \in \text{Dom}_{\text{seq}} \land (q_{\text{from}} \leq n \leq q_{\text{to}}) \}$$

ii. Percent-different sieve

This policy picks a particular difference percentage, denoted as $d$, as the vacuuming criterion. Those versions having less than $d\%$ difference will be vacuumed. In such case, the vacuuming predicate is $P_{\text{vac}}(v, \text{Dom}_{\text{seq}}, d, t_{\text{begin}}, t_{\text{end}})$.

iii. Time-window sieve

Given a time window range, $[t_1, t_2]$, file versions inside the time range will be vacuumed. The vacuuming predicate is $P_{\text{vac}}(v, \text{Dom}_{\text{seq}}, 100, t_1, t_2)$.

iv. Time-window combines periodic sieve

Given a time window range $[t_1, t_2]$ and the chosen sub-collection of file version numbers $B$, the vacuuming predicate is expressed as $P_{\text{vac}}(v, B, 100, t_1, t_2)$. It will vacuum the first $x$ versions out of every $y$ versions, as long as they are inside the time window range $[t_1, t_2]$. Assume the first and last file versions inside the time window range have the sequence number $b$ and $e$ respectively, then

$$B = \{ n \mid n \in \text{Dom}_{\text{seq}} \land b \leq n \land n \leq e \land ((n - b) \mod y < x) \}$$
v. Time-window combines percent-different Sieve

Given a time window range \([t_1, t_2]\) and the chosen file difference ratio \(d\), the vacuuming predicate is expressed as \(P_{\text{vac}}(v, \text{Dom}_{\text{seq}}, d, t_1, t_2)\).

vi. Version-window combines periodic sieve

Given a version window range \([q_{\text{from}}, q_{\text{to}}]\), applying the periodic sieve (vacuum the first \(x\) versions out of every \(y\) versions) to file versions that are inside the window. We have vacuuming predicate \(P_{\text{vac}}(v, B, 100, t_{\text{begin}}, t_{\text{end}})\), where

\[ B = \{ n \mid n \in \text{Dom}_{\text{seq}} \land q_{\text{from}} \leq n \land n \leq q_{\text{to}} \land ((n - q_{\text{from}}) \mod y < x) \} \]

vii. Version-window combines percent-different sieve

Given a version window range \([q_{\text{from}}, q_{\text{to}}]\), applying percent-different sieve (with the difference ratio \(d\)) to file versions that are inside the window. The vacuuming predicate is \(P_{\text{vac}}(v, B, d, t_{\text{begin}}, t_{\text{end}})\), where

\[ B = \{ n \mid n \in \text{Dom}_{\text{seq}} \land (q_{\text{from}} \leq n \leq q_{\text{to}}) \} \]

**Definition 3** [Vacuuming Function]

We denote \(G\) to represent a set of vacuuming predicates defined on \(H\). That is, \(G = \{G_1(v, B_1, d_1, t_{11}, t_{21}), G_2(v, B_2, d_2, t_{12}, t_{22}), G_3(v, B_3, d_3, t_{13}, t_{23}), \ldots\}\), where each \(G_i(v, B_i, d_i, t_{1i}, t_{2i})\) is a vacuuming predicate on \(v \in H\), where \(B_i \subseteq \text{Dom}_{\text{seq}}, d_i \in \text{Dom}_{\text{diff}},\) and \(t_{1i}, t_{2i} \in T_{\text{now}}\) are given constants.

The vacuuming function on a file version, \(f_{\text{vac}} : H \times \{G\} \rightarrow H\), is defined as follows:
Thus the vacuuming on a file version history can be denoted as:

\[
\text{Vac}(H, G) = \{ v' \mid \text{for some } v \in H, f_{\text{vac}}(v, G) = v' \}
\]

### 3.3 Coalesce

With vacuuming introduced and imposed, a file version history no longer consists only of existing file versions since it may contain vacuumed versions. Each vacuumed version has a tuple in the history table. Chances are that there may exist a sequence of vacuumed versions in continuity. For instance, after applying a version-window sieve with window \([0, \text{current}-20]\), the version history is split into two parts, with the first part being a sequence of vacuumed versions and the second part being a sequence of un-vacuumed versions.

The example in Figure 3-8 shows the querying for vacuumed file versions V5 and V6. We find that the two queries are equivalent in the version history context, since both V5 and V6 share the same surrounding un-vacuumed versions. As to V5, the previous un-vacuumed version is V4, and the following un-vacuumed version is V7. File version V6 has the same properties as V5.
Coalesce merges adjacent vacuumed versions. As a result, one obliterated version replaces a sequence of vacuumed versions. The obliterated version extends in time from the first vacuumed version in the sequence to the last vacuumed version in the sequence. Recall that a file version consists of a known time and an assumed time, so does the obliterated version. Its known time begins at the start of the known time interval of the first vacuumed version, but ends at the end of the known time interval of the last vacuumed version. Its assumed time interval is duplicated from the last vacuumed version. It also replicates the version number from the last vacuumed version, for the reason that it can keep track of the used version numbers. Figure 3-9 shows an example. Vacuumed file versions V1 and V2 are replaced by obliterated file version V2'. V2' has the same assumed time interval as V2. Vacuumed file versions V5 and V6 are replaced by V6' as well. The known time interval of V6' expands till the assumed time of V6, although V5 has an assumed time interval.
To offer flexible control, a time window [from, to] is allowed to restrict the range of coalesce. Only file versions inside the time window are affected by coalesce. For the example above, the time window is deemed as [begin, now].

### 3.4 A Formal Model for Coalesce

This section constructs a formal model for coalesce. We will use the symbols and definitions in Section 3.2.

**Definition 4** [Coalesce function]

The coalesce function on a file version, $f_{coa}$, is defined as follows.

$$f_{coa} : H \times T_{now} \times T_{now} \rightarrow H$$
Let \( f_{\text{coa}}(v, t_1, t_2) = v' = (n', \, d', \, m', \, r', \, s') \), where \( v, \, v' \in H, \, t_1, \, t_2 \in T_{\text{now}} \). Let sequence \( H = v_1, \, v_2, \ldots, \, v_k \). Define two partial functions \( f_{\text{pre}}, \, f_{\text{next}} \) to denote the previous and next version for one particular version. That is, for each \( i \geq 2, \, f_{\text{pre}}(v_i) = v_{i-1}, \) and \( f_{\text{pre}}(v_1) \) is undefined. For each \( i \leq k-1, \, f_{\text{next}}(v_i) = v_{i+1}, \) and \( f_{\text{next}}(v_k) \) is undefined.

Now, we can define the coalesce function in detail as follows:

1) If \( v.m \not\in [t_1, \, t_2], f_{\text{coa}}(v, t_1, t_2) = v \)

2) Otherwise, let

\[
\begin{align*}
n' & = \begin{cases} 
n. \, n, \text{if } v.s \in \text{Dom}_{\text{stat}}^{\text{UV}}, \text{or } f_{\text{next}}(v) \text{ is undefined, or } f_{\text{next}}(v).s \in \text{Dom}_{\text{stat}}^{\text{UV}} \\
n. \, n, \text{otherwise, where } w = f_{\text{coa}}(f_{\text{next}}(v), t_1, t_2) \end{cases} \\
d' & = \begin{cases} 
d. \, d, \text{if } v.s \in \text{Dom}_{\text{stat}}^{\text{UV}} \\
00, \text{otherwise} \end{cases} \\
m' & = \begin{cases} 
m. \, m, \text{if } v.s \in \text{Dom}_{\text{stat}}^{\text{UV}}, \text{or } f_{\text{pre}}(v) \text{ is undefined, or } f_{\text{pre}}(v).s \in \text{Dom}_{\text{stat}}^{\text{UV}} \\
m. \, m, \text{otherwise, where } w = f_{\text{coa}}(f_{\text{pre}}(v), t_1, t_2) \end{cases} \\
r' & = \begin{cases} 
r. \, r, \text{if } v.s \in \text{Dom}_{\text{stat}}^{\text{UV}}, \text{or } f_{\text{next}}(v) \text{ is undefined, or } f_{\text{next}}(v).s \in \text{Dom}_{\text{stat}}^{\text{UV}} \\
r. \, r, \text{otherwise, where } w = f_{\text{coa}}(f_{\text{next}}(v), t_1, t_2) \end{cases} \\
s' & = \begin{cases} 
s. \, s, \text{if } v.s \in \text{Dom}_{\text{stat}}^{\text{UV}} \\
'\text{obliterated}', \text{otherwise} \end{cases}
\end{align*}
\]

Thus the coalesce on a file version history can be denoted as:

\[
\text{Coa}(H, \, t_1, \, t_2) = \{ \, v' \mid \text{for some } v \in H, \, f_{\text{coa}}(v, \, t_1, \, t_2) = v' \ \}
\]

3.5 Obliteration

Seeing that coalesce eliminates sequences of vacuumed tuples from the file version history, we can carry out one-time obliteration to eliminate all the tuples in a file version
history. In an e-shop scenario, it is general that a merchant discontinues offering one product, for the reason that the producing branch is cut off. The web master, hence, thinks of the history information of the web page for that product is no longer useful and would like to throw it away. Obliteration helps him to achieve this goal.

Obliteration removes all the file versions from the archive, and shrinks the version history to a single one – an obliterated version. Figure 3-10 shows an example of obliteration. The obliterated version (V7′) possesses the time interval of the entire history, from the first file version (V1) to the current file version (V7).

![Figure 3-10 Obliteration Example](image)

Similar to coalesce, the time window property is also allowed to accompany the obliteration. Only file versions inside the time window are obliterated.
3.6 A Formal Model for Obliteration

This section constructs a formal model for obliteration. We will use the symbols and definitions in Section 3.2 and Section 3.4.

**Definition 5** [Obliterate function]

The obliterate function on a file version, \( f_{obl} : H \times T_{now} \times T_{now} \rightarrow H \) is defined as follows:

\[
f_{obl}(v, t_1, t_2) = f_{coa}(f_{vac}(v, G), t_1, t_2), \text{ where } G = \{ P_{vac}(v, Dom_{seq}, 100, t_1, t_2) \}.
\]

It means vacuum file versions in the time window range first, and then coalesce them.

Similarly, the obliteration on a file version history can be denoted as:

\[
Obl(H, t_1, t_2) = \{ v' | \text{ for some } v \in H, f_{obl}(v, t_1, t_2) = v' \} \text{ or } Obl(H, t_1, t_2) = Coa(Vac(H, G), t_1, t_2), \text{ where } G = \{ P_{vac}(v, Dom_{seq}, 0, t_1, t_2) \}
\]

3.7 Repairing Queries for Vacuumed Documents

The effect of vacuuming is reflected when querying the vacuumed version. Once some document versions are vacuumed, the archive no longer holds a complete and perfect status of the past. Queries that request an un-vacuumed version will not see the influence of vacuuming. But queries that request a vacuumed version will encounter an HTTP_NOT_FOUND (404) error.
Here, we design a simple and feasible functionality – query repair strategy – to improve the interaction between online users and the web server, by offering a flexible but affordable response to online users. Additional explanatory information (secondary result) may accompany the query result (primary result). Only requests which get the same query results when they are submitted to the vacuuming-enhanced transaction-time web server as when submitted to the normal transaction-time web server, will not be answered with additional warning/notification information. Several strategies could be used when responding to requests that will be affected by vacuuming. The following sections describe these strategies in detail.

3.7.1 A warning that shows the requested version has been vacuumed.

Compared with the HTTP_NOT_FOUND reply, this repair technique provides more information to guide the user to the next query operation. It generates a result page, notifying the requested version is vacuumed, along with recommended links for querying un-vacuumed versions. In the case of an interactive online user, this approach normally will lead him to adjust or correct his request to fetch an un-vacuumed version. One more round of request processing will be involved for the web server. The general cost is a linear increase, with the worst case being double.

3.7.2 A warning that shows this request is affected by vacuuming, followed by a repairing query result.

Notice that file versions are organized in time order during its lifetime, and the nearby versions are more similar than those versions far away in time line. The time-close-similar properties of file versions give us a hint on designing the repairing mechanism.
One method is to find the closest available version younger than the requested, vacuumed version as a replacement, called the *redirect to next version repair*. The technique is to search forward along the timeline, from the time of the desired vacuumed version till an un-vacuumed is found. Recall that our vacuuming rule “the current file version is never vacuumed” ensures that the un-vacuumed version will always be available under this technique.

An alternative method, in contrast to the above, is to find the closest available version older than the requested vacuumed version. We call this the *redirect to previous version repair* strategy. This is similar to the previous strategy, except the searching is along the reverse direction of the timeline. If no such version exists, a 404 error is generated.

The cost for redirection depends on how it will be implemented in the application. It can be an automatic internal redirection carried out by the web server, or it can be an integrated processing when the web server is processing the original user request. Either way, the cost is linear in the number of version tuples in the history table.

Figure 3-11 shows four versions of “*sports.html*”: V1, V2, V3, and V4. Assume V1 and V3 are vacuumed. When a user requests V3, V2 will be returned under the previous-version-redirect repairing approach, while V4 will be returned under the next-version-redirect repairing approach.
3.7.3 No explanatory result information is given.

No friendly response is given to the user under this approach. Users only get a notification, saying this document is not found if the result happens to have been vacuumed. It eases the web server from internal burden for the first time, but as long as the user will issue another request after seeing this message and changing part of the request, the web server still has to deal with the second, maybe more incoming requests from the same user.

3.8 A Formal Model for Query Repair

Query repairing is imposed when the required version is vacuumed or obliterated. The resulting version history differs from the original history. To construct the formal model of query repairing, we first define the query repair function with redirection, and then give the definition for resource version, stored resource version history, and viewed resource version.

**Definition 6** [Repairing function]
To define repairing function, we need to denote a repairing type, so let $\text{Dom}_{\text{rep}} = \{\text{‘past’, ‘future’}\}$.

We define $f_{\text{end}} : H \rightarrow T_{\text{now}}$ to compute the end point time of each file version as

$$f_{\text{end}} (v) = \begin{cases} (f_{\text{next}} (v), m) - 1, & \text{if } f_{\text{next}} (v) \text{ is defined} \\ \text{now}, & \text{otherwise} \end{cases}$$

Repairing function, $f_{\text{rep}}$, is defined as a mapping from $H \times T_{\text{now}} \times \text{Dom}_{\text{rep}}$ to $H$:

$$f_{\text{rep}} : H \times T_{\text{now}} \times \text{Dom}_{\text{rep}} \rightarrow H$$

Assume $f_{\text{rep}} (v, t, p) = v'$, where $v, v' \in H$, $t \in T_{\text{now}}$, and $p \in \text{Dom}_{\text{rep}}$. Now, we can define the repairing function in detail as follows:

1) If $t \in [v.m, f_{\text{end}} (v)]$ and $v.s \in \text{Dom}_{\text{stat}}^{UV}$, then $v' = v$.

2) Otherwise, we have

- If $p = \text{‘past’}$, then $v' = \begin{cases} f_{\text{rep}} (f_{\text{prec}} (v), t, p), & \text{if } f_{\text{prec}} (v) \text{ is defined and } v.m < t \\ \text{undefined, otherwise} \end{cases}$

- If $p = \text{‘future’}$, then $v' = \begin{cases} f_{\text{rep}} (f_{\text{next}} (v), t, p), & \text{if } f_{\text{next}} (v) \text{ is defined and } f_{\text{end}} (v) > t \\ \text{undefined, otherwise} \end{cases}$

**Definition 7** [Stored Resource Version]

A resource consists of one main file and several included files. A resource version is created by a change in either the main file or any of the included files. Thus, resource versions are constructed from the file versions of the main file and its included files. We call it a *stored resource version*, indicating the resource version in the original scenario.
Assume a resource has a main file $M$. Denote the file version history of $M$ as $H(M)$. For each version of $M$, let $m_i$ be the modification time of the version, $s_i$ be the file version status, and $F_i$ be the set of included files. The file version history for any included file $f$ is written as $H(f)$. $T_{\text{mod}}$ 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. That is,

$$T_{\text{mod}} = \{ t | \exists v. t = v.m \land (v \in H(M) \lor \exists v_i \in H(M), f \in F_i \text{ such that } v.m \in (v_i.m, f_{\text{end}}(v_i)) \}$$

Let

$$Sort(T_{\text{mod}}) = (t_1, t_2, \ldots, t_j)$$

be a sorted list of times such that $t_i < t_{i+1}$, for every $1 \leq i \leq j-1$.

We denote resource version $u_i$ of the resource as $u_i = (m_i, r_i, s_i, F_i)$, which is constructed as follows.

- $m_i = t_i$,
- $F_i$ is the set of included files in $M$ as of time $m_i$,
- $r_i = \min( v.r )$, for all $v \in H(f)$, with $f \in F_i$ and $v.r \in [t_i, t_{i+1})$,
- $s_i$ is computed in the following steps:
  1. $s_i = v.s$, for some $v$ such that $v.m \in [t_i, t_{i+1}]$ and $v.s \notin Dom_{\text{stat}}^{UV}$, where $v \in H(M)$.

Notice that only one $v$ in $H(M)$ can satisfy $v.m \in [t_i, t_{i+1}]$. 
2. \( s_i = \text{'obliterated'} \), if the condition in 1 above is not true, and if there exists \( f \in F_i \) and \( v \in H(f) \), such that \( m_i \in [v.m, f_{end}(v)] \) and \( v.s = \text{'obliterated'} \).

3. \( s_i = \text{'vacuumed'} \), if the conditions in 1 and 2 above are not true, and if there exists \( f \in F_i \) and \( v \in H(f) \), such that \( m_i \in [v.m, f_{end}(v)] \) and \( v.s = \text{'vacuumed'} \).

4. \( s_i = v.s \), if the conditions in 1, 2, and 3 above are not true, and for some \( v \in H(M) \) and \( v.m \in [t_i, t_{i+1}) \).

\( s_i \) is computed using the following rules.

- If the main file version at time of \( m_i \) is a ‘vacuumed’ or ‘obliterated’ version, the resource version is regarded as the same status of the main file version. Because of the building of \( T_{mod} \), it’s guaranteed that only one file version for the main file could be inside \([t_i, t_{i+1}]\).

- Otherwise, if any one of the included files at time of \( m_i \) is an ‘obliterated’ version, the resource version is regarded as ‘obliterated’.

- Otherwise, if any one of the included files at time of \( m_i \) is a ‘vacuumed’ version, the resource version is regarded as ‘vacuumed’.

- Otherwise, the main file and included files are un-vacuumed, the resource has the same status of the main file.

**Definition 8** [Stored Resource Version History]

The stored resource version history is an ordered sequence of resource version, \[ \{(m_1, r_1, s_1, F_1), (m_2, r_2, s_2, F_2), \ldots, (m_k, r_k, s_k, F_k)\} \], denoted as \( R \). It indicates the history of resource in the original scenario.
With the specified time $t \in T_{\text{now}}$, we can find the corresponding stored resource version from the stored resource version history:

$$f_{\text{store}} : \{R\} \times T_{\text{now}} \rightarrow R$$

$$f_{\text{store}}(R, t) = \begin{cases} (m_i, r_i, s_i, F_i), & \text{if } \exists i \text{ such that } (m_i, r_i, s_i, F_i) \in R, \text{ and } t \in [m_i, m_{i+1}) \\ \text{undefined, otherwise} \end{cases}$$

**Definition 9 [Viewed Resource Version]**

The resource for a time-slice query could differ from the original one, since the main file version or some included file version could be repaired. Here, we define a *viewed resource version* to describe such scenario.

For a specified time $t \in T_{\text{now}}$ and repairing option $p \in \text{Dom}_{\text{rep}}$, the viewed resource version consists of a set of file versions, one and only one of which is for either a main file or an included file of the resource. We denote it as $\{w_m, w_i,1, w_i,2, \ldots, w_i,n\}$, with $w_m$ stands for the file version of main file $M$, $w_i,j$ stands for the file version of included file $f_j$. Assume the resource has $n$ included files, written as $f_j, j = 1, 2, \ldots, n$. The viewed resource version can be constructed in the following steps.

1. Compute the stored resource version history $R$,
2. Find the specified stored resource version $f_{\text{store}}(R, t) = (m_i, r_i, s_i, F_i)$,
3. $w_m = f_{\text{rep}}(v, t, p)$, where $v \in H(M)$, with $m_i \in [v, m, f_{\text{end}}(v)]$,
4. $w_{i,j} = f_{\text{rep}}(v_j, t, p)$, where $v_j \in H(f_j)$, with $m_i \in [v_j, m, f_{\text{end}}(v_j)]$.

Finally, we can denote the construction as the following function.


\[ F : \{ R \} \times T_{\text{now}} \times Dom_{\text{rep}} \rightarrow H(M) \times H(f_1) \times H(f_2) \times \ldots \times H(f_n) \]

### 3.9 Syntax

Apache v2.0 adds vacuuming functionality to Apache v1.0. Resource owners can specify the vacuuming policies for files and directories. Vacuuming affects the queries since the queries may ask for vacuumed versions. Resource owners can also specify the repairing strategy for the vacuumed versions to improve query responsiveness. Moreover, \( \pi \)Query is enhanced by allowing queries on different file version categories. Resource owners can also reorganize the web architecture by migrating resource objects to new location.

The BNF for vacuuming related functionalities, such as vacuuming policies, vacuuming settings, and obliteration, as well as forwarding functionality is given in Table 3-3. Table 3-4 briefly describes each vacuuming policy together with their arguments. Table 3-5 illustrates the option terms for repair setting, include parent setting, and obliteration.
<vacuum query> ::= vacuum( <vacuum option> | <vacuum setting option> |
| <vacuum remove option> )
<vacuum option> ::= <version-window vacuum> | <time-window vacuum>
| <periodic vacuum> | <diff-sieve vacuum>

<vacuum remove option> ::= remove

<coalesce query> ::= coalesce[ (recursive) ][ (from time), (to time) ]
<obliterate query> ::= obliterate[ (recursive) ][ (from time), (to time) ]

<forward query> ::= forward=<file URL path>[ (from time), (to time) ]
<file URL path> ::= <directory path> /<file name>
<directory path> ::= ε | /<directory name>[ /<directory path>]
<file name> ::= file name
<directory name> ::= directory name

Table 3-3 BNF for Vacuuming
### Vacuum Operation

<table>
<thead>
<tr>
<th>Periodicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency: (a)</td>
<td>For versions inside the time window ([ft, tt]), vacuum the first (a) versions among every group of (b) versions. So, we have (a \leq b).</td>
</tr>
<tr>
<td>periodicity: (b)</td>
<td></td>
</tr>
<tr>
<td>from time: (ft)</td>
<td></td>
</tr>
<tr>
<td>to time: (tt)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Percent-Different</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>percentage: (d)</td>
<td>For versions inside the time window ([ft, tt]), vacuum versions less than (d)% different from previous version.</td>
</tr>
<tr>
<td>from time: (ft)</td>
<td></td>
</tr>
<tr>
<td>to time: (tt)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version-Window</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start version: (sv)</td>
<td>Vacuum versions with the version numbers inside window ([sv, ev]).</td>
</tr>
<tr>
<td>end version: (ev)</td>
<td></td>
</tr>
<tr>
<td>from time: (ft)</td>
<td></td>
</tr>
<tr>
<td>to time: (tt)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time-Window</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from time: (ft)</td>
<td>Vacuum versions inside the time window ([ft, tt]).</td>
</tr>
<tr>
<td>to time: (tt)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Remove</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Remove all vacuuming strategies.</td>
</tr>
</tbody>
</table>

**Table 3-4 Notation for Vacuuming Policies**

<table>
<thead>
<tr>
<th>Repair Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No repairing is applied. Default option. Return 404 error.</td>
</tr>
<tr>
<td>Alert</td>
<td>When version is not found, give alert message.</td>
</tr>
<tr>
<td>Past</td>
<td>When version is not found, use the latest version earlier than the requested one as replacement.</td>
</tr>
<tr>
<td>Future</td>
<td>When version is not found, use the earliest version later than the requested one as replacement.</td>
</tr>
<tr>
<td>from time: (ft)</td>
<td>Repairing is workable for versions inside the time range ([ft, tt]).</td>
</tr>
<tr>
<td>to time: (tt)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Include Parent Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>The vacuuming policies specified on parent directory should also be applied to the current entry. If there is no vacuuming policy specified on the current entry, the search will automatically be directed to its parent directory.</td>
</tr>
<tr>
<td>No</td>
<td>The vacuuming policies specified on the parent directory need not be applied to the current entry. It is the default value when there is a strategy specified on the current entry.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coalesce/Obliteration Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive</td>
<td>If the obliteration/coalesce is designated on a directory, applying it on its children directories and files inside as well.</td>
</tr>
<tr>
<td>from time: (ft) to time: (tt)</td>
<td>Applying obliteration/coalesce on versions inside time range ([ft, tt]).</td>
</tr>
</tbody>
</table>

**Table 3-5 Notation for Option Terms**
The BNF for enhanced \texttt{TT}Query is given in Table 3-6. And its options for version category are described in Table 3-7.

\begin{verbatim}
<tt query> ::= ε | <qtt> [ , <ptt> ] [ , <repair setting> ] [ [ <version range> ] ]
<version range> ::= <one range> [ , <one range> ]
<one range> ::= k | a | v | o
\end{verbatim}

<table>
<thead>
<tr>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>Non-vacuumed/obliterated versions that have known time.</td>
</tr>
<tr>
<td>a</td>
<td>Non-vacuumed/obliterated versions that have assumed time.</td>
</tr>
<tr>
<td>v</td>
<td>Vacuumed versions.</td>
</tr>
<tr>
<td>o</td>
<td>Obliterated/Coalesced versions.</td>
</tr>
</tbody>
</table>

Table 3-6 BNF for Enhanced \texttt{TT}Query

Table 3-7 Notation for Version Category
4 Implementing Vacuuming in \texttt{Apache}

\texttt{Apache} v1.0 is implemented as an extension of the Apache v1.3 HTTP server. It directly hacks the source code of Apache v1.3 to implement the transaction-time functionality. Recently, Apache v2.0 was released. The Apache Software Foundation spent nearly three years working on Apache v2.0. Their goal was to make it the best HTTP server on the planet. It is substantially different (internally) from Apache v1.x. We describe the implementation of the transaction-time and vacuuming functionality in Apache v2.0 in this chapter. Hence, we name the web server \texttt{vTT} Apache v2.0, meaning “vacuuming-enhanced transaction-time web server on Apache v2.0”. The porting work is described in Section 4.1. Data representation concerns are stated in Section 4.2. An overview of the overall system implementation is given in Section 4.3.

4.1 Port to Apache v2.0

In this section, we focus on describing the porting of Apache v1.3 to Apache v2.0. The architecture of those two versions varies greatly. The changes influence our design, the implementation of \texttt{TT} Apache and the performance. We explain the primary differences between the two versions of Apache that concern this project in Section 4.1.1. Then, we state how the upgrading work is carried out as to our system in Section 4.1.2.

4.1.1 New Features of Apache v2.0

There are several new features in Apache v2.0. The largest one is the creation of an abstract layer: the Apache Portable Runtime (APR) library. APR encapsulates the details of the underlying operating system by offering a standard, uniform programming interface to the server application. The APR library eases the programmer from platform-
specific file I/O, network I/O, process and thread management, memory management, mutual exclusion, and synchronization; it ensures that the program can be written once, complied and run everywhere. A further advantage of APR is that the code becomes much more manageable and maintainable. Each APR implementation uses the native API and data structures of the underlying operating system, so the performance won’t be affected when developing application using APR. Figure 4-1 shows how the APR works in a general view.

![Figure 4-1 Overview of the APR Layer](image)

A second notable feature in Apache v2.0 is the concept of a filter. In some sense, the Apache v2.0 architecture could be looked upon as a filtered, or layered, I/O system. A filter makes it easier for one module to modify the output data that was generated by an earlier module. Web page content can be viewed as a stream of information. In Apache filter terminology, the information is composed of a sequence of chunks, each chunk is stored in a bucket, and lists of buckets form brigades. Lists of brigades can create a web document, as shown in Figure 4-2. Filters operate on one brigade at a time, and are called upon repeatedly until the entire document has been processed. In general, document data flows through the filter chain before being sent back to the client browser along the network connection. Since the data are split into buckets, filters can work concurrently,
thus improving the performance of processing the request. Figure 4-3 illustrates the role of filters in Apache v2.0. Incoming HTTP request passes through a chain of input filters and is then processed by input processor. After other internal processing, the content generator creates the output data that will go through a chain of output filters before they are finally sent back to the client.

![Diagram](image)

Figure 4-2 Bucket-Brigade View of a Document

One other change that should be taken into consideration when porting Apache is the reorganization of its module structure. The whole Apache v2.0 package is highly modularized, with only one outside server framework as the main entry. Each individual module takes part in the duration of processing each HTTP request. Apache v2.0 has a
special API for easily writing extension module. The API is mainly used for communication directed from modules to Apache core. The Apache API consists of a predefined set of functions that modules can call, for example, to access Apache data structures. That way, modules can access server or connection specific configuration data, and the module specific data as well. As we can see in Figure 4-4, modules and core interact with each other. Apache breaks down request handling into a series phases. A module contains several event handlers, each of which will join a phase to handle the request. Server core looks at each of a succession of modules to see if each of them has a handler for the phase, and attempts invoking it if so. Handlers themselves are functions.

![Figure 4-4 Core and Modules in Apache v2.0](image)

Besides those, there are other new features like improved performance on non-Unix platform, hybrid thread/process modes for speed and greater scalability, multi-processing modules (MPM), etc. Details can be obtained from the Apache website [Apa03].

### 4.1.2 Upgrade

Before working on upgrading T7Apache to use Apache v2.0, we had to clarify several tasks. First, T7Apache v1.0 hacked the Apache core functionality to add
evaluation of transaction-time queries and lazy archiving; that is, the implementation is
tightly coupled with the Apache core source code. The drawback of this approach is
apparent. Every tiny modification related to the transaction-time processing has to be
done by recompiling and re-launching the whole Apache package, which is less than
ideal from a software development cycle perspective. Second, the transaction-time
support has to be reprogrammed into each new release of Apache.

Viewing the modular feature of Apache v2.0, the attempt here is to encapsulate the
transaction-time facilities into an individual module, an extension to the server. This
module is built as a separate library, loaded into the Apache core dynamically. By
registering the event handler with the core, each incoming transaction-time query can be
captured and processed inside the module code. Apache source remains unchanged. The
developing, compiling, and launching of the module is entirely separate from the Apache
source distribution, which can shorten the developing/testing cycle of the task. Table 4-1
lists all the event handlers used in the implementation.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Event Handler</th>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create server configuration</td>
<td>create_server_config</td>
<td>Read database settings from configuration file. Initialize shared memory.</td>
</tr>
<tr>
<td>2. Server initialization</td>
<td>tt_post_config</td>
<td>Open database.</td>
</tr>
<tr>
<td>Child process initialization</td>
<td>tt_child_init</td>
<td>Attach to the shared memory. Register tt_child_exit.</td>
</tr>
<tr>
<td>3. Last chance to modify things before content generator</td>
<td>tt_fixupper</td>
<td>Add output filter function tt_filter to the filter chain in server.</td>
</tr>
<tr>
<td>4. Content generator</td>
<td>tt_handler</td>
<td>Transaction-time functionality.</td>
</tr>
<tr>
<td>5. Child process exit</td>
<td>tt_child_exit</td>
<td>Detach from the shared memory. The last child should close database and destroy the shared memory.</td>
</tr>
</tbody>
</table>

Table 4-1 Event Handlers for TTApache
Transaction-time queries contain a restructuring part, which supports rewriting of the
hyperlinks inside the requested document to be the version of a desired time slice. In
Apache v1.0, the rewriting task is done by parsing the document, locating the
hyperlinks, appending the restructuring part, and sending this modified file back.

In Apache v2.0 we can use an output filter for restructuring. Modification is
applied by iterating the bucket brigade list. When encountering the hyperlinks, the
restructuring part can then be inserted into the bucket at the right position.

Both the module structure and filter structure are required now; hence the final
transaction-time module is designed to be a combination of an HTTP request handler (by
registering event handlers) and output filter (by registering filter function).

4.2 Data Representation

Historical versions of all files are stored in a large data collection – the archive. We
use a history table for associating each document with their own history versions. We
focus on the changes in the history table for vacuuming in Section 4.2.1. In Section 4.2.2,
we describe a new database schema to keep track of vacuumed versions.

4.2.1 Various Version States

![Figure 4-5 History Schema](image)

Figure 4-5 shows the schema of the history table. Each record is a pair of key and
value. A file is identified by a file path, which is stored in the key field. Each file path is
associated with many version records that are stored in the value field. Each version record has four fields. The version records are stored in temporal order from the latest to the earliest.

In TT Apache v1.0, only the known and assumed file versions are recorded, with the “fv” referring to an existing archived file version. In vTT Apache v2.0, there are more possible file statuses. Table 4-2 shows the possible statuses of a file version and their meanings.

<table>
<thead>
<tr>
<th>Version Status</th>
<th>Meaning</th>
<th>Refer to a file in archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid version</td>
<td>The same meaning as in TT Apache v1.0.</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-existent version</td>
<td>The version is not at the location identified by the file path. This nonexistence can be observed by the web server during an HTTP request.</td>
<td>No</td>
</tr>
<tr>
<td>Permission-denied version</td>
<td>The version is not allowed read access.</td>
<td>No</td>
</tr>
<tr>
<td>Vacuumed version</td>
<td>The file version is vacuumed.</td>
<td>No</td>
</tr>
<tr>
<td>Obliterated version</td>
<td>The file version is obliterated or coalesced.</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4-2 File Version Status

The additional functionality in vTT Apache v2.0, such as vacuuming and obliteration will lead to file version status change. Figure 4-6 depicts the possible state transitions.

4.2.2 Policy Table

A policy table stores the vacuuming settings for target objects, which could be a file or a directory. Similar to the history table, each record in policy table is a pair of key and value. The absolute pathname of the file or directory works as the key. The value field is the combination of the repairing strategy, the include parent setting, and a sequence of vacuuming policies. Figure 4-7 depicts the policy table relation schema.
The vacuuming policies are organized as a configuration tree structure. If the includeparent setting is enabled, the parent directory’s vacuuming settings are also effective on this entry. We give an example to clarify this statement. Consider the directory tree structure shown in Figure 4-8 and their vacuuming policies on each node in the tree.
Each triple such as \([\text{past, yes, -}]\) represents the repairing strategy, includeparent setting, and vacuuming policy on that entry. For instance, the root directory node \("/Home"\) has an “alert” repairing strategy, a vacuuming policy of “\((\text{periodic},2,5)\)”, and the includeparent option enabled; while the child directory node \("/Home/index.html"\) has “future” repairing strategy, no vacuuming policy (“-” specified), and the includeparent option disabled.

The vacuuming policies that are applicable to each HTML file in the tree are given Table 4-3.

<table>
<thead>
<tr>
<th>File node</th>
<th>Working policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Home/index.html</td>
<td>-</td>
</tr>
<tr>
<td>/Home/customers/info.html</td>
<td>(t-window, begin+6, now), (v-window, 2, current-10),</td>
</tr>
<tr>
<td></td>
<td>(periodic, 2, 5)</td>
</tr>
<tr>
<td>/Home/customers/order.html</td>
<td>(earliest-diff, 30)</td>
</tr>
<tr>
<td>/Home/merchants/trade.html</td>
<td>(periodic, 3, 7)</td>
</tr>
</tbody>
</table>

Table 4-3 Policies for Each File

The “yes” option on the leftmost branch leads to the “\((\text{periodic},2,5)\)” setting on the root node propagates till the leaf node, “/home/customer/file1.html”. In
contrast, the “no” option on “/Home/merchants” prevents the “(periodic,2,5)” affecting its child leaf node, “Home/merchants/trade.html”.

4.3 System Implementation

Implementation details are explored in this section. We briefly describe the implementation design for vacuuming and repairing in Section 4.3.1. Then, we describe the overall system architecture in Section 4.3.2.

4.3.1 Implementation Design

Implementing vacuuming involves two issues. One is to impose vacuuming policies on the archive combined with coalesce or obliteration on the history table. The other is to implement the repairing strategy for queries on vacuumed versions.

We choose the per-request implementation approach for vacuuming. That is, the web server lazily carries out vacuuming. For each incoming HTTP request, the web server looks at the history information of the requested file and then applies all vacuuming policies that are present. It can be achieved when we are making backup decision of the requested file. As an alternative approach, the web server can eagerly vacuum versions upon the vacuuming request by the resource owners or web master. In such case, the vacuuming is performed on demand.

Repairing works as an auxiliary procedure for processing HTTP requests. Here we focus on the implementation of redirect repair. After processing a πQuery, the task control is switched to repairing. If a requested file version has been vacuumed, repairing will fetch a non-vacuumed version as a replacement.
4.3.2 System Architecture

Figure 4-9 gives an architectural view of the system functionalities. The “Apache Server Core” and “Standard Modules” belongs to the released Apache v2.0 source package. Our vacuuming-enhanced Apache functionalities are encapsulated in an “Extension Module” hereby being an intact source module and seamlessly integrated to the original Apache server. The extended functionalities are hidden from the online users. The Apache server takes the responsibility of accepting HTTP request, dispatching task to modules, getting a result and sending back to online users.

In the case that an HTTP request is a Query, the “Query Handler” in the “Extension Module” will be invoked to work on the request. It checks the file version being requested and triggers the “Vacuum-enhanced Backup Handler” if necessary, which combines both backup and vacuuming together thus fulfill the lazy backup and lazy vacuuming simultaneously.

If the issued HTTP request aims to manage the resource, the “Apache Server Core” will invoke the “Resource Manage Handler” to perform the respective tasks. If the request involves vacuuming settings, the “Vacuum Handler” will be triggered to carry out the eagerly vacuuming operation. Details of each handler are described in the following sections.
Vacuuming-enhanced TTApache Model

Figure 4-9 System Architecture Figure
4.3.3 Handler Description

When processing a TTQuery, the TTQuery Handler first builds a resource version list for the requested document. As to TTQueries indicating the constraint of version state category, only resource versions matching the constraint are fetched. During this stage, the main file or included file for the resource will be requested as a TTQuery, which invokes the Vacuum-enhanced Backup Handler. Then, the handler walks through the version list to locate the requested version. If a vacuumed or obliterated version is found and there exists repairing strategy, the handler tries to perform repairing action and then returns the result to the user.

For the requested file, the Vacuum-enhanced Backup Handler first constructs a file version history list from the current version to the earliest one. If a new file version is observed while not been recorded, it will be added to the front of the list and the new file will be backup to the archive. Next, this handler traverses the policy table to build a list of vacuuming policies that are applicable on the requested file. For each vacuuming policy, the handler will invoke the corresponding type of Vacuum Handler to carry out the policy specific vacuuming operation on the history list. The algorithm for the second step is shown in Appendix 7-1. Finally, the handler updates the history information of this file in the history table.

Vacuum Handler realizes what we have designed in Section 3.1. It classifies different vacuuming policies and performs the tasks.

Resource Management Handler processes request for coalesce or obliteration, forwarding history, setting vacuuming options, and removing vacuuming policies.
Therefore, it first differentiates those requests and then dispatches action for those requests. For the request of setting repairing (includeparent) option, it will update the value field of repairing (includeparent) in the policy table for the target object, a file or a directory. For the request of removing vacuuming policies, the handler will truncate policy records for the target object. For the forwarding history request, the handler fetches a piece of the version history conforming the time window constraint, and inserts a new history entry in the history table by replicates that piece to be the data value for the forwarded path. As to the coalesce or obliteration request, this handler will apply coalesce or obliteration for file versions conforming the specified time window. The algorithms are shown in Appendix 7-3 and Appendix 7-4 respectively. In the directory case, the handler will iterate all entries below the directory and perform coalesce or obliteration in turn. The algorithm is shown in Appendix 7-2. For the one time request of setting vacuuming policy, this handler can trigger the Vacuum Handler immediately. The final step for Resource Management Handler is to compose a response page for the user.
5 Experiments

The implementation of \( \nuT\) Apache v2.0 involves works of two phases. One is to port the \( \mu\) Apache v1.0 onto Apache v2.0 platform; the other is to realize the vacuuming functionality. Hence, we carry out a series of experiments to measure the overhead and performance of our web server in each phase. We design our experiments in Section 5.1. We compare the Apache v2.0 and \( \mu\) Apache v2.0 in Section 5.2, which follows [Lin02] as the samples. In Section 5.3, we compare \( \mu\) Apache v2.0 and \( \nu\mu\) Apache v2.0 and analyze the results.

5.1 Experiment Design

We perform experiments to measure the overhead imposed by the additional functionalities. The factors to be measured are discussed in Section 5.1.1. Details of the experiments are described in Section 5.1.2.

5.1.1 Measured Factors

The HTTP request-response time (turnaround time) is the primary concern for online users. We measure the turnaround time in the experiments to analyze the effect imposed by the transaction-time functionality. Second, we gather statistics of the system resource utilization: memory usage, disk space consumption, and CPU cost. Memory is used by file cache system, virtual memory, kernel, and processes. Cache stores the most recently accessed files in memory for later usage, so as to save the time cost of disk I/O operation. Hence, the turnaround time can be reduced. Virtual memory works to swap out less used part of memory to the disk, so as to optimize the usage of memory. Swapping operation is time-consuming and will thus increase the turnaround time. More disk capacity is
required in a transaction-time server for the storage of the archive and the database. Disk I/O will increase when doing backup operation and database read/write, thus affecting the turnaround time. When the process is waiting for the completion of disk I/O, the CPU will be idling for that process. Therefore, CPU utilization is decreased.

File update rate is another factor we consider. Frequently updated file will involve more versions created in the archive, and the history table is getting larger. Disk I/O will be busier. So, we measure the effect of different file update rates on our web server.

File size is also a crucial factor in determining the overhead. The size of a main file for the resource is related with the building of resource version history. We traverse the file content to figure out the included files in order to get the versions for all the included files. Hence, we experiment with different file sizes.

The final measured factor is the percentage of different TT Queries among all requests. We predict TT Queries will be more time-consuming than normal HTTP requests for the additional database accesses, for the inner-loop of building version histories, and for the file version backup operations.

With the additional vacuuming functionality implemented, request processing will involve vacuuming operations, such as accessing the policy table, performing database updating, and removing versions from the archive. We experiments the overhead brought out by those tasks.
When repairing strategy is supported, the \texttt{Query} processing could include more steps to do repairing. We measure and compare the query processing difference between those having no repairing and those having repairing to determine the effect of repairing.

### 5.1.2 Experiments

We design experiments to independently measure factors of file update rate, file size, percentage of \texttt{Queries}, vacuuming policies, and repairing. The experiments will issue a burst of requests to the web server so that we can measure the performance of the web server under a heavy load situation. We monitor and gather statistics of the system cost in the experiments.

Each experiment consists of four steps. At first, the web server is started from a clean state that the database and archive are new for each experiment. Second, 3000 pages are pre-fetched before running the actual tests. We design this operation in order to reach a similar memory and cache condition before a test starts. Hence, the turnaround time difference will be due to the different factors in the experiments. Next, we perform our experiments. We design the experiments to run repeatedly for several times. And the to-be-requested files are chosen randomly in each run. We take the average of those runs to avoid the potential bias. After those runs finish, we shutdown the web server.

### 5.2 Compare Apache v2.0 with \texttt{Apache v2.0}

We summarize the experiments to compare the performance overhead of Apache v2.0 and \texttt{Apache v2.0} in this section.
5.2.1 Turnaround Time

Figure 5-1 shows the effect of the file size on the turnaround times for experiments running on normal Apache v2.0 and TTApache v2.0 with nine update rates, 0%, 1%, 2%, 5%, 7%, 10%, 15%, 50%, and 100%. Four columns marked as Apache_1KB, TTApache_1KB, Apache_60KB, and TTApache_60KB are compared for every update rate. The label such as Apache_1KB means a 1-kilobyte file experiment with different update rates is performed on normal Apache v2.0. In the small file experiment, Apache and TTApache have very close turnaround times when the file update rate is less or equal to 15%. However, in the large file experiment, the turnaround time of TTApache increased dramatically compared with Apache.

Figure 5-2 shows the effect of file size (1KB, 10KB, 20KB, 40KB, and 60KB) on the turnaround times for TTApache v2.0 with update rates 0%, 5%, 15%, 50%, and 100%. When the file sizes increase steadily, the turnaround times for TTApache increase proportionally. Therefore, the effect of file size dominates the magnitude of a turnaround time.

Recall that the turnaround time is calculated for 3000 requests. So, for the 1-kilobyte file case, TTApache can process approximately 110 requests per second at an update rate from 0% to 15%. There are no significant difference between Apache and TTApache. Therefore, the additional processing in TTApache (accessing database and archive file) is trivial. When the update rate is 50% or more, TTApache slows down the processing since file archiving is more frequently at higher update rates. At the 100% update rate, TTApache processes about 65 requests per second.
Figure 5-1 The effect of file sizes with different update rates

Figure 5-2 The effect of file sizes on TTApache
For the 60-kilobyte file case, 77Apache is able to process about 30 ~ 36 requests per second at an update rate from 0% to 15%. When the update rate is higher than 50%, 77Apache handles 23 ~ 26 requests per second.

Therefore, if a site is averagely accessed below 23 requests per second, 77Apache can run well without harming the performance even though the large files are requested.

5.2.2 Virtual Memory Statistics

We gather information about the memory usage, disk activity, and CPU activity in this section. Memory statistics show the memory used in the virtual memory, buffers, caches, and also the idle memory. Disk activity includes block I/O and virtual memory swap I/O. CPU activity shows the utilization of CPU.

5.2.2.1 Memory

Figure 5-3 and Figure 5-4 show the amount of virtual memory in usage for 1-kilobyte and 60-kilobyte experiments respectively. In the 1-kilobyte experiment, we can see that the normal Apache and 77Apache of file update rate from 0% to 2% have the smallest amount of virtual memory used. As the file update rate increases, 77Apache enlarges the usage of virtual memory gradually; that is because more files need to be reloaded from the disk, extra database read/write and file version archiving. In the 60-kilobyte case, the amount of virtual memory used is close for those tests with different update rates.
Figure 5-3 Virtual Memory used for different update rates (1KB)

Figure 5-4 Virtual Memory used for different update rates (60KB)

Figure 5-5 and Figure 5-6 shows the idle memory in 1-kilobyte and 60-kilobyte experiments. In both figures, the normal Apache 2.0 maintains its idle memory in a
nearly static amount during the entire experiment, with about 10 times difference between the two file sizes. In the 1-kilobyte experiment, the idle memory for \textit{T}_7\textit{Apache} decreases gradually to the lowest point. While in the 60-kilobyte experiment, the idle memory decreases rapidly to the lowest point at the beginning of the experiment. Memory is more intensive in this scenario. Moreover, we capture that \textit{T}_7\textit{Apache} finally maximizes memory usage whatever the file size is, whereas the normal Apache does so only for large files.

![Figure 5-5 Idle memory for different update rates (1KB)](image-url)
Figure 5-6 Idle memory for different update rates (60KB)

The information of used buffer is given in Figure 5-7 and Figure 5-8. Most of the experiments for file size 1-kilobyte have no significant changes in the buffer usage. However, in the 60-kilobyte experiment, only the normal Apache v2.0 represents an approximate stable amount of buffer used. For 77Apache, more memory is consumed for the buffer usage, especially at the update rates of 50% and 100%. When the update rate is high, more file versions need to be archived; thus the requirement for buffer as to a large file size will increase much faster.
Cache usage is shown in Figure 5-9 and Figure 5-10. We can see that in each experiment, the amount of used cache is almost 10 times of the amount of used buffer.
shown in the previous two figures. So buffer is less crucial than cache when considering their effect. The normal Apache v2.0 increases the cache usage for about 33% in the 60-kilobyte experiment, compared with the 1-kilobyte experiment. For Apache, the amount of cache in the 60-kilobyte experiments reaches the highest point immediately, but slowly in the 1-kilobyte experiments. As a comparison, it’s found that the idle memory shown in Figure 5-6 decreases dramatically for 60-kilobyte experiment. Large files need more memory for cache, hence less idle memory is available. In the 1-kilobyte experiment, idle memory decreases gradually shown in Figure 5-5, which conforms to the cache usage information.

![Figure 5-9 Cache used for different update rates (1KB)](image)

Figure 5-9 Cache used for different update rates (1KB)
5.2.2.2 Disk Activity

Figure 5-11 and Figure 5-12 show the disk I/O activities in the two experiments with different update rates. We can see that the 60-kilobyte experiment is nearly 10 times greater than the 1-kilobyte experiment. Large size files occupy more disk blocks. The figures also show that the disk I/O is almost proportional to the file update rate. Tests with higher update rates have busier disk activity for archiving file versions. The exception case is the normal Apache in the 1-kilobyte file size experiment. It keeps a balanced cumulative disk I/O activity even though the update rate is increasing. For the 60-kilobyte experiment, the disk I/O in 77Apache is almost double of the normal Apache at update rate 100%. Database read/write and file versions backup require the extra disk I/O.
Figure 5-11 Cumulative disk I/O for different update rates (1KB)

Figure 5-12 Cumulative disk I/O for different update rates (60KB)
Figure 5-13 Cumulative swap I/O for different update rates (1KB)

Figure 5-14 Cumulative swap I/O for different update rates (60KB)

Figure 5-13 shows there is no virtual memory swapping in the 1-kilobyte experiment except for the 100% update rate test. 100% update rate test invalidates the files
maintained in the memory for every request, thus the swap activity between memory and disk is frequent. For the 60-kilobyte experiment shown in Figure 5-14, the swapping increases step by step as the file update rate increases, since the memory usage is intensive in this case.

5.2.2.3 CPU Utilization

Figure 5-15 shows that the CPU is finely used for the 1-kilobyte experiment. The normal Apache even reaches 100% CPU utilization. We can infer this conclusion from the analysis of memory and disk activity in the previous sections. Normal Apache has a nearly flat and low disk I/O character and has no memory swapping operation. So, there is no requirement for the CPU to wait for either the disk operation or memory paging, thus finally makes full usage of CPU. For 77Apache, we can see there is a series of peaks indicating the CPU is waiting for either disk I/O or memory operation. Figure 5-16 shows the CPU is less effectively used in the 60-kilobyte experiment than in the 1-kilobyte experiment. The normal Apache has an average of 90% CPU utilization, with peaks showing the CPU’s waiting for the disk I/O operation. 77Apache has worse CPU utilization, which vibrates around 40%. It implies the database access and archiving takes most of the processing time, so that the process has to wait for the disk I/O operations.
Figure 5-15 CPU used for different update rates (1KB)

Figure 5-16 CPU used for different update rates (60KB)
5.2.3 Effect of TTQueries Percentage

We carry out experiments to measure the effect of the percentages of TTQueries (mixed queries) on the turnaround times for 1-kilobyte file. Nine file update rates are applied, 0%, 1%, 2%, 5%, 7%, 10%, 15%, 50%, and 100%. Four mix rates of TTQueries are tested, 1%, 5%, 20%, and 80%. The result is shown as a 3-D plot in Figure 5-17. The turnaround times are calculated from the average turnaround times of five runs. For each mix rate, the turnaround time increases when the update rate is getting higher. The longer time is spent in archiving more file versions and traveling through a longer version history.

![Figure 5-17 The effect of the ratio of TTQueries (1KB)](image)

When the mix rate is of 1%, 5%, and 20%, the turnaround times are not significantly different if the file update rate is less than 50%. As the file update rate increases, the turnaround time gets longer. For the high mix rate 80%, the turnaround times increase.
accordingly when the file update rate increases from 0% to 15%. With the 50% and 100% file update rates, the turnaround times increase significantly. The high mix rate and high update rate complicate the request processing, since more file versions are created, thus more database read/write and version backup are required. Version history contains more versions and thus involves more loop operations to build the history.

5.3 Compare \textit{TT}Apache v2.0 with \textit{vTT}Apache v2.0

In this section, we first measure the overhead of the vacuuming functionality. Then, we perform a set of experiments to compare the performance of different vacuuming policies designed and implemented in \textit{vTT}Apache v2.0. Finally, we compare the turnaround times of \textit{TT}Queries with those of repaired \textit{TT}Queries. Different file update rates and file sizes are measured as well. Five consecutive runs are applied for each test, so that we take the average as the result data.

5.3.1 Effect of Vacuuming

We perform experiments of sliding version-window vacuuming [0,\textit{current}] policy, and compare them with experiments on non-vacuuming \textit{TT}Apache. The vacuuming policy only keeps the current file version in the archive. Figure 5-18 shows the result combining factors of file update rate and file size. The label TT_1KB means experiments on \textit{TT}Apache for 1-kilobyte file size, and the label vTT_1KB means experiments done on \textit{vTT}Apache for 1-kilobyte file size. The meaning of TT_60KB and vTT_60KB can be inferred accordingly. As we can see, the turnaround times for \textit{TT}Apache and \textit{vTT}Apache are close to each other for every test of a particular file update rate and file size, except the 50% and 100% update rate for the 60-kilobyte experiments. The higher update rate
and large file size work to slow down the request processing, and vacuuming is applied more frequently under a higher update rate. In general, the vacuuming functionality doesn’t impose an extra load for the web server.

![Figure 5-18 The effect of vacuuming with different update rates and file sizes](image)

### 5.3.2 Different Vacuuming Policies

We design and implement various vacuuming policies in this thesis. In the experiments, we pick policies from different categories to measure and compare their effect on the request processing. We choose sliding version-window policy of $[0, \text{current}]$, sliding time-window of $[\text{begin}, \text{now}]$, and the earliest-percent-different policy of 100. Hence the vacuuming based on versions, time, and file content are all covered. The first two policies will vacuum file versions other than the current version, while the third policy will keep the earliest version and the current version. Figure 5-19 shows the
experiment results. Both file update rate and file size are tested. We can see that there is no significant difference between the version-window and time-window policies. However, the percent-different policy has a relatively longer turnaround time for high file update rates or large file size.

![Graph](image)

**Figure 5-19** The effect of different vacuuming policies with different update rates and file sizes

For the 1-kilobyte experiment, the percent-different policy is close to the version-window or time-window policies when the file update rate is less than 15%. When file update rate gets higher, more actions are needed for applying the vacuuming policy since more versions are created in the history table. Recall that our vTT Apache is implemented to lazily vacuum file versions. Vacuuming is carried out upon incoming request. As more versions are generated, more content comparing actions among versions are involved. Hence, the percent-different policy slows down the turnaround time.
For the 60-kilobyte experiment, the content comparing takes longer as file is larger. So, the percent-different policy is slightly slower than the other policies when the file update rate is low, but this policy slows down the turnaround time more seriously when the file update rate is as high as 50% or 100%.

5.3.3 Effect of Repairing

Figure 5-20 shows the experimental results for queries of vacuumed versions without repairing and queries with repairing. No matter what the file update rate or file size is, the turnaround times for tests without repairing and tests with repairing are very close. It’s no wonder since repairing strategies are integrated into the query processing.

![Figure 5-20 The effect of repairing with different update rates and file sizes](image)

6 Conclusions and Future Work

We summarize the entire work of this thesis in this section and state out future work.
This thesis presents a complete idea of a vacuuming-enhanced transaction-time web server. It presents a model design, model definition, implementation and experiments.

A transaction-time web server stores versions of resources it maintains in an archive. The information of versions is recorded in a history table. Each modification of the main file or included file for the resource will create a new file version in the archive. So, the archive will grow over time and finally fill in all available storage capacity. Vacuuming is introduced to restrict the ever-growing archive size by selecting and eliminating some versions from the archive. Versions are selected by vacuuming policies. This thesis explores a various range of vacuuming policies, such as vacuum every second version or vacuum all versions older than one year. However, the history information is no longer complete and intact when file versions are vacuumed from the archive. Transaction-time queries on vacuumed version are affected. We introduce repairing strategies to handle this scenario.

The history table is append-only in the transaction-time processing, thus will grow over time as well. This thesis introduces coalescing and obliteration to shrink the size of the history table. Coalescing merges adjacent vacuumed version tuples in the history table. Obliteration works as a combination of vacuuming and coalescing, except that it is a one-time operation issued by the web master.

The thesis also presents other resource management capabilities. It allows resource migrate in the archived environment. When a resource moves to a new location, its corresponding history is migrated as well. Operations like setting vacuuming policies, removing vacuuming policies, and setting repairing strategies are supported. Transaction-
time queries are now allowed to specify different version categories as the argument, so that only the matched versions are fetched.

We developed logical models for the above functionality and defined the formal models in detail. All queries are HTTP protocol compatible. We defined the syntax format for those queries meanwhile.

The implementation is done on Apache v2.0. Much effort is spent in making the transaction-time and vacuuming related functionality as independent extension module to Apache v2.0. The whole system is a re-design and re-implementation of Apache v1.0, along with the additional vacuuming idea fulfilled.

We developed a set of empirical experiments to measure the overhead on performance brought by transaction-time and vacuuming functionality, since the versioning, archiving, query processing, and vacuuming involves more work. The experiments are run for varying file sizes, file update rates, mixed Queries, and vacuuming policies. Experiments’ result show that the performance of Apache v2.0 is proportional to the size of requested file, the file update rate, and the ratio of Queries. The comparison of Apache v2.0 with Apache v2.0 also shows that vacuuming functionality won’t compromise the system’s performance. Various vacuuming policies have marginal impact on the request processing. Queries with repairing mechanisms are almost the same as non-repaired queries.

There are several works left for improving our system in the future. First, the syntax of setting repairing has an optional property, time window, as we described in Table 3-3.
It means repairing works only on file versions inside the time window. The current v77 Apache does not cover this property, but it will be implemented in the future. Second, we designed a composite vacuuming policy that defines a \textit{time-periodicity} dimension. An example is to vacuum all versions other than the last one for every month, where the “month” stands for this time-periodicity dimension. It is to be implemented in the next generation of v77 Apache. Third, we adopted the approach of storing entire files when archiving file versions. And vacuuming is operated on file versions. It is worth to explore the storing file difference technique as a way to reduce the archive size. Meanwhile, the vacuuming policies and obliteration will be affected, so as also need to be studied.
WalkThroughVacuum(u: URI, h: FileVersionHistory)

comment: step a
VacuumPolicyList = ε
currEntry = u;
WHILE( currEntry is below DOC_ROOT )
  IF( currEntry exists in S )
    Write the data value pointed by currEntry as s(r, p, v)
    FOR each policy v_i of s.v
      InsertToList( VacuumPolicyList, v_i )
    END FOR loop
    IF( s.p is TRUE )
      comment: If the includeparent option is enabled, trace back to the
      upper level path to locate more vacuuming policies
      currEntry = parent path of currEntry
    ELSE
      go to END WHILE
    END IF
  ELSE
    comment: If no entry found in the policy table, turn to its upper level path
    currEntry = parent path of currEntry
  END IF
END WHILE

comment: step b
FOR each policy v_i in VacuumPolicyList
  ApplyVacuum( h, v_i )
END FOR

Note:
- S_i stands for the policy table, with tuples of (key,data) pair format.
- s(r, p, v) represents a policy tuple pointed by u, where u is a file path; r is the
  repairing option, p is the includeparent option, v is a set of vacuuming policies
  specified on u.
- ApplyVacuum( h, v_i ) performs vacuuming operation on file version history h, under
  the vacuuming policy v_i. It iterates each file version tuple in h, makes decision of
  vacuuming and applies vacuuming on each tuple if required.

Appendix 7-1 WalkThroughVacuum Algorithm
Obliterate-Coalesce( u: URI, t₁, t₂: time )

IF( u is specified on a file )
  *comment: obliterate or coalesce the specified file if it exists in the history table*
  IF( u exists in Hᵢ )
    Build file version history hIn, according to the data value pointed by u
    hOut = ε
    CALL Coalesce( hIn, hOut, t₁, t₂ ) or Obliterate( hIn, hOut, t₁, t₂ )
    Update the data value pointed by u, according to file version history hOut
  END IF
ELSE
  *comment: iterate each tuple in the history table, obliterate those under this directory*
  FOR each tuple Tᵢ(u,h) in Hᵢ
    IF( Tᵢ,u locates below u )
      Build file version history hIn, according to the data value pointed by Tᵢ,u
      hOut = ε
      CALL Coalesce( hIn, hOut, t₁, t₂ ) or Obliterate( hIn, hOut, t₁, t₂ )
      Update the data value pointed by Tᵢ,u, according to file version history hOut
    END IF
  END FOR
END IF

Note:
- Hᵢ stands for the history table.
- Tᵢ(u,h) represents one tuple in the history table, with u be the key value, h be the data value meaning file version history.
- Obliterate and Coalesce functions are described below.

Appendix 7-2 Obliterate-Coalesce Handler Algorithm
Coalesce( hIn, hOut: FileVersionHistory, t₁, t₂: time )

hOut = ε
temp: File version tuple

*comment: bRoundBegin marks the beginning of consecutive vacuumed file versions*
bRoundBegin: Bool

FOR each tuple \( T_i \) (m,r,f,s) of hIn
IF(\( T_i.m \leq t_1 \) )
  *comment: \( T_i \) is outside the time range, replicate it to the file version history*
  AppendToVersionList( hOut, \( T_i \) )
ELSE IF(\( t_1 \leq T_i.m \leq t_2 \) )
  *comment: \( T_i \) is inside the time range*
  IF( \( T_i.f \) is an un-vacuumed version )
    IF( bRoundBegin )
      *comment: record the obliterated version on the file version history*
      AppendToVersionList( hOut, temp )
      bRoundBegin = false
    END IF
    AppendToVersionList( hOut, \( T_i \) )
  END IF
  IF( bRoundBegin == false )
    *comment: keep track of the earliest modification time*
    bRoundBegin = true
    temp.m = \( T_i.m \)
  END IF
  temp.r = \( T_i.r \), temp.s = \( T_i.s \), temp.f = obliterated
ELSE
  *comment: \( T_i \) is outside the time range*
  IF( bRoundBegin )
    *comment: record the obliterated version on the file version history*
    AppendToVersionList( hOut, temp )
    bRoundBegin = false
  END IF
  AppendToVersionList( hOut, \( T_i \) )
END IF
END FOR

Note:
- hIn is the original file version history list which consists of a sequence of file version tuples; hOut is the resulted file version history after the coalesce operation.
- Coalesce works only on file versions inside the closed time range \([t₁, t₂]\).
- \( T(m,r,f,s) \) stands for a file version tuple, where \( m \) is the modification time, \( r \) is the last read time, \( f \) is file version status, \( s \) is sequence number.
- A file version of vacuumed or obliterated status is not an “un-vacuumed” one.

Appendix 7-3 Coalesce Handler Algorithm
Obliterate( hIn, hOut: FileVersionHistory, t₁, t₂: time )

hOut = ε
*comment: bFound is used to mark the file versions inside the time range*
bFound = false
temp: File version tuple

FOR each tuple Tᵢ (m,r,f,s) of hIn

IF(Tᵢ.m ≤ t₁ )
*comment: Tᵢ is outside the time range, replicate it to the file version history*
AppendToVersionList( hOut, Tᵢ )
ELSE IF(t₁ ≤ Tᵢ.m ≤ t₂)
*comment: Tᵢ is inside the time range*
IF( bFound == false )
*comment: keep track of the earliest modification time*
    bFound = true
    temp.m = Tᵢ.m
END IF
    temp.r = Tᵢ.r, temp.s = Tᵢ.s, temp.f = obliterated
IF( Tᵢ.f is an un-vacuumed version )
    Remove this file version from archive
END IF
ELSE
*comment: Tᵢ is outside the time range*
IF( bFound )
*comment: record the obliterated version on the file version history*
AppendToVersionList( hOut, temp )
    bFound = false
END IF
    AppendToVersionList( hOut, Tᵢ )
END IF
END FOR

Note:
- hIn is the original file version history list which consists of a sequence of file version tuples; hOut is constructed as the resulted file version history after the obliterate-vacuum operation.
- Obliteration works only on file versions inside the closed time range [t₁, t₂].
- Obliteration will remove the archived versions.
- T(m,r,f,s) stands for a file version tuple, where m is the modification time, r is the last read time, f is file version status, s is sequence number.
- A file version of vacuumed or obliterated status is not an “un-vacuumed” one.
8 References


