Increasing processing capacity in a data intensive dynamic content model
by Jared Phipps Bratsky

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
no abstract found in this volume

INCREASING PROCESSING CAPACITY

IN A DATA INTENSIVE DYNAMIC CONTENT MODEL

by

Jared Phipps Bratsky

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

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 2002

APPROVAL

of a thesis submitted by

Jared Phipps Bratsky

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Gary Harkin _____ (Signature) 9/2/02 Date

Approved for the Department of Computer Science

Rockford J.Ross _____ (Signature) 8/30/02 Date

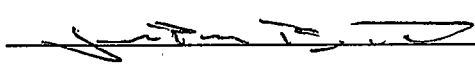
Approved for the College of Graduate Studies

Bruce R. McLeod _____ (Signature) 9-9-02 Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature 

Date 8/30/02

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Many websites deliver substantial quantities of dynamic content. Clients expect their data to be personalized, correct and delivered within a reasonable response time. Popular sites will be required to serve large volumes of data continuously or during peak periods of user demand. Providers utilizing the dynamic content model often generate and manage the content through a common gateway interface and a database management system. The database provides a flexible back-end storage mechanism given large volumes of available and frequently updated data.

Utilizing a DBMS to provide a dynamic content model produces a bottleneck in delivery throughput capacity, because database resources can be simultaneously requested by many clients. While the read requests can be carried out concurrently, the write requests must be carried out atomically, blocking all subsequent requests until it finishes its update. Otherwise, concurrent writes could corrupt the database. As client demand for the content grows, the processing capacity of the DBMS is reached and requests backup and fail to be served.

Given a large volume of available and frequently updated data, how can the processing capacity of this dynamic content model be increased without sacrificing the client requirements of personalization and data integrity? We will be designing a high performance solution, which will increase the throughput capacity of such a model. This

research will consider alternatives to solving this problem and demonstrate the results of implementing the chosen solution.

This high performance solution is designed to address this problem by increasing page-processing capacity, while maintaining the elements of personalization and integrity. Its basic strategy is to combat the shared resource problem by eliminating the majority of the writers. The dynamic content delivery software environment load-balances traffic over multiple web servers, and each of these web servers logs each page request. A utility will process these logs periodically to replicate the functionality of the standard software statistical collection process. This allows the data collection routines on nearly every page request to be bypassed, thus eliminating updates to shared resources. Overall, this strategy provides the increased capacity by allowing more users to concurrently read from the shared resources.

The high performance solution is actually a multiple-phase high performance architecture. The first phase's primary goal is to simply increase the throughput capacity of the current product architecture. Its secondary goal is to provide the foundation for the second phase of the high performance implementation. These design goals effectively produce a read only interface to high-traffic areas of the product. The second phase will extract the resources used for this into a separate physical layer, which will be replicated and serve as a distributed cache. This final architecture produces a potentially unlimited potential for linear expansion in processing capacity.

CHAPTER 2

BACKGROUND

The requirements of serving dynamic content differ from provider to provider, such as the volume of data to be served, the frequency in which the data is updated, the amount of data collected during requests, and the freshness of the data. The deployment methods also differ in terms of software design and hardware architecture. The solution presented will target typical dynamic content model requirements deployed through a specific software design and hardware architecture. This chapter details this product architecture outlining the requirements, design goals and necessity for the high performance solution.

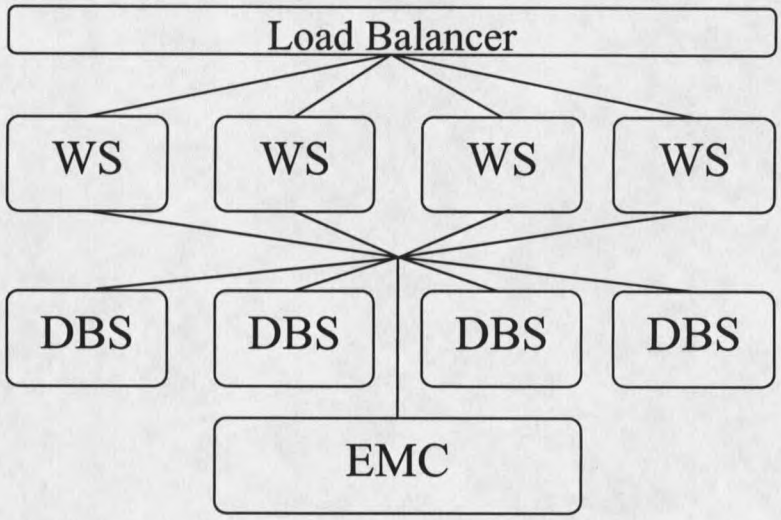
Hardware Environment

The product hardware architecture utilizes a widely-used load balancing environment consisting of a load balancer, a high speed disk, multiple front-end web servers, and multiple back-end database servers.

All requests first come through a hardware load balancer, where the requests are evenly distributed over four to six web servers. Each runs Red Hat Linux and uses the Apache web server. The web servers are connected to a single high speed disk device, a Celerra developed by the EMC Cooperation, which will be referred to as the EMC. The EMC is run by an embedded operating system hosting an NFS file system, which is mounted on each web server. The backend is made up of a variable number of database

servers. Each load balanced environment hosts many individual installations of the product software composed of scripts, CGI's, utilities, and configuration files.

Figure 1. Hardware Environment



To summarize the architecture, consider the flow of a page request. It first enters the environment through the load balancer, which directs it to one of the web servers. The web server handles the target installation's CGI, which connects to the database server hosting the requesting installations database. The page content is then generated and returned to the client.

While an installation's site may be served over multiple web servers, it possesses a single software set residing on the EMC and a single database located on one of database servers.

Product Design

The product design has many integrated features that will need to be maintained in the high performance solution. Each customer installation has a few conceptual key components, such as the knowledge base, a logical portion of an installation's database with a business logic wrapper, and two primary web portals, the end user interface and the administration interface. The end user interface provides an interface for end users to browse the installation, where activities include browsing, searching, and viewing knowledgebase content. The administration interface provides the tools to manage the knowledgebase content, answer end users questions, and analyze business data.

We will now outline some of those features and although all features are deemed important to the business solutions, specifically we will focus on the features that affect the design decisions for the high performance solution.

First, comprehensive end-user session analytics are provided to administrative users, such as content viewed, searches performed, pages turned and customer tracking. To perform the analysis, the user sessions are tracked by generating a unique session id for each distinct user. As each page is turned, data will be inserted into the database based on the session flow. Examples include keywords searched, pages visited, content viewed, and filters chosen from drop down menus. If the end user is logged into the site during their visit, the session will be further tracked by linking the unique id with the customer record of the visitor.

Second, privileged access levels are definable as end user data filters. When a customer record is created it may be associated with one or more administrator defined

access levels. When the customer logs in to the end user interface, their access level will determine the view into the knowledgebase they will receive. For example, a user with full access permissions executes a keyword search. They receive a complete list of knowledgebase content items associated with that keyword. If another user with lesser permission searches on the same word, they may receive only a subset of those items. In like manner, when a user chooses an item from the returned search list, the actual content displayed in the detail page may be tailored around the users access level.

Knowledgebase content relatedness is also determined from end user activity. As users visit the site and view multiple knowledgebase items their relatedness implicitly grows. Business logic generating the end user interface uses the content relatedness to aid future visitors. For example, an end user viewing a knowledge item may get a dynamically generated list of links to other related items based on previous end user visits.

Finally, the actual look and feel of the web site is highly customizable by the administrators of each installation. Pieces of page output may be positioned in the layout or even turned off and on through configuration variables. If these layout configuration options do not sufficiently satisfy users, they are given a final option of gaining access to the actual scripts that generate their user interface, which can be completely customized by a web developer.

Architecture Summary

All of the previous features impact the type of high performance solutions that may be implemented. The specifics of each feature are not as important as the type of feature. The sum of these feature types, commonly found in dynamic content models, results in a small set of requirements. The user interface is highly dynamic providing a large amount of personalization based on user identity. Activity analysis must be performed and at each page and the complete user session data must be tracked and correlated. The content is highly customizable in both knowledgebase content and interface presentation. So, overall the product is very personalized, customizable, and data intensive. This type of dynamic content requires substantial database access to generate each end user page. Additionally, each page turn requires multiple database updates, both to dynamically shape the knowledgebase structure and to perform the session activity analysis.

A Case Study

A case study of a typical customer using the product demonstrates the need for a high performance solution. Over the past year, they have been experiencing an average of 80,000 hits per day. Most of these hits occur over a ten-hour period, which is equivalent to an average hit rate of approximately two hits per second. Peak rates are more than 10 per second, which request serving completely dynamic content, requiring substantial database access. Each page may execute a half dozen database updates and two to three

times as many database queries. Customer service representatives are concurrently accessing the same resources through the administration interface. Large reads may be executed through reporting or customer tracking, which may block the incoming updates for what, in web terms, is considered a large amount of time.

Later in 2002, this customer's load is expected to increase as they release a new line of products through their support interface. Early estimates from the company show projections as high as five times the current page request volume. While this estimate may be a bit high, it is still assumed their traffic will at least double.

Their current load has already prompted changes in their hardware architecture. They have allocated a dedicated database server to help serve their traffic volume, while a typical customer would share hardware with several additional hosted customers. Given the fact that they have already begun to outgrow their current environment, their projected volume increase causes some concern. Thus, the need for a high performance solution has developed.

CHAPTER 3

SOLUTION DESIGN

This chapter will outline the design objectives in the high performance solution implementation, which coupled with the background information, provides a basis for considering existing design alternatives. Each alternative will be explained and measured to how well it meets the solution goal without sacrificing product design requirements, and the final selected solution will be presented.

High Performance Product Design Objectives

As mentioned in the introduction, the first phase of the high performance solution has two design goals: increase the page processing capacity of the current product, and provide a foundation for the second phase of the high performance architecture.

Achieving these goals comes with a strict set of design and implementation objectives. Foremost, the implementation needs to be cost effective, both in terms of development time and hardware resources. Specifically, the solution will need to be designed, implemented, tested and deployed over a six-month period. Additionally, the current hardware environment provides an architecture requiring a relatively small amount of hardware resources to serve a large customer base. The solution must attempt to maintain this low ratio of hardware to customer installations.

Next, the solution is to be architecturally transparent. The high performance solution can be thought of as an optional transparent layer over the existing core product

implementation, and therefore it is a requirement to make few changes to the current architecture. This becomes an important objective because development on the main product can proceed without much concern for its impact on the high performance solution. The architectural transparency provides that the core product and high performance solution be deployed on mutually exclusive release schedules. One last important aspect of this transparency is that it needs to be extended to the end user. Every feature in the core product, down to its lowest level of functionality, needs to be completely preserved by the high performance solution.

Last, the solution itself needs to be highly efficient. If its execution consumes system resources for an extended period of time, the achievement of the primary goals may be unachievable.

Alternative Solution Design: Proxy Caching

Proxy caching has been used for performance improvement for some time. Whether the requested data are static or dynamic, proxy caches store data at locations that are remote to the actual server that generated the data. Proxy caches are often used to store data in locations that are geographically closer the requester, thus reducing network latency. They also reduce server load for the creator of the data source by providing the requested data from their intermediary locations. [8]

Due to the rapidly changing dynamic content generated by the core product, proxy caching was never an option. The current implementation goes to great lengths to ensure that portions of the user interface are not inadvertently cached. A page served

from an intermediary source would disrupt the functionality in the product. The servers need to know of the request for statistical analysis. If the end user received a cached page many problems arise, such as outdated content. But most importantly, due to the dynamic nature of content generation, the links within the cached page may no longer be valid.

Alternative Solution Design: Local Caching of Dynamic Data

Another form of caching was an early design path, caching the dynamic pages locally. Caching dynamically generated pages can take many forms, but in general, it performs as a typical cache. A portion of memory is initially allocated for the cache. This memory can be main memory, disk, or a combination of both. As requests for dynamic content are made to the web servers, it first checks the cache for the requested page. If the page is a cache hit, it is served directly from the cache memory. When a cache miss occurs, the CGI is executed and its content is delivered to the requester and stored in the cache [12].

In the product environment, the proposed cache would exist in the form of shared memory on the high speed EMC disk. Caching would be monitored by a custom-built management server, where cache population, page replace, expiration algorithms would be implemented.

Locally caching dynamic content provides extremely fast processing capacity when the hit rate is high. Early testing showed a cache performance nearing 250 pages per second. Other similar cache design show similar, and even better performance with smaller page sizes [8, 9]. This solution would meet the design objectives: cost

effectiveness, transparency, and efficiency, and therefore, this was heavily researched in the early stages of design.

Session tracking and personalization presented major challenges in local cache design. Each page generated contains many links to other areas of the product; these links may be in the form of user interface elements such as tabs and buttons or simple hypertext links. Some links always exist, and some are dynamic, such as search results. To map user sessions together, their unique id must be passed through to the next page, which means it must be embedded in each link available. So, the embedded content within the page is distinct for each user, which does not work well with a traditional cache of web pages.

Simple personalization issues follow in the same manner, where portions of some pages may be hidden or display to a client based on identity, access level, and past activity. Through the solution design objective, these features needed to be preserved in their entirety.

Therefore, a hybrid cache design was proposed. The cache manager would parse the pages returned from the CGI and place a logic-encoded image of the page in the cache. It consisted of two logical portions, a header with generation logic, and a page shell containing the actual HTML to be returned. When a page was generated from the cache, the header was examined to determine a plan for expanding the requested HTML content. Then the session specific dynamic components would be replaced as the page content was pushed to the user, from personalization information down to every unique id in the embedded links.

Searching complicates the cache design. The search lists must be filtered based on the client's identity and preferences, the user may specify the number result items they received per page.

To solve all of these types of tracking and personalization issues, the cache was designed as a multi-layer architecture, where pieces of process pages, lists of search results, and client information were all stored into separate data structures. The cache manager would then examine each incoming request, determine its ability to generate the page based on the cache data structures, and either construct the page for the user or pass it along to the application software.

While this strategy is highly efficient and architecturally transparent to the core product, it is also very complex, nearly re-implementing the product software. In reality the previous description doesn't give its complexity the justice it deserves. Small issues affecting the design continued to surface as each product feature was examined. Also, because the cache manager basically tries to replicate the CGI execution results, changes to the core product will require changes to the cache architecture. Therefore, this solution does not prove to be cost effective or developmentally transparent.

Alternative Solution Design: Data Modeling for a DIWS

The product implementation truly is a data intensive web site (DIWS), which is a web site that uses a database as a back-end for the storage of large volumes of frequently updated data and show high demand for data accessed through their web pages, continuously or in specific peak periods [3]. A number of papers have been published

presenting solutions in working with these data model issues in a data intensive web environment [2, 3,9,10], all of which have a common underlying theme to break out content generation into a multi-tier design process. A high-level definition language is used to define the general page layout and structure, and multiple intermediate layers of content definition are used to fill in the holes during page generation. In this model, the page generation is still dynamic, however, each layer of page content may be cached separately. A page may be built by interleaving layers generated by the HDL, cached structure content, and content directly from the database. Accessing the upper layers cache for page layout cuts page-processing load, and accessing the lower layers for content reduces database requests.

In many ways this strategy is similar to the hybrid cache explained in the previous section. Its main difference is its layer of implementation. The cache is a transparent entity over the existing product; this strategy is an alternative method of CGI processing for the product itself. Therefore, although this data representation model provides some server performance benefits it does not lend itself well to the core product design. Starting with the HDL used for page layout, the entire user interface portions of the CGI would need to be rewritten, violating the design goal of architectural transparency. More importantly, this scheme does not effectively combat the shared resource problems because each page generation will still require multiple updates to the database as it only reduces queries. This scheme's main performance benefits will be achieved on the web servers, which generate the page layout. This scheme will exacerbate the existing

bottleneck because the web servers have already been proven relatively lightly loaded in comparison to the database servers.

Alternative Solution Design: Caching Database Query Results

We have seen how altering or rewriting the CGI as an efficiency strategy is undesirable because it violates architectural transparency. Furthermore, we have seen that a hybrid dynamic cache, which transparently reproduces the CGI activity, is also undesirable based on complexity. An alternative solution is to not attempt to alter or reproduce the CGI. Caching database query results is a low-level performance enhancing strategy. This strategy would execute core product normally, but attempt to increase performance by reducing the number of queries to the database server by caching query results.

The key problem is keeping the query result cache up to date after database updates are executed. Middleware solutions to this problem have been implemented, where the query cache is kept up to date through variations of DUP (database update protocol) [6]. These solutions vary in performance depending on the update algorithm chosen to refresh the cache. Performance degrades as the cache update frequency increases, so there is a strict tradeoff between cache performance and cache integrity.

Once again this scheme does not work well with the data intensive model of the core product. Both personalization and statistical analysis require a near real time view of the knowledge base. Therefore, in practice, the frequent updates database would produce

a cache that was continually being refreshed. So, the resource contention persists with the bottleneck around the DBMS and cache activity.

Alternative Solution Design: Web Clusters and Distributed Databases

Web clustering is similar to the current hardware architecture. Its primary goal is to build a performance increasing architecture by increasing the number of back end resources. A typical web clustering environment also load balances requests over the multiple web servers in order to distribute system load. Much work has been done in this area and it has proven to be an effective solution for serving high volumes of static and dynamic content [4].

Web clustering in general terms has some key differentiating factors separating it from the current product architecture. First, it expects that in many cases the highly requested content is static, and therefore may be replicated over each node in the cluster. Very little to none of the core product feature set calls for static content generation. Second, if the content is dynamically generated it proves most efficient if the generation is completely handled by the serving node and its dedicated resources. This means that the nodes in the cluster should be able to generate content independent of any shared resource. The product architecture requires a common point of access for each page generation, the installation's DBMS.

The current hardware architecture provides a partial web cluster. The CGI is actually processed on the load balanced web servers, which provides some scalability, but also accesses a common database introducing a bottleneck. Thus, the clustering

performance benefits are defeated by this single access point used for dynamic page generation.

This dilemma leads itself to an obvious question, "can the data store used for dynamic generation be distributed throughout the web cluster?" Unfortunately, this would break existing core product functionality. Consider a simple example: a client logs in to the web site. Their unique session id is recorded in the database and mapped to their client record. Now, they proceed to the second page during their visit. A different node in the cluster, which has its own copy of the database, handles this request. This node has no record of the customer session id mapping and therefore the customer is deemed not to be logged in. Second, the session data is now spread over multiple nodes in the cluster, therefore breaking the existing business logic in analyzing user session activity for knowledgebase management and reporting purposes. This example is one of many in terms of features breaking because they are unable to handle the distributed data. All of these problems could eventually be corrected through sufficient change in the business logic processing. However, this violates the design goal of cost effectiveness and architectural transparency by requiring a major rewrite to existing portions of the core product.

Alternatively, it would be possible for each node to contain a true replica, and the updates to each node to be propagated across the cluster. And after a good number of synchronization issues are resolved, a distributed architecture is produced on which the core product could operate normally. Unfortunately, the new architecture also requires the same total number of updates per database, thereby defeating the original goal of

improving performance, because in addition to replicating the database on each cluster node, we have also replicated the high load average. While this is a significant problem, some aspects of this design can be readily used, as discussed later.

The High Performance Product Design

A common failure in all of the previous design failures is they were unable to simultaneously provide a transparent architecture and increase the performance of the core product. Some solutions, namely the web clusters and query caching, provided transparently but did not effectively increase performance. The solutions were hampered by the product's need to continually update the database. The second set of solutions, including data modeling for a DIWS and all the caching schemes, proxy, local, and local hybrid, provide substantial performance improvements, but could not be implemented transparently to the existing product. These solutions violated the transparency objective by one of two reasons. One, they required major modifications to the core product. And two, they were unable to effectively serve highly personalized and dynamic content in the manner that fully reproduced the current product feature set.

Exploring possible solutions demonstrated a core set of fundamental problems. Architectural transparency is difficult to achieve through an environment that serves highly dynamic and personalized web content because as we have seen, performance-enhancing designs for this model require major modifications to the software architecture or a reduction in feature set. Furthermore, whether the solution is transparent or not, the

requirement for frequent database updates undermines efforts to increase performance by leaving a bottleneck around DBMS activity.

An improved design concept is to provide a two-phase transparent solution. First, attack the DBMS bottleneck by finding a way to reduce the database update frequency. This will increase performance, but more importantly, it will provide a foundation for greater performance improvements. If the updates are sufficiently suppressed, database activity essentially becomes read only. This opens the door to many design options for the second phase of the high performance solution. Since the interface can now be generated from a read only database source, the previously rejected solutions can be revisited. For example, if latency still persists around the database, the query-caching scheme could be implemented. The cache would achieve a high hit rate due to the now static database data. The web cluster could also be fully implemented, with the read-only content distributed to each node in the cluster, reducing an already widened database bottleneck by distributing the workload over multiple servers. The important result is the elimination of the frequent updates to provide the foundation for additional performance enhancements, and the read only data access reopens design options to provide this scalability with architectural transparency. The first phase of the design will reduce frequency of database updates requested of the DBMS.

As pages are requested, the web server writes detailed logs containing information about the request and the requestor. A page request, or URI, contains the request destination appended with parameters collected for the form submission. Since business logic within the CGI normally manages the database updates, all the information needed

to perform these operations is contained within the URI. The implementation strategy is to collect all of these web logs and reproduce the database updates from their content. Sawmill is the utility designed to do this processing. Once an hour, the logs will be transferred from the web servers to the mill, where Sawmill will process them, apply the business logic from the core product, and update the database.

Sawmill meets all three design objectives. First, it is cost effective, providing a simple yet elegant design that requires no additional hardware and acceptable development resources. Second, it is architecturally transparent. The web logs are inherently transparent, as the CGI has no knowledge of them. It provides application transparency because minimal changes to the product are needed, simply a configuration variable which when turned on will cause the database update routines to be bypassed. Furthermore, it completely and exactly reproduces the data normally generated through the core product, providing transparency to the end user and administrator. Sawmill also provides developmental transparency, as it can be developed independently of the core product. It actually incorporates the business logic through shared libraries, which will automatically update Sawmill's functionality as the product changes. Third, it is efficient. Considerable work went into its implementation design, providing a robust and extraordinarily efficient application. Each Sawmill run compresses and executes all of the business logic and database management, normally performed by thousands of CGI executions, into a fraction of the previously needed processing time. This efficient design implementation will be detailed in the next two chapters.

The first phase of this solution also has two primary design goals. The first is to increase the page processing capacity of the current product, and the second is to provide the foundation for the second phase of the high performance architecture. The results and conclusions chapter will examine to what degree each of these goals were met. It will also explore the exciting scalability levels that will be achieved in implementing the second phase of the high performance architecture.

CHAPTER 4

SAWMILL SUPPORT IMPLEMENTAION

One of Sawmill's primary objectives is to provide architectural transparency, which includes reproducing the data normally generated by the product. To completely reproduce this data, Sawmill needs a little help. This chapter will describe Sawmill's support implementation.

The support comes in three forms that will be outlined in this chapter's three sections. First, changes to the log format will be needed. Second, a change in content needs to be performed before log entries are written and before the CGI is invoked. Third, a common location for log storage will be created to separate working data from logs that are currently building. Their location on a common file system will simplify the implementation.

The first two support items come about because the web logs are Sawmill's single source of data. Therefore, it needs to have sufficient data to completely reproduce the database content. The third support item is more a convenience than a necessity, however, we will describe later how it does provide some robustness to the log processing.

Web Log Format

Sawmill needs an extended log format to determine enough data from the web logs to update the databases. The first item in the new log format is the host domain name in the URL in request. Sawmill will use it to help determine to which database the entry is destined. Next the Unix timestamp is included for efficiency; the date and time will not need to be parsed out. Other items include the client IP address, the request's http status, and the http referrer. Based on these data items, Sawmill may choose to disregard the entry. The request may have been a redirect, error or the client may be in the list of excluded requestors. A detailed list of the items in the log format is included in Appendix A.

Web Server Module

A change in URL content is needed to associate users' sessions together. Consider a user who types in the URL of a site's home page, then visits two additional pages during their visit. The product views this as a single user session with three page hits.

The CGI can simply accumulate this information. When the first page is hit, with no parameters, the CGI determines a new session has begun, generates a new session id and stores it in the database. When the user moves to the next page, this session id is passed along and the database is again updated with the current session id. The unique session id is passed from page to page in order that a distinct user's data can be mapped together.

Sawmill cannot map a complete session without a little help. From the previous example, we know the user typed in the URL representing the first page hit. Since the user cannot add his own session id, the URL is missing this key component. As the page is served, the CGI will generate the missing id, but the URL with an empty parameter string will have already been written to the web log. The session id will only appear in the URL for the next two page requests. Thus, Sawmill is unable to correlate the first page hit with the last two.

To solve the session mapping problem, the URL content is modified as the web server receives it. A custom-built Apache web server module, mod_sawmill, is installed on each server. The module inspects each URL before it can be handled by the CGI or written to a web log. When a URL with no arguments is encountered, a new session id is generated and appended as the GET parameters. When the CGI processes the request, the valid session id will be detected and passed along the subsequent pages that user may visit. This start of session URL is written to the log with the newly generated id.

The module has two more functions, it must distinguish between requests for different sites and it must handle session expiration. The first issue is not an issue for Sawmill but an issue for the sites that are not Sawmill enabled. The normal CGI execution has logic based around the generation of a new session id, for example, a counter, representing the number of unique sessions, is incremented each time a new session id is created. If a valid session id is appended to the request before the CGI processes it, new ids would not be created, and the counter would not be incremented.

So, a new Apache configuration directive was created, SawmillEnabled, which goes in the web server's configuration file and read by the module during initialization. A SawmillEnabled entry is entered in the configuration file for each site enabled. The directive's first argument is the hostname of the target site and the second is its interface name. When the module reads these entries they are stored in a hash table, which will exist as long as the web server runs.

As the module examines a URL, it extracts the host name of the request from Apache's internal data structures. It then attempts to look up that entry in the persistent hash. If it exists, Sawmill is enabled for this site, and the URL is further inspected for a valid session id. If it does not exist, the module will not alter the request.

The second function was handling session expiration. Each site is allowed to configure a session expiration length in minutes. The module cannot simply validate a URL because it contains a session id. If the session is expired, the CGI will generate a new one and the session id listed for the request in the web log would differ from the actual id associated with that page. So, the module must also validate the session age. The timestamp of session id's creation is encoded in four bytes of the id. The module parses these out, checks its age, and generates a new session id if the session has expired. Since each site may have a different session expiration length, this value is passed into the module as the third SawmillEnabled directive argument. It is stored in the module's persistent hostname hash and retrieved when the hostname of a request looked up.

Log Rotation

The log rotation is handled by a script run as a scheduled task on each web server. Each hour the logs are moved from their original location to a common location on the EMC server. The common location serves two purposes for Sawmill, it does not need to know detailed information about each web server in its environment and it simplifies Sawmill's decision-making process in log selection. So each log in the common location is guaranteed to be available and ready for processing. If Sawmill sought its logs in their native locations on each web server, it would need to differentiate between complete and active logs.

CHAPTER 5

SAWMILL IMPLEMENATAION

The primary implementation goal of Sawmill was to maintain data integrity. Each database needs to be updated with entirely correct and complete data. Sawmill is designed to be a very robust utility in order to ensure that the data integrity demands are met. At the point Sawmill begins to run, the data within logs is assumed to be complete. It does little verification of actual log data; it only maintains that the logs to process actually exist. When the run begins, Sawmill takes the responsibility of updating each database correctly, and in doing so, it needs to be able to recover from a variety of error states. The machine could crash, Sawmill could be manually terminated, etc.

The second implementation goal is efficiency, which proves to be challenging given the large amounts of data. It is required to process combined logs, gigabytes in size, and update each database in as little time as possible. Along these lines, C was chosen as the language for implementation.

This section breaks execution into phases. Sawmill is able to recovery for error conditions arising in any of these phases.

Phase 1: Configuration

Sawmill will first check to make sure all necessary configuration files exist for execution. It will read in the sawmill.conf file to obtain four important pieces of information for the run: the number of web servers in the target environment, the path to

the logs to be processed, the path to which the processed logs belong, and the path the mill or Sawmill's working directory. The configuration also checks for the existence of the siteinfo.conf file, which will later be read to obtain information about the databases it will be operating on.

Execution will be terminated if the configuration files do not exist or are not in the specified format. An error message is displayed explaining the problem. If execution halts at this phase, no recovery is needed as no data has been processed.

Phase 2: Setup

Now that Sawmill has learned its execution plan, it will verify the existence of all specified paths: available logs, processed logs, and the Sawmill working directory. If the location of the mill does not exist, Sawmill will attempt to create it. It also checks a temporary directory within the mill and all files in it are unlinked.

The files that may exist in the temporary directory are logs built by a previous run. Their existence means Sawmill exited in an error state during that run. However, that fact will be ignored during this phase because the error may or may not have been fatal to the data collection. These logs and their purpose will be explained in detail in the Log Processing and Shutdown phases.

Any errors encountered performing the task during the Setup phase will halt execution of Sawmill. As in the previous phase, no data has been transferred so recovery will not be needed on subsequent runs.

Phase 3: Status Check

Before processing new data, Sawmill will check to make sure it successfully completed its previous run by looking for a file name 'sawmill.status' in the working directory. If this file exists, then Sawmill did not complete its previous run and will enter into recovery mode (described in the next chapter).

If this file does not exist, the last run was a success. Phase 4 will create a new sawmill.status file and update it during the run. Upon successful completion of its tasks, the status file will be removed, thus alerting the next run of its completion.

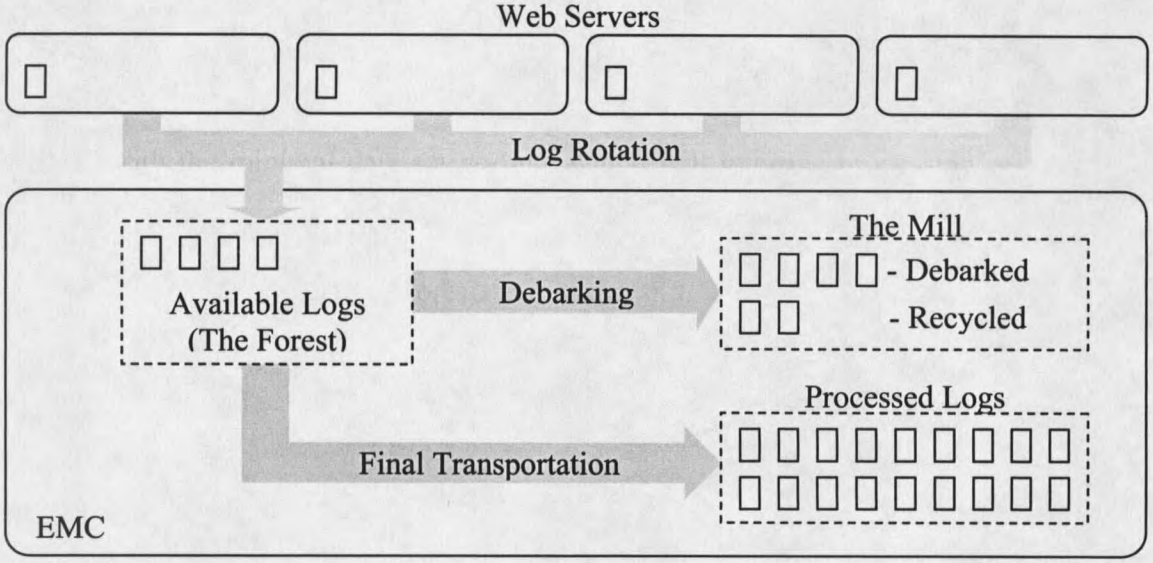
Phase 4: Log Transfer and Preparation

This phase's first task is to load the logs for processing. Sawmill will look into the available logs directory, The Forest, and make sure all of the logs needed for the run exist. If it is the noon hour, it will make sure there is one log from each web server for the 11:00 hour. If any older logs are found they are added to the list to be processed as their existence means a previous run was skipped for some reason, but the data is still needed for processing.

It will make an image of each log in the mill, which is not an exact copy, because the log contains much information that will not be needed for processing. For example, the log will contain http requests for gif, js, css files, etc. So the image of each log is created with the minimal data set needed, which will improve processing performance. Think of this setup as the Debarking. After a log is debarked into the mill, the actual log

will be moved from the forest to the processed log location. This continues until all of the available logs are depleted.

Figure 2. Log Transfer



Removing the logs permanently from the forest is a very important step. This ensures no log will be attempted to be processed twice. Additionally, no log will need the expense of being debarked multiple times.

Sawmill looks into the mill and stores the names of all the logs in an array for processing. Two types of logs will exist: the newly created debarked logs and the recycled logs that Sawmill builds during the data transfer phase. These logs do not need to be debarked as those collected from the available logs location; they only contain information essential to the data transfer.

Finally, an empty 'sawmill.status' file is created, which is critical for the recovery mode. If Sawmill were to be terminated at this instant, the next run would see that the

empty status file had been created, and it can determine that the logs had already been debarked and moved to The Mill and processing would then proceed from here.

If execution halts during this phase before the status file is created, before the logs are moved, or even during the move, Sawmill can recover without recovery mode. The easy case is that no logs had been moved as the logs will all still exist in the forest and will be debarked and moved during the new run. Next, Sawmill can halt while debarking a log, for example, log 2 of 3. This too recovers since the logs are not removed from the forest until after the debarked copy is created. The first log would have been removed from the forest and its debarked copy would still reside in the mill. Logs two and three still exist in the forest. The new run will overwrite the partially created debarked copy of log two and move on to log three. The logs are eventually removed from the mill during shutdown.

Phase 5: Memory Allocation and Management

The next task is to allocate memory for use during processing. As a whole, this phase is an important piece in achieving the performance goal Sawmill. The data sets involved are very large, meaning that performing incremental allocation, reallocation, or inefficient access during processing could become expensive. So, Sawmill completely manages every byte of its own memory to aid performance.

Large chunks of memory are allocated for use during processing. The memory will be used for two main purposes, a general storage bin and a conduit for database transfer. The general storage heap is allocated as a linked list of segments, which are

dynamically added as they fill. This strategy is used to eliminate the need for many incremental allocations. It also eliminates reallocation costs because new segments are added to the heap rather the heap being resized. The initial segment size is estimated as a factor of the total size of the logs to process, which cuts down on the total number segments to be allocated.

Access to the heap memory has also been optimized. Once the data is copied into the heap, the goal would be to access the memory in place, rather than copy it back out. However, the heap is void memory and the data placed in the heap is of variable type and size. Sawmill uses a heap API to align all data and manage structures.

To improve data transfer performance, several smaller heaps are allocated to hold the data in a format optimized for the database. This memory is treated as a pipe, which fills as the logs are processed. When full, it is transferred to the target databases in batch mode. Once transferred, the pipe begins to fill again. This process continues throughout processing.

The data transfer heaps are sized differently than the general storage heaps. Rather than being based on the total log size, they are sized for optimal database transfer. Normally, batch mode updates to a database are much faster than sequential updates. However, when doing these batch updates to a database, a tradeoff situation may be encountered. As the batch update builds, a transaction file is also built. If the transaction file grows extremely large, it can actually take longer to process the group of updates than it would to do them sequentially. The data transfer heaps are sized to avoid this threshold.

Phase 6: Dynamic Libraries, Loading and Linking

A typical visit to the site will invoke a number of resources, which will require the database associated with the target site be accessed. In each case the access is done through a common dynamic library, which contains the database API.

An installation has a single database, which can be MySQL, Oracle, SQL Server, or other. To handle this transparently from a development standpoint, the shared object containing the database API exists in multiple versions. A version is built for each platform/database combination offered. Although the code in each version differs, the API is identical, thus allowing developers transparent access from the code base.

In an addition to a database, each installation has its own set of utilities, scripts, CGI, and dynamic libraries. The libraries are built for the specific platform/database combination of each installation. Sawmill needs to be able to access multiple database types during a single run, because it touches all installations in a single run, where typical utilities are run for each individually.

The goal of this phase is to reuse the common API code. Achieving this goal has two benefits. First, it is better to reuse the tested API functionality than to re-implement the database connectivity. Second, it provides means for Sawmill to update the multiple database types from a single code path.

The first step in implementation is to load each version of the shared object. As each is opened, the address of each symbol within the library is located and stored in a data structure in memory, which at its highest level is indexed by database type. Each