

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
© Copyright by Jared Phipps Bratsky (2002)

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

© COPYRIGHT

by

Jared Phipps Bratsky

2002

All Right Reserved

N378
B7359

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

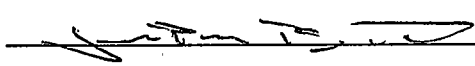
Approved for the College of Graduate Studies

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

1. INTRODUCTION 1

2. BACKGROUND 3

Hardware Environment 3
Product Design 5
Architecture Summary 7
A Case Study 7

3. SOLUTION DESIGN 9

High Performance Product Design Objectives 9
Alternative Solution Design: Proxy Caching 10
Alternative Solution Design: Local Caching of Dynamic Data 11
Alternative Solution Design: Data Modeling for a DIWS 13
Alternative Solution Design: Caching Database Query Results 15
Alternative Solution Design: Web Clusters and Distributed Databases 16
The High Performance Product Design 18

4. SAWMILL SUPPORT IMPLEMENTAION 22

Web Log Format 23
Web Server Module 23
Log Rotation 26

5. SAWMILL IMPLEMENATAION 27

Phase 1: Configuration 27
Phase 2: Setup 28
Phase 3: Status Check 29
Phase 4: Log Transfer and Preparation 29
Phase 5: Memory Allocation and Management 31
Phase 6: Dynamic Libraries, Loading and Linking 33
Phase 7: Initialization 34
Phase 8: Log Processing 34
Phase 9: Data Transfer 38
Phase 10: Shutdown 42

6. SAWMILL RECOVERY MODE IMPLEMENTATION 43

Error Detection 43
Recovery Mode 43

7. RESULTS AND CONCLUSIONS 46

Design Objective: Cost Effective Implementation 46
Design Objective: Architectural Transparency 47
Design Objective: Execution efficiency 48
High Performance Goal: Increase page processing capacity 50
High Performance Goal: Foundation for Expansion 52

REFERENCES 57

APPENDICES 61

APPENDIX A: Requirements 62
APPENDIX B: Mod_sawmill 64
APPENDIX C: Custom Log Format 66
APPENDIX D: Log Rotation 68
APPENDIX E: GGI Preparation 70
APPENDIX F: Sawmill Environment 72
dlls 73
Samill.conf 74
Siteinfo.conf 75
tr.sawmill 77
APPENDIX G: Running Sawmill 79
APPENDIX H: Running Sawmill – Recovery Mode 81

LIST OF TABLES

Table Page

1. Sawmill execution statistics 49

2. Page Processing Capacity Results 51

LIST OF FIGURES

Figure Page

1. Hardware Environment 4

2. Log Transfer 30

3. Processing Phase Memory Structure 36

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.

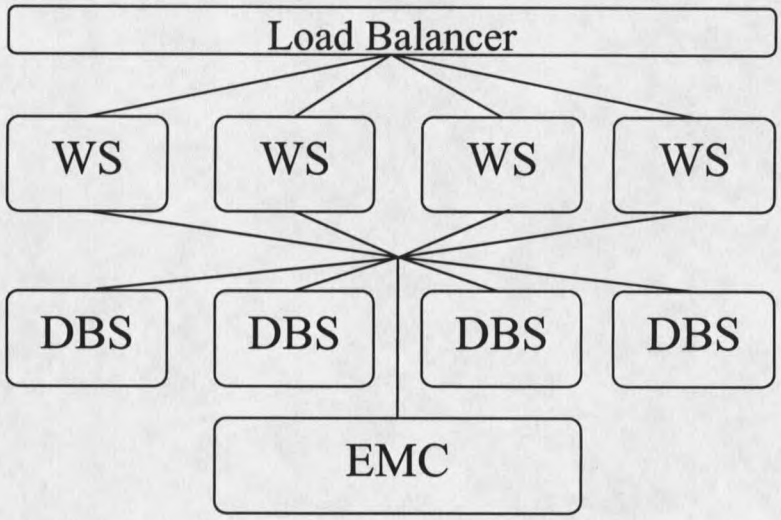
Hardware Environment

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.

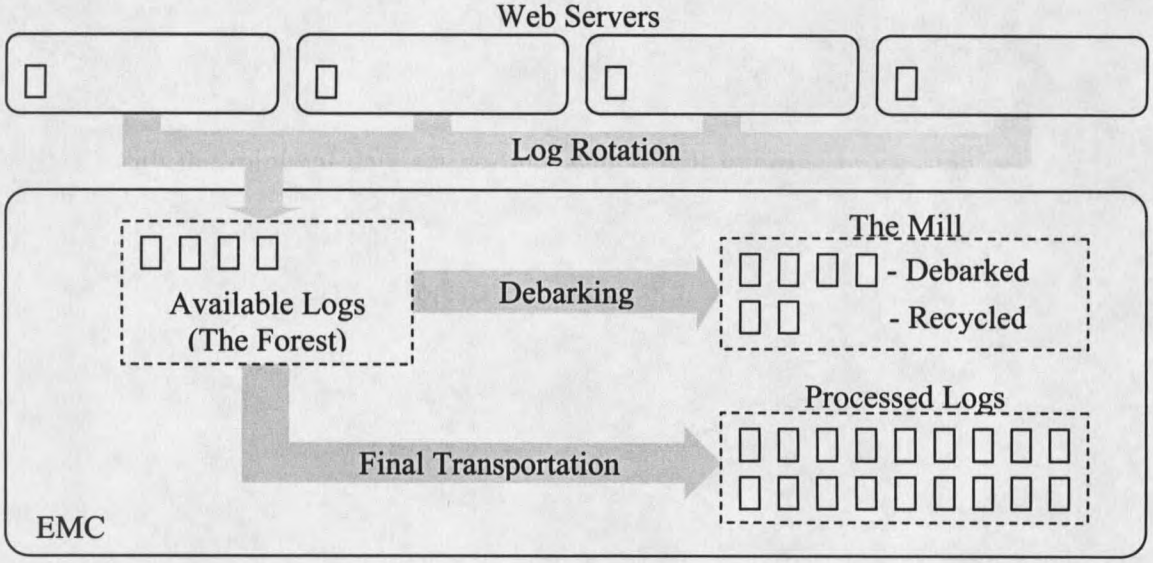
Phase 4: Log Transfer and Preparation

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

routine used during the processing phases of Sawmill will access the database through this global data structure. Therefore, the correct symbols will be used from a single code path by simply knowing the database type of the installation being processed.

Phase 7: Initialization

Initialization reads the siteinfo.conf file to obtain information about the databases it will be updating shortly, such as, type and login information for the database, the session expiration length, and the time-zone skew between the database location and the database's interface setting. These settings and how they are used are described in detail in the Processing phase.

Sawmill may be processing data for thousands of databases during one run. Site information for each database will be looked up multiple times during the run requiring an efficient lookup. Therefore, this phase builds a hash table referencing each database's structure of configuration information. The unique key is the domain name of the site concatenated with the interface name. This hash will later be referenced as the main hash.

Phase 8: Log Processing

Sawmill will read in each log from the mill and break them into individual page hits entities. As the logs are processed, these 'chips' are organized into bins for the data transfer step.

The chips are completely processed and stored into memory during this step making use of the allocation provided during initialization. Any chips determined to be unusable are discarded as some may be for sites that are not Sawmill enabled or may contain information not needed by the database.

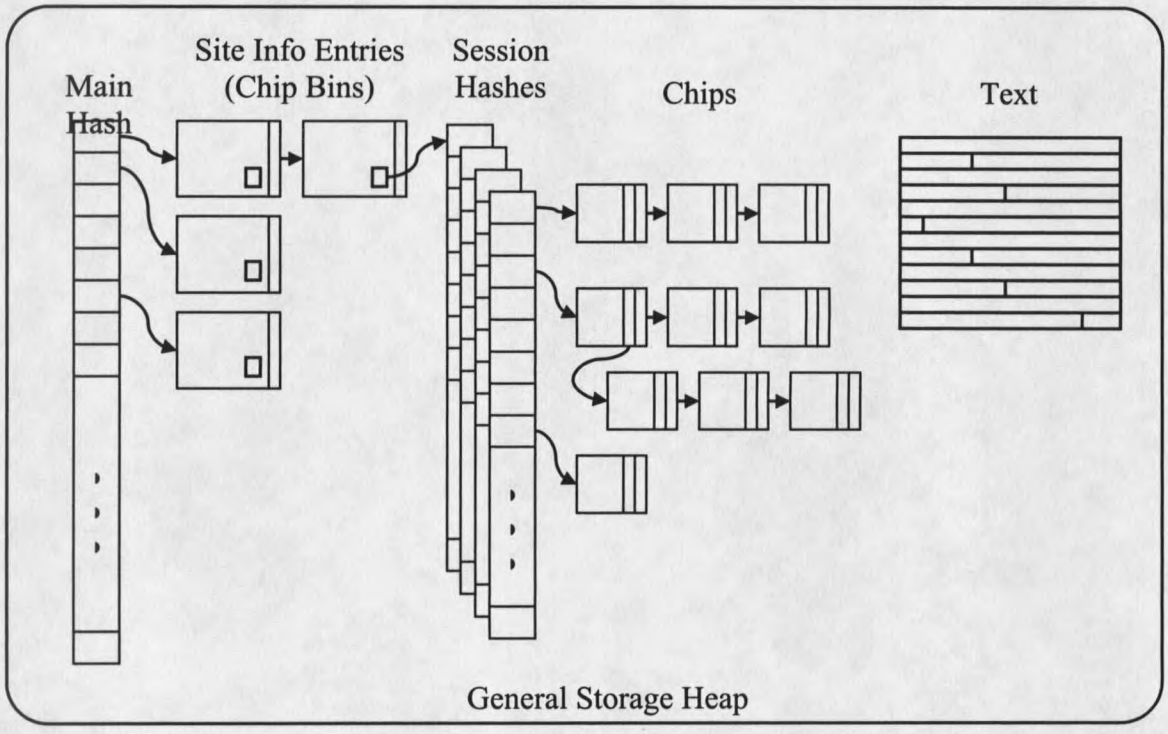
The log processing design is important for the internals of this phase, as well as, the setup for the data transfer phase. It needs to be efficient due to the millions of chips created during processing. It also sets the stage for the next phase by organizing the chips into data structures that will streamline the data transfer.

To understand this design, first consider the key component of a chip, which is simply a data structure that contains parsed data for each entry in the web log. First, each chip contains its basic uniquely identifying components, the user session id and the timestamp of the request. The chip contains many other components needed by the database, such as requested page name, client IP address, a reference to the GET parameters in the request, etc. Only a reference to the URL parameters is stored in the chip, because the text is highly variable in length and in some cases not accessed by the data transfer stage. Therefore, a compressed version of the text is stored in a separate segment within the general heap.

The chip has two other identifying pieces that are not actually stored within its data structure. The domain and interface name of the request are also parsed out of the log entry. However, this data will be used to group the chips into bins and therefore be stored at a higher level.

This brings us to the organizational structure that groups the chips. This structure is defined for the data transfer phase, which at a high level needs database information grouped together. At the next level, it will need user sessions grouped together.

Figure 3. Processing Phase Memory Structure



The organizational structure starts with the main hash described in the Initialization phase. This hash has entries containing site-specific information about each database. In addition to this configuration information, the entries contain a hash for user sessions targeting that database. This is why the domain and interface name are not needed in the chip. They are stored in the main hash.

So the main hash is actually a hash of hashes. The entries in the inner hashes, or session hash, are linked lists of chips representing a user session. The session hash key is

the user session id from the chip. The head of each session list may actually be multiply linked. The first linking dimension handles collisions when multiple session ids map to the same index. The second linking dimension builds the actual list of chips that make up the session.

The ordering of the session list is an important aspect of the data structure. As the web server environment is load balanced, a user session may have been spread over multiple web servers. This means that session's entries will be spread across multiple logs. Since each log is processed sequentially before moving onto the next, it is unlikely the pieces of the session will arrive in order. Each session list is ordered by the timestamp of the request. New session chips are inserting into the correct position in the session list, taking care that both dimensions of linking are preserved.

Handling these sessions distributed across multiple logs is the reason Sawmill needs to verify each log exists during the log transfer stage. If a log is missing for some reason, pieces of some sessions may be missing, corrupting the data going into the database.

The size of each hash is optimized for the working environment. The typical number of installations handled by one load balanced environment multiplied by the average number of interfaces per installation is approximately the maximum number of sites Sawmill will process. This maximum proves an obvious choice for the main hash size. The total combined log size is then evaluated to determine the approximate number of log entries. The number of entries is divided by main hash size and an estimate of the average session length. A factor of this dividend determines each session hash size.

To conclude the Log processing phase discussion, we will recap the flow of events. Sawmill begins to sequentially process logs in the mill. Each log entry is parsed into a chip. The domain and interface name are used to locate the correct chip bin in the main hash. That bin contains a session hash, where the chip is inserted into the timestamp ordered list composing that session.

Phase 9: Data Transfer

After the chips have been organized into bins in memory we can transfer this information to the database. Each bin, which holds data for a single database will be processed separately. As mentioned in the memory allocation phase, the bins will be further processed into database heaps used for batch mode updates and inserts into the target database tables.

Normally, the CGI updates the database. Sawmill eliminates the need for those updates by reproducing the data from the web logs. This phase handles that task by reproducing that data from the web log entry chips that have been prepared. Five tables need to be updated, which hold information such as counts of sessions, hits, content viewed, and searches performed, and when each occurred, as well as, session tracking and client activity analysis.

The phase begins by checking the main hash for valid data. As the logs were processed an overall minimum and maximum page request time was stored in the global data section. These values are checked to ensure actual log entries were encountered. If

so, processing continues. If not, a warning message is displayed and the process flow skips to the shutdown phase.

Next the data heaps are initialized. These are the heaps used for data transfer described in the memory allocation phase. Specifically, the phase designates portions of the general heap to be used for the transfer conduits. Each table receives its own conduit, which will fill at different rates. A full conduit is flushed to the database and then begins to refill as the chips are continually processed. Depending on site activity, most conduits will be filled multiple times per database. The conduits are reused throughout the data transfer stage and therefore must also be flushed as processing moves from database to database.

Each conduit is structured to the database table it will be updating. Some tables summarize their data per hour and have an additional layer of structure, and therefore the conduits were not initialized until the processing was completed. At this point, the minimum and maximum page requests are known, so the total number of hours being processed is also known. So these summarized table conduits are divided into hour sections where stats will be aggregated. As the chips are processed into these conduits, their data will need to be grouped with other chips that were requested during the same hour. By predefining the hour boundaries, simple arithmetic operations may be done on the request time to determine its bin, rather than needing an mktime to be preformed for each chip. Also because the data collected is summarized within the bins, these conduits only need to be flushed one per database, they do not fill like the others. The number of

bins is usually small, because although the number of databases is large, the number of hours being processed is usually small.

Data transfer is ready to begin, so the main hash is traversed. The main index is scanned sequentially stopping at each database entry to process its chip bin. The processing of each database begins with pulling its configuration information from the main hash. A connection to the database is established and the conduits are primed for transfer.

Now the session hash is traversed. It is sequentially scanned, evaluating each session and summarizing its data. From the processing phase, each session was built as a linked list of chips. The session is examined to collect data for the conduits, page views counters can be incremented by the requested page name, search text and answer id can be pulled from the parameter text, session length is determined from the timestamps, etc. References to these values were determined and stored within each chip during the processing page. So a small amount of extra storage space is used to hold references in order that the log entry text need only be touched once.

Once the session hash is completely traversed data collection for this database is complete. Any data left in the conduits are flushed and the connection to the database is terminated. Now traversal of the main hash continues on to the next database.

This phase is significant in error recovery. As each bin is processed, the sawmill.status file will be updated. Bin processing has five steps, one step for each table it will update. When a bin begins to be processed, the main hash index is inserted into the status file. As each step completes and is committed to the database, the status file is

again updated with the step number that has finished. The recovery mode can determine the exact database and step where the previous execution was halted. Therefore, it is able to pick up where it let off, without danger of updating a database twice or missing one all together. This concept is explained in detail in the next chapter on recovery implementation.

This phase also builds the recycled logs referred to in the log transfer and preparation phase. The recycled logs exist because of Sawmill's need to process user session with complete data. For example, the session length cannot be determined unless each entry in the session can be examined. Since the maximum length for a session is configurable and may differ from database to database, the length is obtained from the site info file and stored in the main hash.

Consider an example where Sawmill encounters a session that may not have been completed. It is the noon run, and processing encounters a session with page request timestamps of 11:54, 11:55, and 11:57. If that installation's session expiration is set to 10 minutes, that session could still be in use, the final page might occur at 12:01, which will be in the next set of logs, so Sawmill cannot update this information. Instead it recycles these chips by building up its own recycled logs with these half completed sessions in the mill's temporary directory. During the next execution, Sawmill will combine the recycled logs with the newly available logs. Thus, it will correctly see all four pages requested during this user session and update the database correctly.

The recycled logs are assembled in the mill's temporary directory for error recovery. When Sawmill loads, it combines the available logs with existing recycled logs

already in the mill. If recycled logs were built in the mill, a partially created log may exist from a previous run that exited in an error state. Since it exited in error, Sawmill might not have had the chance to clean up after itself, leaving the partial log. It would then inadvertently be read in with the real logs during the next run, causing databases to be corrupted. Therefore, the logs are built in the temporary directory and moved to the mill during shutdown, which only happens after a successful run. Partial logs that are left in the temp directory because of an error state are cleaned up in the next run's setup phase, which unlinks all files in the temporary directory.

Phase 10: Shutdown

Sawmill has completed all of its database tasks. Now a few important tasks are executed. First, the debarked logs residing in the mill are unlinked. They will never again need to proceed with another run. Note that in the log transfer stage, the original log was transported to the processed log location and will still exist in the obscure event the data would ever need to be reprocessed. Next, the recycled log file is closed and transported from its temporary directory to the mill.

To complete the cleanup, the sawmill.status file is unlinked, which effectively signals the subsequent run of Sawmill that the execution was successful. If you were to examine the file, it would contain a line for each entry in the main hash with a flag for each successfully completed step.

Finally, all system resources will be freed and any temporary data will be deleted, Sawmill exits with a successful error code.

CHAPTER 6

SAWMILL RECOVERY MODE IMPLEMENTATION

Sawmill's self-detecting recovery mode is designed to recover from nearly any fatal error on previous runs. Sawmill normally will be run as a scheduled task, running once per hour. An administrator may not be aware of a failure, so Sawmill must have the means to detect a failure on the previous run, correct the error, and continue with its scheduled run. However, a system administrator may have stopped Sawmill or known it to have halted unexpectedly and may rerun it immediately. In this case, the self-detecting recovery mode is a convenience, allowing the administrator to execute it without needing to move files, change configurations, or know any aspect of Sawmill's internal workings.

Error Detection

The third phase of Sawmill execution performs a status check. If the sawmill.status file is discovered the recovery mode is entered and the status file's contents are not examined. Depending on when the previous failure occurred, the file may be empty, complete, or somewhere in between.

Recovery Mode

When recovery mode is entered, Sawmill is in its third phase of execution. The first two phases have been completed verifying that everything is configured properly and

the needed system resources are in place to begin processing. Sawmill now deviates from the normal execution path by skipping the fourth phase, log transfer and preparation, as it will operate on the logs that currently exist in the mill because they have not been fully processed.

Note in the shutdown phase; logs in the mill are removed before the status file is unlinked to prevent an error condition. Consider a failure during shutdown while the processed logs were unlinking. If the status file did not exist, the next run would load the previously processed logs thinking them to be valid.

Sawmill continues on with normal execution, phases five through eight are carried out, where memory allocation, dynamic libraries and initialization are all done as usual. The processing phase operates on the unfinished logs in the mill. This completely reproduces a main hash and its associated data structures, creating a general heap that would have mirrored the failed run.

As the Data transfer phase is entered, Sawmill tests the need for recovery. If the file is empty, no data had been transferred during the failed run. The main hash is traversed in its entirety, as a normal run would. If the file contains data, Sawmill seeks to the last line which pinpoints the database and stage of transfer being processed during failure.

For example, it may have been at database B at step two. Sawmill skips to this index in the main hash. The step number in the status file was two, so the failure occurred during step three. This means the first two data conduits had been transferred to the database and the remaining three were left outstanding. Now the chips from session hash

are reprocessed into the last three conduits, and database B is updated with its missing data.

Any databases left untouched during the failed run must also be updated. Since the general heap is reproduced during the recovery mode and the main hash is traversed sequentially, entries before this index were processed before the failure, and entries after were untouched. So traversal of the main hash continues from this index completing the work of the failed run. As the recovery mode continues, its progress is appended to the status file emulating a normal run.

When the main hash has been traversed, ending the data transfer phase, Sawmill enters its final phase. Shutdown is carried out as usual, where the completely processed logs are removed and the full status file is unlinked. Thus, the recovery mode has been completed.

This may have been a normal scheduled run of Sawmill, and although it has now completed the work of the failed run, it has yet to do its own work. Therefore, execution loops back to the start. Phases one through three are unnecessary, configuration and setup have already been validated and an additional status check is not needed. Phase four begins in normal mode. If Sawmill was invoked manually, no logs will be available yet, so the process will exit. If it was a scheduled run, the logs will be available, transferred to the mill, and processing will proceed normally. When Sawmill finally exits, two complete runs will have been executed, the first in recovery mode and the second in normal mode.

CHAPTER 7

RESULTS AND CONCLUSIONS

The goals of Sawmill are to increase the throughput capacity of the current product architecture, and to provide a foundation for the second phase of the high performance implementation. As described later, this final architecture produces a potentially unlimited potential for linear expansion in processing capacity. Achieving these final goals was limited by the need to be cost effective, to limit changes to the current architecture and to conserve system resources.

This chapter will explain how each implementation goal was met and to what degree the primary performance goals were achieved.

Design Objective: Cost Effective Implementation

The implementation was to be cost effective in both development and hardware resources. Development cost is measured in terms of number of personnel needed to complete the project and the length of the research and development timeline. Given a fairly tight main product release schedule a fairly small budget was allocated. The project could take no more than six months from design to deployment. This includes research, implementation, testing and installation. The timeline was met with few major obstacles. Fortunately, development finished ahead of schedule giving the Quality Assurance engineers additional testing time.

Hardware cost is measured in terms of the actual cost of additional hardware needed for product installation. Since the solution implements on top of the current architecture, no new hardware was required.

Design Objective: Architectural Transparency

Architectural transparency allows developers concurrently working on the same areas to continue development without concern of merging code changes or the architecture of the high performance layer.

This goal was met with only a very minor change to the current product. A configuration variable was added specifying that Sawmill is enabled for that installation. Then a small change was made in the main CGI, checking the value of the new configuration variable. If enabled, statistic collection is disabled.

The benefits of architectural transparency impact cost effectiveness. An embedded solution would require major modification and even rewrites of modules composing the main product. This would interfere with the product's development cycle and it would force development to adapt to and merge with the embedded solution forcing the main release schedule to be delayed. The functionality would be tightly coupled, where changes in the high performance solution would require a change in the main product, and vice versa.

Architectural transparency has desirable benefits on development schedules and modularity. Since the solution is designed to be its own modular entity, it can be developed, tested, and deployed independently of the main product. Therefore,

enhancements and fixes can be dropped into the production environment on a separate release schedule, leaving the main product and the high performance product loosely coupled in terms of functionality. The high performance solution is a separate and optional transparent layer over the existing product implementation.

Design Objective: Execution efficiency

If components of the high performance solution require extended execution time and tie up shared resources, the primary goals of this project are defeated. The primary concern being the shared resources in the form of database access and processing cycles.

Sawmill will run on a dedicated utility server, which solely exists to execute all utilities; it does not host database or web server activity. The largest overall execution flow bottleneck revolves around the database servers, where data requests from the CGI, utilities, and external sources are served. So in comparison, the load average on the utility and web servers is lower than that on the data servers. Therefore, the primary objective is a low database load, but execution time is also important. All sites will not see their statistical data until Sawmill has completed its hourly run. If the execution time is long, there would be a large update delay. Consider a report scheduled to run near the start of each hour. Its usefulness may be diminished if it needed to be run half past the hour in order to wait for the statistical updates, at a minimum the wait would be undesirable.

A typical setup would be something on the order of dual gigahertz processors, with 2 gigabytes of RAM. Testing Sawmill on this environment yielded the following results.

Table 1. Sawmill execution statistics

	Execution time measured in seconds. Sample size 50 test runs. 0.5 gigabyte input size.			
Execution Phase	Average	Minimum	Maximum	Std. Dev.
Phases 1-8 (Config. – Init.)	5.7	4.6	7.8	0.48
Log Processing	3.8	3.2	6.0	0.41
Data Transfer	4.6	2.9	8.8	1.81
Total Execution	13.4	11.0	20.4	1.34

Since Sawmill runs hourly, the target execution time was to be less than two minutes, and the average execution time falls well under an acceptable run duration. A total log size of a half a gigabyte was chosen for testing, which is above the average hourly log size of 300 megabytes. Therefore, it is reasonable to assume this large dataset will present a fairly unbiased estimate of average execution speed.

The first two sections in execution breakdown, phases 1-8 and log processing, have fairly consistent execution duration with little deviation from the mean. This would be expected, as their most notable activities include log loading and processing. Both of these activities require a consistent amount of resources, linearly based on log size, where log loading is performing EMC disk access and log processing is burning CPU cycles.

Execution time of the data transfer process has more variance than the other phases, as the data transfer is the only phase affected by outside influences, the databases it will update. If they are heavily loaded or locked, the data transfer stage may be blocked for short periods of time waiting for resources to free. Since the data is summarized in the streamlined data conduits, transfer is quick once Sawmill finally gains access to the database.

High Performance Goal: Increase page processing capacity

The first overall goal of the high performance solution was to increase the page processing capacity of the product. In the background chapter we examined a case study of a site testing the limits of the product experiencing peak traffic rates over 10 requests per second, which is expected to at least double later in 2002.

The goal of increasing page processing capacity aims to satisfy popular sites, but also aids the overall performance of the hardware architecture. Most sites do not generate extremely high load on the database servers and therefore do not technically require a high performance solution. However, since increased processing capacity is achieved by eliminating conflicts between shared resources, the load average of the database servers is reduced. As the load average on a database server drops, additional databases can be added to the machine, cutting hardware costs in the hosting environment by increasing the number of databases per server.

To begin examining the success of the solution, we will take a look at the raw numbers in processing capacity. This test was setup and performed on a replicated hosting environment. Each page was tested by Apache Bench, a load-testing tool, to determine the number of times it could be generated by the software per second. The product has many total pages, but analyzing current customer data shows three to be the most visited. The most frequent is the Support Home page, which is the common starting page for an end user session. The next two are the Answer List and Answer Detail page, which respectively show the results of a knowledge base search and the content of a knowledge base item. These three pages receive on average seventy-five percent of a

site's activity. The remaining activity is relatively evenly distributed over the rest of the pages. The load testing was performed on identical sites with and without the high performance solution implemented to approximate the page processing capacity increase.

Table 2. Page Processing Capacity Results

	Average processing throughput measured in pages per second.		Pct. Increase
Page	Standard	High Performance	Pct. Increase
Support Home	11.76	17.62	66.7 %
Answer List	7.10	16.04	126.0 %
Answer Detail	8.16	22.12	171.2 %
All Pages	8.06	20.94	159.8 %

The results demonstrate significant improvements, with the average capacity to serve pages nearly being tripled. However, this test only shows the effects of eliminating the resource conflicts between end users. In a production environment, the shared resources are not only accessed by the end user, but also by administrators on the backend interface. Depending on size, site traffic, and knowledge base content, a customer may have one backend user or one hundred. End user activity is typical, where users browse the by searching for and viewing knowledge base content, but backend activity is highly variable. An administrator's view into the knowledge base is much more broad allowing analytical features such as reporting. Reporting and other administrative features may affect the end users because database resources will be concurrently requested by both sides of the product.

Incorporating load on the administrative interface into the processing capacity testing is a difficult task, due to its highly variable nature. Therefore, a more truly accurate test of the high performance solution's effectiveness would be to measure a

production database server's load average before and after the solution's deployment. Unfortunately at the current time, June 2002, this project is still in system test, so this data is unavailable. However, we can make some predictions based on the capacity testing performed on the end user interface. Not having the load average data is not detrimental in evaluating the solution's success. We can expect a greater percentage increase in processing capacity than what is shown in the results table because by not incorporating load on the administration interface, the processing capacity of the current product was overestimated. As mentioned above, the current product still shares resources between the end user and administrator. In some cases, the administrator actually locks out the end user for extended periods of time. An administrator may request a complex read from a shared resource, while the end user waits to write to the same resource during their page request processing. The end user perceives a slow turn around time, as their page generation was blocked for a period time.

So the test results overestimate the capacity of the current product by not including administrative load. They do show a good estimate of high performance processing capacity, because the solution eliminates conflicts between concurrent end users and between end users and administrators, with the percentage increase in page processing capacity is projected to be higher than that shown in the results table.

High Performance Goal: Foundation for Expansion

The second overall goal of this project was to provide a foundation for future phases of the high performance version of the product. The increase page processing

capacity solves the current performance problems facing high traffic installations. In the near future, as the customer base and the uses for the product grow, the performance bar will again be raised. The case study customer has already predicted page request volumes as high as two million per day, translating to 80 per second over the 10 peak hours each day. Although the first phase of high performance implementation nearly triples the throughput capacity, it still falls short of these expectations. However, it does provide the foundation for the next level of high performance processing.

We will take a look at how the future performance requirements will be achieved. For this section, we will consider everything implemented in this document as the first phase of the high performance solution, and the future work will be the second phase. The first phase lays very important groundwork for the next phase of implementation and in some regards it solved most of the difficult tasks.

Eliminating the contention between shared resources increases processing capacity. More specifically, the need for the client interface of the product to update the database is eliminated, effectively producing a read only interface.

Producing a read only interface is the key to the second phase of the high performance solution. The current solution provides an extended average throughput of almost 20 pages per second, only one-fourth future expectation. However, this is twenty pages per second from an environment where a single database server remains the bottleneck. The second phase of implementation will replicate the read only portions of the database required to serve the high traffic interface. Then each web server will receive one of the replicated database copies. When a page is requested, the dynamic

content will be entirely generated by the web server processing the request. Its database queries will be served from the replicated database onboard the web server, rather than the primary copy on the database server. Now each web server is producing twenty pages per second, achieving the future performance goals with four total web servers, which already is the minimum number of servers in the current hosting environment.

Twenty pages per second for each web server is only an estimate. The servers have additional processing responsibilities beyond database requests, such as actual CGI execution. However, since the database access is read only the efficiency of query processing may be slightly increased. The replicated copies could be configured to do dirty reads without attempting to enforce a read lock. It is not important whether the processing capacity drops to 15 per second or increases to 200. The result is that a high performance architecture has been produced which yields a linear increase in page processing capacity. Complete page generation has been localized to each web server. So, if a site needs 180 pages per second, and 30 pages per second is the proven capacity, then expand the cluster to six web servers. In practice, the page capacity per server will be somewhat important from a hardware cost perspective. Because the data store is read only, many options are available to optimize the data retrieval. These options ranging from DBMS optimization to query caching can be combined with CGI optimization techniques to provide a very high page processing capacity per server.

The first phase could not increase by replicating the database. If it were replicated without being read only, updates to each duplicate would need to be propagated to all the other copies. Therefore, the total number of updates each server receives is the same as

the database in our original problem. This does not produce an increase in throughput. Even if a replication scheme could be worked out to increase the throughput, it would not have been implemented in the first phase. It defeats one of the primary design objectives of providing a cost effective solution. Replication inherently takes away the ability of a single set of servers to serve a large set of customers. Whether the web servers hosted the replicated databases or another set of servers, the hardware cost per customer has significantly increased. The current implementation of the first phase allows for an across the board solution to all customers on the existing hardware. The second phase's implementation may require additional hardware, but it will be reserved for the small set of customers who will have extreme requirements in processing capacity.

Much work still needs to be done in planning the second phase. The means to linearly increase processing capacity has been achieved, but the capacity per server can always be refined. After distributing the replicated databases, performance will need to be reevaluated to determine the new bottlenecks in processing. It may still revolve around database activity, but weight will shift towards CGI processing as a new shared resource has been added, the CPUs of the web servers. Preliminary evaluation in this area reveals much can be done to aid CGI processing. For example, script compiling and caching, shared object caching, and work around database connectivity. In fact enough work exists to propose a third phase in the high performance product implementation.

In conclusion, the first phase of high performance implementation has succeeded in providing a foundation for future work. The second phase will proceed with the same design practices exercised in the first. First, provide a cost effective solution, both in

terms of development hours and now to a lesser extent, hardware resources. Second, provide a solution that is efficient in execution, including the solution itself and any utilities designed to aid its implementation. Third, provide architectural transparency, it will be implemented as a mutually exclusive layer over the first, and of course will still be transparent to the lowest layer, the core product.

REFERENCES

[1] V. Padmanabhan, L. Qiu. The Content and Access Dynamics of a Busy Web Site: Findings and Implications. 2000.

[2] P. Atzeni, G. Mecca, and P. Merialdo. Design and Maintenance of Data-Intensive Web Sites. 1997.

[3] C. Bouras and A. Konidaris. Computer Engineering and Informatics. Modeling Data Intensive Web Sites for Personalization, Integrity and Performance. 2000.

[4] E. Casalicchio and M. Colajanni. Scalable Web Clusters with Static and Dynamic Contents. 2000.

[5] V. Cardellini, E. Casalicchio and M. Colajanni. A Performance Study of Distributed Architectures for the Quality of Web Services. 2001.

[6] L. Degenaro, A.Iyengar, I. Lipkind and I. Rouvellou. A Middleware System Which Intelligently Caches Query Results. 2000.

[7] D. Florescu, A.Levy and D. Suciu. Optimization of Run-time Management of Data Intensive Web Sites. *Proceedings, 25th International Conference on Very Large Data Bases.* 1999.

[8] A. Iyengar and J. Challenger. IBM Research Division. Improving Web Server Performance by Caching Dynamic Data,1997.

[9] A. Iyengar, J. Challenger and P. Dantzig IBM Research Division. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites, 1998.

[10] K. Kang Sang, H. Son, J. Stankovic and T. Abdelzaher. Department of Computer Science University of Virginia. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. 2001.

[11] E. Markatos. Institute of Computer Science, Foundation for Research & Technology. On Caching Search Engine Results. 2000.

[12] Y. Lu and R. Sunderraman. Department of Computer Science Georgia. Application-level Caching for Web-based Database Access. 2000.

[13] J. Song. Design Alternatives for Scalable Web Server Accelerators. *ISIT: Proceedings IEEE International Symposium on Information Theory, sponsored by The Information Theory Society of The Institute of Electrical and Electronic Engineers.* 2000.

[13] K. Yagoub, D. Florescu, V. Issarny and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. *The VLDB Journal.* 2000.

APPENDICES

APPENDIX A

REQUIREMENTS

The following is a list of requirements Sawmill needs to run:

- mod_sawmill - a custom built apache module designed to modify the url format before logging to aid in post processing
- Custom Log Format – a custom log format needs to be specified in the apache configuration file.
- Log Rotation – The logs from each web server need to be rotated off into a common place once an hour.
- CGI Preparation – Changes were made to the CGI to allow it to not collect data if Sawmill is enabled.
- Sawmill Environment
    - DLLS – Sawmill will require both a MySQL and Oracle version of libcmnapi.so. Both in a common dll directory.
    - sawmill.conf – The Sawmill configuration file.
    - siteinfo.conf – Site configuration information needed for Sawmill to operate on each interface of each site enabled.
    - Sawmill Executable
    - tr.sawmill – optional tracing

APPENDIX B

MOD_SAWMILL

Mod_sawmill is the apache module to be installed on each web server. Once installed, it will need to be enabled in the httpd.conf file of each server. Additionally, a directive will be added for each interface Sawmill is enabled. The directive format is:

```
SawmillEnabled  domain_name  interface_name session_exp_len
```

Example:

```
SawmillEnabled  abc.com  abc_int1 60
```

APPENDIX C

CUSTOM LOG FORMAT

The log format will contain all of the information needed to reproduce the statistics needed to update the database. The new custom log format specified in Apache's httpd.conf would be:

```
LogFormat "%v %h %u %t %{%s}t %B %T %>s \"%r\" \"%{User-Agent}i\"" sm_log
```

The meaning of the fields are as follows:

%v - Virtual Host requested

%h - Requesting address

%u - Username, if logged in via apache authentication

%t - Human readable date-time

%{%s}t - Epoch time

%B - Bytes transferred

%T - Time to serve request

%>s - Status of request returned by the web server

%r - In quotes, the first line of the client request

%{User-Agent}i - In quotes, the user-agent

'LogFormat' is the directive name. 'sm_log' is the log format name.

APPENDIX D

LOG ROTATION

Log rotation will be done once an hour. Sawmill will require one log for each web server for each hour. The following directive must be also added to Apache httpd.conf file to aid log rotation:

```
CustomLog
"|/usr/local/sbin/cronolog/var/log/httpd/sawmill.websc01.%Y%m%d%H.log"
sm_log
```

This will name the log in the form samill.webserver.YYYYMMDDHH.log. cronolog is a script provided by hosting that will help move the logs to a common place (directory on NFS).

The Log Rotation needs to precede the running of Sawmill each hour.

APPENDIX E

CGI PREPARATION

Since Sawmill will be doing the data collection, the CGI will not need to do so. A config verb mod_sawmill_enabled has been added. If this is set to true, then Sawmill will process the stats belonging to that interface and CGI will skip the data collection steps during execution. If it is set to false, Sawmill will ignore this interface and data will be updated as it is done currently, during page execution.

There are two main types of HTML requests: GET and POST. A GET request places all of the page parameters (p_sid, p_faq_id, etc) in the url. This is important because this means apache will see the parameters when the page is requested, meaning it can add them to the log. Sawmill will process all these requests.

When a POST page is requested, the parameters are sent after the page request. POSTs are necessary and usually reserved for pages that need to submit large amounts of data because there is a maximum url length accepted by most web servers. Apache won't be able to log the POST parameters, therefore Sawmill will not be able to process them. So, in order to collect this data, the CGI must always do the updates in these special cases.

Most of the end user pages are GET requests, however, a few accept large data fields and file attachments and therefore are POSTs, but are very infrequently viewed and therefore will not pose an efficiency problem.

APPENDIX F

SAWMILL ENVIRONMENT

Sawmill is unlike most utilities. The main difference being each site will not have its own executable. There will be one execution that will update all enabled sites on the hosted servers. Another difference is that Sawmill will be required to make multiple database connections, to multiple types of databases.

In order to run, Sawmill needs two configuration files and a set of dlls. The configuration files need to be in Sawmill's working directory and the dlls need to be grouped together in the same directory.

For the following descriptions, let's consider Sawmill's working directory to be `/home/sawmill` on one of the web servers. In reality, Sawmill could be on any machine that has access to the database servers.

dlls

Example location:

```
/home/sawmill/dll/libcmnapi.5.x.x.so
/home/sawmill/dll/libcmnapi.oci.so
```

Because Sawmill will run over multiple sites during one execution, it may encounter stats for both MySQL and Oracle databases. Therefore both versions of libcmnapi.so are needed.

The MySQL version is expected to be named with the current version convention, for example, libcmnapi.5.0.0.so. The Oracle version will obviously need to be renamed as to not have a conflict, the name expected is 'libcmnapi.oci.so'.

Before running Sawmill, the shell LD_LIBRARY_PATH environment variable should be set to the example dll path.

Samill.conf

Location: /home/sawmill/sawmill.conf

The sawmill.conf file has 4 directives that tell Sawmill how to run and where to look for data. An example conf would be as follows.

```
WebServers      6
AvailableLogs   /nfs/logs/sawmill
ProcessedLogs   /nfs/logs/processed
TheMill         /home/sawill/themill
```

Direction Descriptions:

1) WebServers - Number of web servers in the web-balanced environment. Sawmill will expect one log per server per hour.
2) AvailableLogs - Path to rotated web logs. Each log will be accumulated locally on each web server, but the rotation script will move them to a common location on NFS. The common location is the correct value for this directive.
3) ProcessedLogs – Where to move the logs once they have been processed.

4) TheMill – Sawmill's data directory. If this path does not exist, Sawmill will try to create it. Sawmill will first move the available logs to this location, moving them to the processed dir only after successful completion. Sawmill will also build its own logs that will be stored here. If the value of this directive is changed, the contents of the old path must be moved also to avoid data loss.

Siteinfo.conf

When Sawmill executes it will need to know various site-specific information about databases it will eventually update. Currently utilities run from a change-rooted environment, which prevents access to other sites' data. Sawmill would be unable to run given this constraint, because it would need to access configuration data for multiple sites during a single execution. This conf file is needed to provide the minimal site-specific information needed to update each enabled database.

A single entry in the file would be as follows:

```
abc.com;abc_int1;MST;EST;60;1;62.197.4.1;1;dbqa01;abc_db;root;foo
;
```

There are 12 or 14 required items depending on the target entries database type. Each enabled domain/interface combination will require an entry.

A description of each:

1) Domain name

2) Interface name

3) Time Zone Interface - corresponds to TZ_INTERFACE.

4) Time Zone Database – corresponds to TZ_DATABASE.

5) Session Expiration Length – MYSEC_SESSION_EXP, rnw_ui.

6) Sawmill Enabled – MOD_SAWMILL_ENABLED (1 = yes, 0 = no).

7) Excluded Admin hosts – MYSEC_EU_ADMIN_HOSTS, rnw_ui.

8) Database Types (1 = MySQL, 0 = Oracle).

If database type is MySQL:

9) Database Server - DB_SERVER

10) Database Name – DB_NAME

11) Database Login – DB_LOGIN

12) Database Password – DB_PASSWD

If database type is Oracle:

9) Database Login – DB_LOGIN

10) Database Password – DB_PASSWD

11) Oracle Home – ORACLE_HOME

12) Oracle Service – ORACLE_SERVICE

13) Oracle SID – ORACLE_SID

14) Oracle NLS – ORACLE_NLS

tr.sawmill

To turn on the optional log files, set up the trout file as with other utilities.

```
$ cd /home/sawmill
$ echo ALL > tr.sawmill
```

Note the trout file will be quite large, so this option will slow execution down. The files also will take up large amounts of disk if left around.

Areas Affected

Sawmill is required to maintain the integrity of all areas of data collection previously done by the CGI during page execution. This results in the updating of 5 tables:

1) Clicktrack – all pages hits and tracked parameters will be inserted.
2) faq_stats – faq stats rows will be inserted, or updated if they exist, for all answered viewed.
3) Faqs – the solved_cnt, solved_1, solved_2,… solved_5 values will be updated for faqs viewed during the session.
4) Stats – The number sessions, hits, searches, and faqs_viewed will be updated in the stats table.

5) Links – links will be added for new to/from paths discovered, and strengths will be updated for existing paths revisited.

APPENDIX G

RUNNING SAWMILL

Running Sawmill can be done two ways. The first, the standard run, is to just type 'sawmill' in the sawmill working directory. Note, this would be after setting the LD_LIBRARY_PATH environment variable. This tells Sawmill to run in normal mode.

The second is 'sawmill –f', or force mode. This will force Sawmill to run on any logs available. In other words, skip the check to make sure there is one log for every web server for the current hour.

After setting up the Sawmill environment as described in the previous section, a typical Sawmill run would go as follows:

```
$ export LD_LIBRARY_PATH = /home/sawmill/dlls
$ cd /home/sawmill
$ ./sawmill
Checking Paths …
Initializing Sawmill …
Running Sawmill …
Running Sawmill …
Shutdown …
$
```

APPENDIX H

RUNNING SAWMILL – RECOVERY MODE

Sawmill's robust recovery mode is designed to recover from nearly any fatal error on previous runs. It is designed to be self-detecting and require no manual intervention. The output will look as follows:

```
$ export LD_LIBRARY_PATH = /home/sawmill/dlls
$ cd /home/sawmill
$ ./sawmill
Checking Paths …
Initializing Sawmill (Recovery Mode) …
Running Sawmill …
Running Sawmill …
Shutdown …
Initializing Sawmill …
Running Sawmill …
Running Sawmill …
Shutdown …
$
```

Note, it first initializes in recovery mode, and completes any unfinished tasks. So it would reprocess the debarked logs in The Mill and start where the previous run left off. Once this task is complete, it runs in normal mode, meaning it looks at the available logs and proceeds with normal execution as described in Chapter 5.

MONTANA STATE UNIVERSITY - BOZEMAN
3 1762 10363309 3