Pachyderm: The Web Proxy That Never Forgets

Haonan Tan Fujun Zhang

Abstract

Pachyderm is a web proxy that offers more functionalities than traditional proxy caches do. It provides web object versioning and per-user browsing history so that users can access multiple "back issues" of a single web object. In the mean time, user advice is used for cache management as well as system stats. This eliminates the gap between user and system and helps maximize the cache utilization. Instead of building the cache directly atop a file system, we use a commodity SQL database as the cache storage backbone. Given our testing workload, it offers good performance in both basic cache performance metrics and its unique functionality measurements.

1 Introduction

A proxy cache (Figure 1) is a proxy server that sits near the network gateway and implements web caching. Web caching provides a temporary storage of web objects (HTML documents, images, style sheets and so on) for later retrieval. The advantage of a proxy cache is significant. First, it reduces bandwidth consumption. With more requests that can be served from the cache locally, fewer requests have to go out onto the Internet. This is a benefit for the network infrastructure. Second, it alleviates the load of web servers. The proxy cache can often times cover up for the actual sever so that the server does not have to serve every client request. Third, on the client side, a lower latency can be expected. This becomes particularly effective when the cache holds some web components that requires a long-haul retrieval otherwise. Fourth, a cache can isolate end-users from network failures. Even though the source server might be unreachable, its objects might still be accessible from the cache. More and more ISP's these days start using proxy caches for all the reasons mentioned above.

One might argue that a proxy cache seems to be redundant because many web applications (say web browsers) today already have their own caches. That is not true though. One reason is that usually application cache capacity is insignificant compared to that of a proxy cache server. Another reason is that application caches are more application-specific and user-specific. In other words, it discourages sharing and has relatively low cache utilization. It is obvious that a proxy cache is a proxy in the first place. And literally it can do everything that a common proxy can do, filtering, monitoring and whatever. Moreover, sometimes a proxy is mandatory. Many firewalls disallow direct connection to web servers and there must be some application layer to relay data between clients and servers.

Our work tries to go beyond the existing proxy caching mechanisms and we have sketched out a prototype for the possibly next-generation proxy cache. Our goal is to support caching multiple versions of a same web entity. As the disk gets large and cheap, we worry less about the cache capacity and versioning becomes affordable. The same argument has been used in Elephant file system [3]. This makes it much

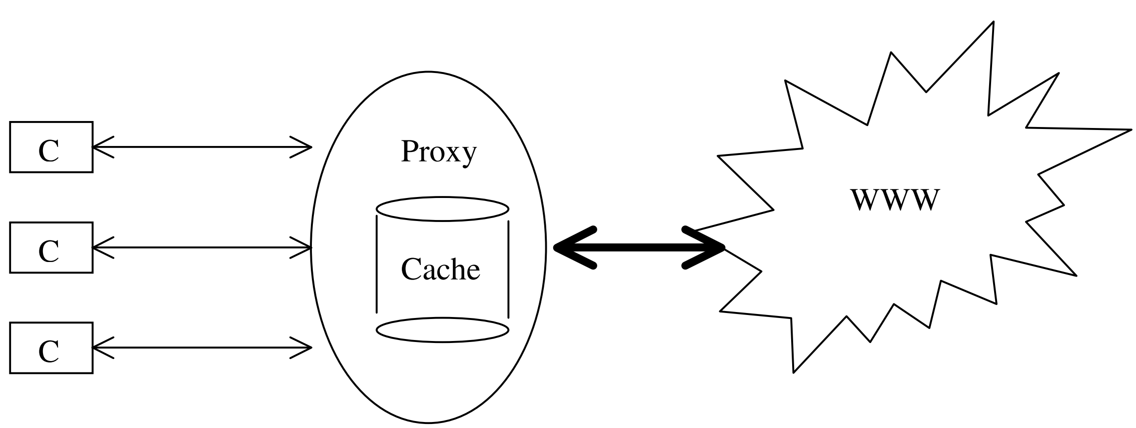

Figure 1. The role of a proxy cache

easier to keep track of web content updates. Another goal is to make the proxy cache more user-friendly. With versioning, the proxy cache is no longer transparent to users but this enables it to provide better services. Pachyderm is such a web proxy that can serve as a personalized web content manager while preserve content sharing. Also, cache management can be aided by user knowledge to achieve higher efficiency.

Here is the basic layout of this paper: Section 2 is an overview on various proxy caches; Section 3 is a description of our methodology; Section 4 gives details on how we implemented Pachyderm; Section 5 reports the results of our performance evaluation; Section 6 is for related work; Section 7 describes some possible future work; Section 8 concludes this paper.

2 A Brief Survey on Proxy Cache

In this section, we give a quick glance over some of the proxy caches available today. In our study, this helped us to probe the problem in a broad scope and to understand the functionality of a proxy cache better. This also gave us a guideline on how to distinguish Pachyderm from all the other products in the market.

2.1 Commercial vs. Non-commercial Products

Proxy cache development is active in both commercial and non-commercial areas. Commercial proxy caches are abundant these days. Some of them can be more hardware specific. For example, Cisco Cache Engine [7] is almost always paired with Cisco Router/Switch. Another example is F5's EDGE-FX Cache [8], which provides good integration with its own network traffic management controller. More and more manufacturers are making proxy cache as a part of their product line.

The most successful non-commercial instance of proxy cache must be Squid [14]. Derived from Harvest Project (no longer active) [4], it is a full-featured proxy cache that supports cache hierarchy with various inter-cache protocols. Funded by the National Science Foundation, it is a freely available and open-source software.

2.2 Attached Caching Modules vs. Standalone Server Packages

Some proxy caches simply come in the form of a module among a service bundle. The well-known Apache web server [6] has such a module. But more proxy caches are standalone servers, such as Viking [12] and Microsoft Proxy Server [10]. Some developers offer both solutions. SoftRouter Plus [15], a product from Vicomsoft, has its own caching module. On the other hand, Vicomsoft also announced RapidCache, a Windows-based dedicated proxy cache. Cache modules are usually simpler and offer fewer features than dedicated proxy caches do.

2.3 Proxy Caches vs. Server Accelerators

Proxy caches sit closer to the clients on the network. Applying the same web caching idea on the server side, we can have server accelerators, also known as reverse proxies. A server accelerator can greatly ease the server load and reduce server response time, especially when the server has to generate large amount of web contents dynamically. SpiderCache [13] is a member in this family. Indeed, there are products that can do both depending on their configurations. Examples are the WebSphere Edge Server [5] from IBM and Netplicator from Unitech Networks [11].

Server accelerators, however, are beyond the scope of our discussion in this paper. Pachyderm emphasizes more on client side caching.

3 Methodology

Various forms as today's proxy caches have, they share some properties in common: First, Caching behavior is transparent to users. Users are never aware of the existence of the proxy cache (except when they set up the browser configurations to use the proxy). The good side of it is that the original client/server illusion is not affected at all and users have less to worry about. The downside would be there is no effective way for a user to control the cache and adjust it more towards his interest. Not much about user friendliness can be done in a traditional proxy cache system.

Second, at most one copy of a certain URL can be cached. When a user sends out a request, the cache will search its repository to see if it has the requested URL. If this is true, we have a cache hit and the cached object will be served. Otherwise the proxy cache will forward the request to the genuine web server and cache the result. But sometimes even if a hit will occur, a user can issue a "refresh" to force an up-to-dated copy to be loaded. Or the proxy itself believes the cached copy is too old to be used again. On these occasions, the proxy fetches the document from the outside server and overwrites the possibly stale copy in its cache. Nevertheless, staled information is not necessarily useless. We would rather use the name "back issues" when referring to it. This is a good analogy between web publishing and traditional publishing.

Third, largely due to user-transparent caching, cache management is totally taken care of by the system. The proxy keeps track of certain cache reference stats such as the age of web components and the access frequency. When the cache is full (or nearly full), the proxy cache uses certain algorithm (usually some LRU-variant) to decide what to be removed out of the cache to make room for the incoming new objects. This is somewhat effective but can certainly be more effective if the system has some knowledge about how the user will actually make references in the future.

Given these considerations, we decided to extend the functionality of a proxy cache so that users can have more control over the cache and the cache management can be more efficient. The key is to add an information path between the proxy and the user. This will have to break the scheme of transparency, but the gain could be much bigger. Here are the unique features that Pachyderm can offer.

First of all, based on the large-disk assumption, Pachyderm can hold multiple copies of a single URL. It also records per-user reference information: "Who visits where, when?" Essentially, this provides a per-user browsing history. A user can manage his back issues by himself or tell Pachyderm how to manage them. This is done within the Pachyderm user interface.

In addition, since the user knows the URL's that he has visited better than the server does, he can pass this knowledge to the server to make cache management more future-oriented to achieve higher efficiency. Such user-assisted management has already been used in some file systems [2]. Besides implicit information such as system stats, Pachyderm now has explicit information from users. This will help improve the cache utilization.

Moreover, per-user prefetch schedule can be used to address cache compulsory misses, although this is not implemented in the current version of our prototype. Also, enhanced features like search on back issues become possible to improve file manageability.

4 Implementation Details

Our prototype consists of three functional units: server, cache storage and user interface. In this section we examine each part.

4.1 Server

The server is our control center. All the user requests are handled here. The server requires user authentication so that it can tell apart requests from different users and record them to the browsing history accordingly. It also performs multiplexing between the cache service and true web service. Our server is implemented in Java. This is a multi-threaded server. The main thread simply waits for a client connection. Whenever a connection comes in, it spawns a child thread to handle that connection, then gets back to wait for the next connection. This enables the system to handle multiple requests at a time thus increases parallelism. For simplicity, we only support GET method in HTTP protocol at this moment. From our own experience, Java makes it easier to program a web application. Our major concern at the coding stage was about the Java virtual machine overhead. It turns out to be acceptable and we will see that in the evaluation section.

4.2 Cache Storage

Most proxy caches today build their cache storage atop the file system. Our decision is to use a commodity SQL database to serve as the cache storage. Compared to a solution using file system, it has advantages as well as disadvantages.

The obvious disadvantage is the extra database management software required. However, from the other side, this can reduce development cycle significantly. Another concern is that we cannot communicate with a DBMS without specific API and this additional layer is likely to incur performance degradation. We will address this issue later.

Because web documents are part of the file system in nature, it would be easier to preserve the original directory hierarchy if we use a file system based cache. But in fact, not many proxy cache implementations (if there are any) choose to mirror the directory tree. On the contrary, many of them rearrange the file layout for faster access. For example, Squid builds a certain number of directories on the disk, each with more subdirectories. It uses a hash function to dispatch web objects into different directories and subdirectories according to the URL. It maintains a hash index in the main memory. Therefore, to look up for a certain URL, Squid can save the number of I/O operations by effectively using the in-memory hash index because its own directory structure is a "fat" tree with only two levels and this "shallow" structure helps reduce the translation time used by operation such as *namei*. Nonetheless, all optimization like this is merely an imitation to the database *B+* tree indexing, and chances are database can do it in a more effective way. That is one of the main reasons why we chose a DBMS as the storage backbone.

By converting a web object (or a web file) into a database record, we are able to assign meta-data other than specified by the file system by adding more fields to the record. This would not be trivial for a cache that is built directly upon a file system. Among these meta-data, version information might be the most important attribute in Pachyderm. Moreover, database can well take care of all the queries, either on the meta-data or on contents of the web documents. Consequently, some enhanced features such as search on back issues become much more straightforward within the context of a DBMS. Furthermore, besides the web objects to cache, we can store user information in this database as well. The user profile includes the user's account information, a personal browsing history and personal preferences on web objects. This integration provides a centralized management environment and makes the administrator's life a bit easier. Last but not the least, since SQL has become such a standard, it would be much easier for others to understand our work. Because we will make Pachyderm open-source and freely available, it is very likely that using SQL can greatly shorten the learning curve if some third party decides to make its own contribution to the software.

In practice, we used MySQL as our database server. There is some tradeoff though. It is free for non-commercial use. It is easy to install, configure and use. Managing a MySQL database is simple and intuitive. It has a large number of built-in functions so that users generally do not have to define their own. Also, the performance of MySQL is quite impressive.

The downside is that some important features are missing in MySQL, such as referential integrity checks, foreign keys, transactions, views and triggers, each of which is more or less relevant to Pachyderm implementation. We could have chosen PostgreSQL, another open-source DBMS that addresses these problems well. Anyhow, the decision is not critical in that we have decoupled the proxy server and the storage. The server will use JDBC to communicate with the database and it does not really care who is behind the scene. As far as Pachyderm prototype is concerned, we can get around these problems relatively easily. For instance, transaction support is important when the cache deals with concurrent client requests. A transaction can make sure that such handling has "ACID" properties [1] (atomicity, consistency, isolation and durability). MySQL uses only table locks to ensure mutual exclusiveness. It certainly makes it difficult in terms of recovery. But we can assume that a small amount of information loss is tolerable for web browsing activities.

Our database consists of four tables.

Account: This table represents the user entity. It holds personal data such as user id, login name and password. It is mainly used for user authentication.

Cache: Each web object is stored in this table. So this is indeed our central storage. In our current implementation, this is a big flat storage space. Each object is identified by its URL string and a version number. A version number is a timestamp indicating when the object was brought into the cache. The granularity of the version is down to one second. This should be adequate (if not too fine-grained) for web components. By default, when a cache hit occurs, the server always serves the latest version of the requested URL. To access a back issue, the user can either go to the Pachyderm user interface or add a special version request string after the URL when they open a location using a browser. The object content is stored in a BLOB (binary large object) field. Sometimes if an object is too large to fit into a single record, multiple records will be used to hold that object in segments (each segment is 64KB in our implementation). To preserve the order among segments, a sequence number has been used as another field in the record. In other words, (URL, version, sequence) constitutes the primary key of this table.

History: This is where we store user browsing history that answers the question “Who visited where, when?” This is a relationship between the user and web objects. Therefore, the user id, URL string and the access time is needed. To differentiate which exact version the user referred to, the version number should appear as well. When the user looks at his personal browsing history, the history is actually “selected” by the underlying database from this one big table according to the user id.

Preferences: This is a relationship representing the user’s knowledge about the URL. The fields are user id, URL string and one or more option fields depending on how complicated the preferences would be. Thus, for a single URL, preferences could vary greatly across different users.

Since we use JDBC for communication between the server and the database, we are curious about how the performance is affected with this additional API layer. This will be shown in the evaluation section.

4.3 User Interface

The most natural and convenient way for a user to access the user interface would be to use a browser so that the user interaction with Pachyderm can be integrated seamlessly with web surfing.

A common technique to achieve this is to use CGI programming. Or, if we want it to be more lightweight, Java servlet is also a good candidate. In practice, we used a CGI script written in Python (greatly due to the fact that we are never good at Perl scripting). Similar to JDBC, Python itself has an API module so that it can talk to MySQL database.

The script resides on a well-known URL for easy access. Because browsing history is more of a privacy issue, the interface requires user authentication. Once a user enters the Pachyderm user screen, everything is non-cacheable. This eliminates some possible complexity and confusion. Below are the features supported by our user interface at the moment.

History management: A user will see a list of URL’s accompanied by the time when he accessed them. He can browse these back issues. Also if he no longer cares about some back issues in the history, he can delete those so that they will not show up again. Note that this kind of deletion only modifies the “History” table instead of deleting the corresponding web object in the “Cache” table.

Preference configuration: This is where the user gives his advice to the system: how often this web site is updated, how frequently he actually visits that web page, how valuable the back issues from this web site are, or how many back issues of that web page he particularly wants to keep. Some information can help the system decide what to keep in the cache. Other information might even enable the proxy cache to download the updated web document in advance to avoid compulsory cache misses. Although not implemented in the current version of Pachyderm, per-user based prefetch schedule at least sounds like a promising feature of the proxy cache. Right now in our prototype, user advice only takes place in its simplest form, a one-digit number indicating user priority of the web object. This scheme is rather coarse-grained and not very user-friendly. However, we see no baffling obstacles that prevent further extension to this mechanism.

Back issue query: This functionality enables users to fully exploit their back issues. It is a personal search engine in contrast to those online search engine giants. Since its search scope is limited to the documents actually visited by the user, from time to time it can serve user’s interest better. We now provide keyword search with access date constraints. A user can specify one or more keywords in a query. The implied logical operation for multiple keywords will be an “and”. It is also possible to limit the searching scope within the documents visited over a certain period, by supplying the beginning and ending date of that period.

To summarize, the user interface is responsible for system-user interaction. It is the bridge between these two parties.

4.4 Cache Management

In this section we elaborate a bit on cache management. This is mainly about cache replacement policies. Ideally, this should never happen because we have claimed that our storage is so large that it can never be filled up. To be more realistic, however, this is bound to happen if we put the system to use. There is no way for us to have infinitely large disks and some upper bound must exist in a system. Huge storage as we can provide, there is no guarantee that the user cannot come up with a way to consume it all. Therefore, replacement policy is still worth discussion and this is similar to the situation in Elephant File System [3].

4.4.1 User Assistance

User preferences on web objects are used to form a user voting system. The overall importance of a web element that users attach to it can be determined from all users' votes. Therefore, an object is likely to survive in the cache when most of the users believe it is important, so this is a "hot" page. Those pages that serve only personal interests are more ready to be swapped out. The goal here is to maximize the cache utilization.

4.4.2 System Judgment

Under some circumstances, user knowledge could be imprecise. This usually happens when a user overestimates the priority of a document, or simply because the user is irresponsible. Therefore Pachyderm still needs system stats to study the actual use pattern. The source of information is richer for Pachyderm due to the existence of the user interface. Besides file access time and access counter, Pachyderm can collect query-oriented stats (has not yet been incorporated in our prototype): hit-by-query counter, most-recently-queried-time, matching score on queries and so forth. A synthetic system priority can be obtained from these stats. Generally speaking, the more frequently a document is viewed or searched by users, the more likely Pachyderm will keep it in the cache.

4.4.3 System Cleaner

The cleaner is a process that picks candidates for deletion from the cache storage. It can either be activated periodically when the system load is relatively low (usually over late night or weekend) or triggered by the proxy when it believes the watermark is too high. In practice, both should be used to complement each other. Currently, we only implemented the first scheme. Every time the cleaner launches, it recalculates the overall priority of each web object if there is a need for cleaning. Right now, we set the watermark at 95% of the cache capacity and the cache needs cleaning if this watermark is reached. After priority recalculation, beginning with the objects with the lowest priority, it deletes up to 15% of the cache content. Similar strategy has been used in Squid and Microsoft Internet Explorer.

This sounds to be a reasonable solution, but one tough decision that system administrators have to make is how to achieve a balance between the user votes and the system judgment. No wonder there is a family of policies we are discussing here. Lots of experiments need to be done before we can find the appropriate weights for each factor. But the mechanism itself is not unfamiliar [2].

5 Performance Evaluation

As a web caching system, there are various aspects about Pachyderm we would like to investigate. Some are relatively easy to obtain while others are more difficult within the current development context. In this section, we will list our interests along with the reasons why we are interested. We will also give some explanations if we feel that some aspects cannot be very well addressed. The results will be presented afterward. But the first thing we have to do is select one or two existing proxy caches to make comparison possible.

5.1 Reference Systems

Squid: This full-fledged proxy cache supports many web protocols such as HTTP, FTP plus SSL. Its support for inter-cache protocols (ICP, HTCP, CARP, Cache Digests) makes it ideal for building cache hierarchies.

WWWOFFLE (short for WWW Offline Explorer) [9]: This is a personal contribution from Andrew Bishop, starting from 1996, currently at version 2.6. It is noticeable that versioning had already been introduced in this system. It can keep up to 3 versions for the same URL and the user is able to browse the cached content. Nevertheless, it is only suitable for individual user in that the proxy cache does not differentiate requests between users and every item in the cache pool can be accessed by other people using the system.

In a sense, it is difficult to find a system that is highly comparable to Pachyderm. Squid is appropriate for large-scale use and WWWOFFLE is handy on a personal computer. Pachyderm seems to sit somewhere between these two extremes, probably closer to WWWOFFLE.

5.2 Points to Ponder On

The *cache hit latency* determines how much the clients can benefit from using a proxy cache. We can compare it with the *cache miss penalty* to see the performance gain we can actually have. Another important measure of cache performance is the hit rate. Usually it can be obtained via trace study or test under real workload. But neither was feasible when we developed Pachyderm. It is also difficult to define a fair comparison in terms of the cache capacity because Pachyderm keeps more back issues. So we left that out for the time being.

Another issue is scalability. It is a two-fold question. First, what if the cached content gets large in volume? Second, what if there are more users linked into the system? We cannot afford a multi-user test at the moment and we will focus on the first question.

Needless to say that the back issue query feature is helpful. But how about its performance? If it costs too much time, users might not even bother to use it.

It is well known that a database depends heavily on its indexes to achieve efficiency. With indexing, a large number of disk operations can be spared. But to what extent does indexing benefit a web proxy system like Pachyderm?

We have a few questions regarding the cleaner as well. How often does the cache need to be cleaned? How expensive is a clean operation after all? The first question depends on how big the cache is and how aggressive our users are. There is no easy answer. We will discuss the expense of the cleaning process in the following subsection.

5.3 Results

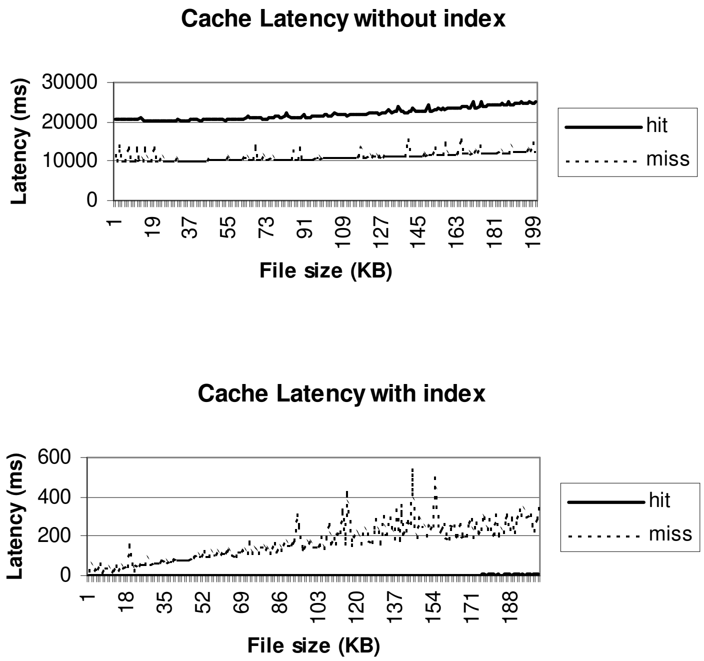

Figure 2 Hit/miss latency for various file sizes when cache almost empty

Our testing platform was an Intel Pentium Celeron 533 MHz running Debian Linux 2.2 with 128 Mb RAM. Figure 2 shows the impact of indexing. We sent requests for web documents with size varying from 1 Kb to 200 Kb (most web objects fall in this range) and timed the latency. Since the server needs to do lookups on the "Cache" table, two indexes are involved, one on URL, the other on version. As we can see, indexes can lower the latency by orders of magnitude. An interesting observation is that hit latency is longer than miss penalty when the table is not indexed. This is because when the content is served from cache, the index on version will be used if it exists. Without this index, the process will be slow. But if a

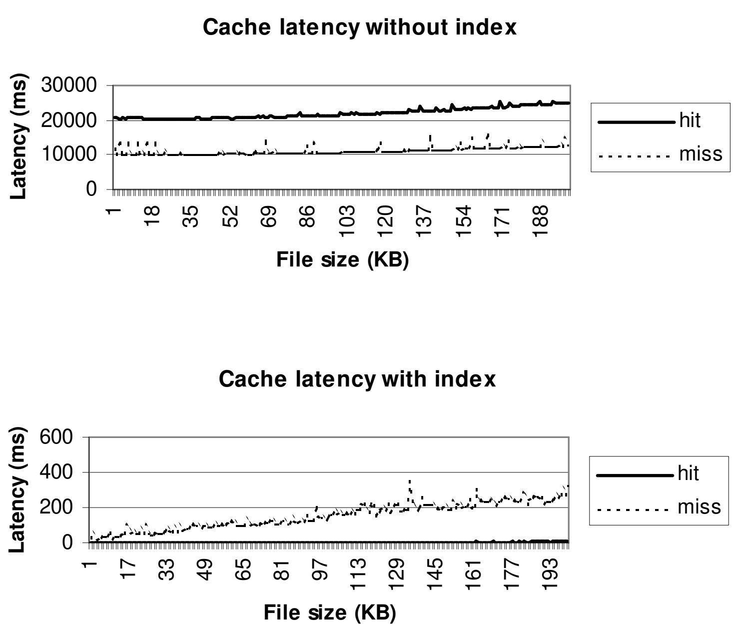

Figure 3 Hit/miss latency for various file sizes when cache is 100MB

miss occurs, the proxy forwards the request to the server. When the response gets back, it sends it to client before caching it into the database. Therefore user will not feel the latency of caching. There is just no way to hide the retrieval latency on a hit. In this sense a cache hit suffers more from lack of indexes than a cache miss does (the network latency is very low in our experiment, but since our host machine is not isolated

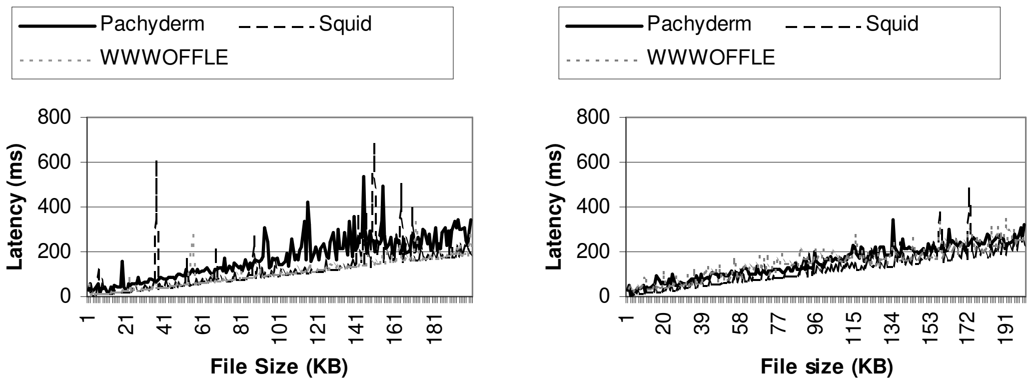

Figure 4. Miss penalty comparison across proxy caches. Left: cache almost empty; Right: cache loaded with 100MB

from other part of the network, some burstiness can be observed). This experiment was carried out when the cache was almost empty. With indexing, the hit latency is too low to be observed. We will show this in more details shortly.

Figure 3 shows the same test carried out when the cache was loaded with 100 Mb of web contents. There is no significant difference in latencies between this one and the previous one. Considering a proxy cache usually has huge storage, it is desirable to test the cache size at the level of gigabytes. Unfortunately, due to hardware limitation, we were not able to perform a test like this.

Figure 4 compares the miss penalty across three proxy caches. They are generally comparable except for some system bursty behavior. When the cache is almost empty (left), the curve of Pachyderm appears to be more outstanding while Squid and WWWOFFLE are very close. We ascribe this to the

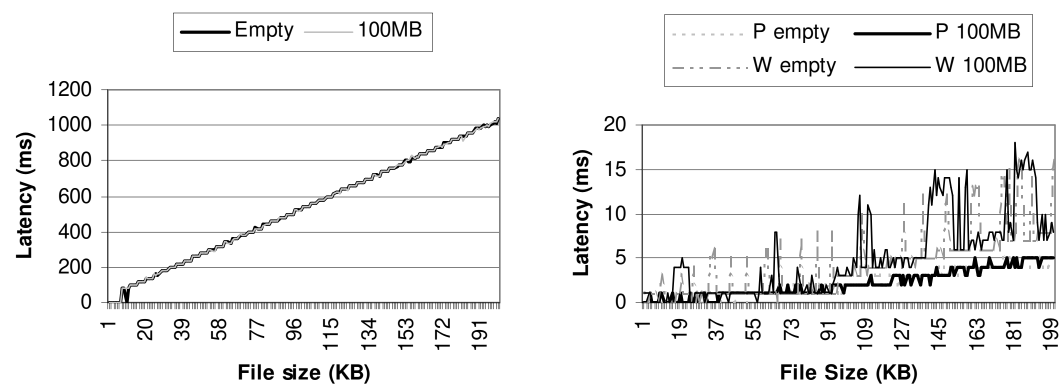

Figure 5. Hit latency comparison across proxy caches. Left: Squid; Right: WWWOFFLE and Pachyderm

overhead of Java virtue machine and JDBC layer (the other two servers are both written in C). When the cache is at 100 Mb, however, we can no longer easily distinguish Pachyderm's curve from the other two. This may imply that latencies of Squid and WWWOFFLE scale up faster than that of Pachyderm.

Figure 5 is the hit latency for three systems. Again we see no impact from the cache size. It is somewhat surprising to see the latency of Squid is much higher. Also its latency increases linearly along with the file size at a rate of 1 sec/Mb. When it comes to Pachyderm, the rate is around 25 ms/Mb, about 40 times faster than Squid. Its behavior is also more stable compared to WWWOFFLE. The variance is much smaller. It is possible that overhead has been introduced when Squid and WWWOFFLE implement their own features. Thus, we cannot simply say Pachyderm is superior although Figure 4 and 5 are both in favor of Pachyderm.

For a query on back issues, here is a typical query statement executed:

```
SELECT DISTINCT c.URL, c.version FROM Cache AS c, History
AS h WHERE h.uid = id AND h.version >= begin AND h.version
<= end AND h.URL = c.URL and h.version = c.version and
c.content LIKE 'keyword' ORDER BY URL, version;
```

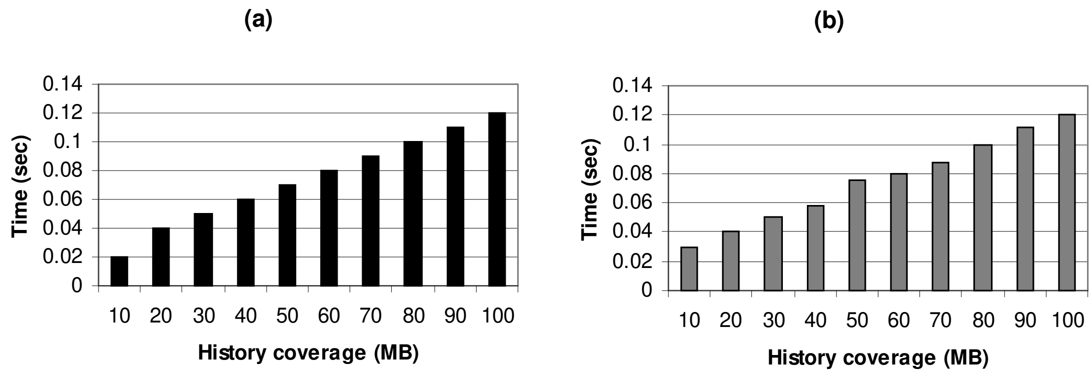

Figure 6. Back issue query time cost. (a) Full-ranged search on access time; (b) 5%-ranged search on access time

Complicated as it appears to be, the most expensive operation it has to perform is the table join. To speed up the procedure, we built indexes on user id, URL and version respectively in the "History" table. The *history coverage* is defined to be the volume of web components that a user history is referring to. We varied that between 10 Mb and 100 Mb. And we did the timing within MySQL so this only relates to the performance of MySQL (and this is our focus of scalability). Figure 6 presents the result. Although we change the time constraint on our queries, the numbers do not differ much. We also tried using the number of keywords as a parameter, ranging from 1 to 5. Still, the result is very much similar. We had expected that different constraints would make the query optimizer come up with different evaluation schemes, which does not seem to be the case in our experiment. It makes sense that the time cost is linearly dependent on the history coverage. The rate at which the time increases, or the slope of the fitted line, is about 1 second/Gb. This is an indication that MySQL can handle large data sets well. But that still relies on to what extent we can do extrapolation as the history coverage increases further. We have conducted the same experiments without any indexes on table "Cache" and "History". It takes 10s or 20s seconds for a single query!

One more issue is about the cleaner. Figure 7 gives some insight to the cost of this process. We

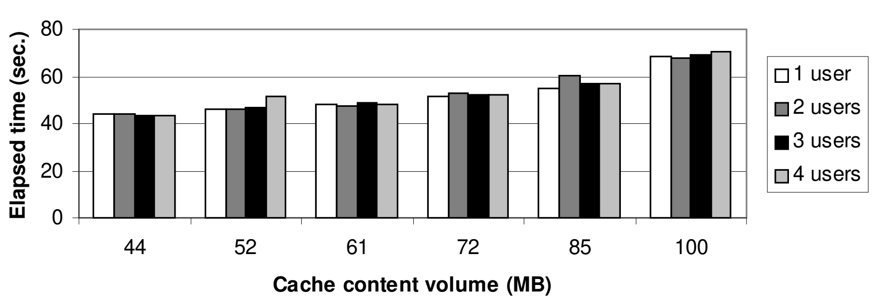

Figure 7. The effect of cache content volume and number of users on cleaning cost

examined two effects: the cache content volume and the number of users assigning web object priorities. It appears that the user effect is not significant. Given a cache volume, even the difference between one user and four users is subtle. The cleaning elapsed time is more sensitive to the cache content volume. With 44 Mb of cached objects, the mean time to clean 15% of the content is 43.8 seconds; with 100 Mb, the average time is 68.8. The increase in volume is 127% whereas the increase in time is 46%. We may infer that the cleaning is a much more expensive process (compared to other operations), but we can probably afford it even if the cached volume exceeds 100 Mb. This requires further testing though.

6 Related Work

All in all, Pachyderm is a novelty based on a collection of good ideas found in existing systems including proxy servers and file systems. We have already given extensive information in Section 1, 2 and 5.1. That is why this section is so succinct.

7 Future Work

Future work includes the implementation of some ideas that we proposed in this paper but were not implemented in our current version of Pachyderm (for instance, some hierarchy in storage system to make aggregation possible, more sophisticated user preference system, cleaning algorithm and prefetch schedule). Also, we need to test the proxy cache in a larger scale, both in terms of cache content volume and the number of users in the system.

We would also like to introduce deletion confirmation to the system. This feature is considered to be important for a user-oriented versioning system like Pachyderm. In a conventional proxy cache, once a document is chosen for deletion, the system can delete it right away without notifying the user because caching is transparent to the user. This is not true for Pachyderm. With the per-user browsing history, users will rely on the proxy cache to keep their web documents around. It would be surprising and frustrating to lose these documents without any notice. We attempt to use deletion confirmation to solve this problem. Whenever a user has items on the deletion list, Pachyderm can serve a page of deletion confirmation so that the user will have an opportunity to save it onto the disk if he really believes that page is essential.

To some extent, this also provides some kind of user training experience. Inexperienced or irresponsible users tend to assign high priorities to documents they do not really care much about. As a consequence, they will see this confirmation page much more often than "normal" users when they surf the web. Gradually, they should be able to figure out what is the right way to cast their votes.

8 Conclusions

Here is what we can conclude from our development and test experience so far. First, Pachyderm is a practically useful system. It provides customizable service on back issues. It promotes effective sharing by incorporating user assistance into cache management. Second, it offers reasonable performance. JVM and JDBC overhead is tolerable. Its hit latency is the lowest compared to that of Squid and WWWOFFLE with cached objects up to 100 Mb. The scalability also looks acceptable as far as we have tested. Finally, using a SQL database as the storage backend seems to be a viable choice. It simplifies the development procedure, improves the system manageability and makes the system more extensible and easier to maintain. Indexing turns out to be an indispensable part for performance. The difference it can make often amounts to several orders of magnitude.

References

[1] J. Gray, *The Transaction Concept: Virtues and Limitations*. Proceedings of VLDB. Cannes, France, September 1981, 144-154

[2] James J. Kistler, M. Satyanarayanan, *Disconnected Operation in the Coda File System. 1*3th Symposium on Operating Systems Principles, Asilomar, California, pp. 213-225. October 1991.

[3] Douglas S. Santry et al. *Deciding When to Forget in the Elephant File System*. Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, December 12-15, 1999, Charleston, South Carolina. Pages 110-123.

[4] http://harvest.transarc.com/

[5] http://www-4.ibm.com/software/webservers/edgeserver/

[6] http://www.apache.org/

[7] http://www.cisco.com/warp/public/cc/pd/cxsr/500/index.shtml

[8] http://www.f5.com/solutions/whitepapers/edgefx.html

[9] http://www.gedanken.demon.co.uk/wwwoffle/

[10] http://www.microsoft.com/proxy/

[11] http://www.netplicator.com/

[12] http://www.robtex.com/viking/main.htm

[13] http://www.spidercache.com/

[14] http://www.squid-cache.org/

[15] http://www.vicomsoft.com/softrouter/sfr.main.html

[16] http://www.web-caching.com/