Managing Cache Consistency to Scale Dynamic Web Systems

by

Chris Wasik

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

AUTHORS DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Data caching is a technique that can be used by web servers to speed up the response time of client requests. Dynamic websites are becoming more popular, but they pose a problem - it is difficult to cache dynamic content, as each user may receive a different version of a webpage. Caching fragments of content in a distributed way solves this problem, but poses a maintainability challenge: cached fragments may depend on other cached fragments, or on underlying information in a database. When the underlying information is updated, care must be taken to ensure cached information is also invalidated. If new code is added that updates the database, the cache can very easily become inconsistent with the underlying data. The deploy-time dependency analysis method solves this maintainability problem by analyzing web application source code at deploy-time, and statically writing cache dependency information into the deployed application. This allows for the significant performance gains distributed object caching can allow, without any of the maintainability problems that such caching creates.

Acknowledgements

I would like to thank my supervisor, Prof. Ajit Singh. Without him, this thesis would not be possible. I would also like to thank the members of my review committee, Prof. Naik and Dr. Bill Bishop, for the assistance they provided. Lastly, thank you to my family for their proofreading help.

Contents

List of Tables

List of Figures

List of Code Listings

Chapter 1

Introduction

As the internet matures, it continues to become more fundamental to the way our society functions. People are increasingly demanding more personalized content from websites. Personalized and dynamic content creates a problem for web application developers - it is difficult to cache data, as by definition, it is dynamic and personalized. This results either in slower response time, or the requirement of more expensive hardware.

1.1 Background

Dynamic web applications typically rely on a "3–layer" approach. Client requests are first received by a web server, and then passed to an application that generates the dynamic content. The web application often interacts with a database for the purpose of maintaining session state, or interacting with application-specific information.

As a website becomes more popular, the load that clients place on the system

becomes too great for a single machine to handle. To scale incrementally (which is a desirable objective), multiple web nodes or database nodes are added. Replication and clustering techniques are used to allow multiple machines to act as a single database. Since no state is stored on the web servers, clients do not have a preference as to which web server handles their request. Thus, web nodes can scale simply by adding more machines.

Caching is beneficial as it allows less hardware to handle the same amount of load. Less demanding hardware requirements often results in a combination of financial savings, and reduced request response time [KD02].

With each client receiving a personalized, unique page on a dynamic site, it is no longer possible to cache pages in the same way as the static content. This makes it more difficult to scale web applications using traditional caching approaches. The easiest solution is to simply throw more hardware at the problem. However, for reasons of both efficiency and economics, this is often not the best solution. Performance benefits can still be gained from caching, but a different approach is needed.

One existing caching solution is to cache full dynamically-generated HTML pages. However, such a solution would not allow for different users to receive different content (based, for example, on information such as their permission level, or the contents of their shopping cart). Other solutions attempt to push caching back to the database layer, in an effort to reduce the amount of time it takes to process a database query.

Benefits can be gained by caching data objects, such as post-processed database result sets. An example of this would be HTML fragments that represent posts on a bulletin board. Some websites have recognized and implemented this need, but

they rely on a single cache manager or log monitoring process, which is a single point of failure, and does not scale past one machine [CLL+01, CDI98, CDI+04].

1.2 The Problem

Currently, the most common way to make use of a distributed cache is to do it manually, in an unstructured way. Items are cached manually, and application designers need to be aware of all occurrences where the underlying data is updated, so that the cache can be kept in a consistent state. This unstructured approach creates a huge maintainability problem - whenever changes to the code are made, designers need to be completely aware of everything that is cached, so they can invalidate elements appropriately, and be assured the cache is never placed in a state inconsistent with the underlying data.

1.3 Contributions

In this thesis, a new approach called **deploy-time analysis** is introduced. According to this method, a distributed object cache is used to cache objects across web nodes. No single process or cache manager is used to ensure the cache is kept consistent with the underlying data. Rather, at deploy time, the code is analyzed for dependencies, and this information is statically written into the application. Each time the underlying data changes, the application automatically invalidates the corresponding cached elements.

The deploy-time method results in considerable maintainability benefits. When changes need to be made, the deploy-time method allows for far fewer lines of code

to be modified when compared to the unstructured approach. This is especially visible in larger applications that maintain more complex webs of dependencies. When modifications to a system were performed, it was found that the deploy-time method decreased the number of line modifications by at least 10% in small systems, and by as much as 95% in a larger system.

Run-time performance tests of two different websites show that distributed object caching offers significant benefits over solutions that do not implement caching, and solutions that implement only database caching as shown in Chapter 5. These benefits are recognized in the form of both reduced response time, and the ability of the hardware to handle greater numbers of clients. Most of all, the deploy-time analysis method solves the significant maintainability penalties that are introduced by the unstructured uses of distributed object caches. Using the deploy-time method, application designers do not need to worry about placing the cache in an inconsistent state - all invalidations are automatically generated at the appropriate times. Furthermore, there are nearly no performance differences between the unstructured (i.e., manually handcrafted and tuned) approach of using a distributed cache and the deploy-time method. This demonstrates that the deploy-time method introduces little or no performance overhead.

1.4 Overview

Chapter 2 gives some background to this problem, and discusses how web applications typically scale. Chapter 3 discusses existing uses of distributed object caches, and how they attempt to deal with the problem of cache consistency. Chapter 4 presents the deploy-time method, and demonstrates the improved maintainability it provides. Chapter 5 provides a run-time performance comparison with existing

methods. Some final conclusions are presented in Chapter 6, along with a discussion of current limitations, and areas for future work.

Chapter 2

Background

2.1 Introduction to Web Systems

Before trying to explain why caching is necessary, it's important to have some preliminary understanding of how web systems typically scale. This provides a basis of knowledge from which distributed caching can be investigated.

When someone "goes to a web page", he or she is really using his or her computer to issue a request to another computer. He or she is a client requesting data (the page) from a server. To interpret and process the request, the server machine must be running a software application called a web server. In the simplest case, the web server accepts the request, finds the page that was requested by the client, and sends it back to the client. This type of interaction is used for static data - web pages, media, and other files that do not often change. An example of this type of request is shown in Figure 2.1.

For more complex websites, the data returned to the client is not always static. For example, on a shopping website, the information returned to the client is a

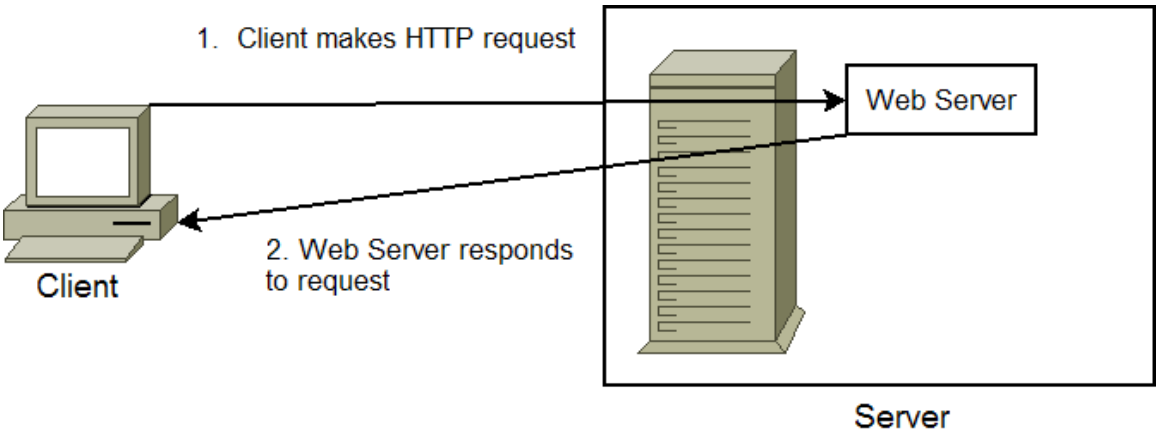


Figure 2.1: A client request to a web server

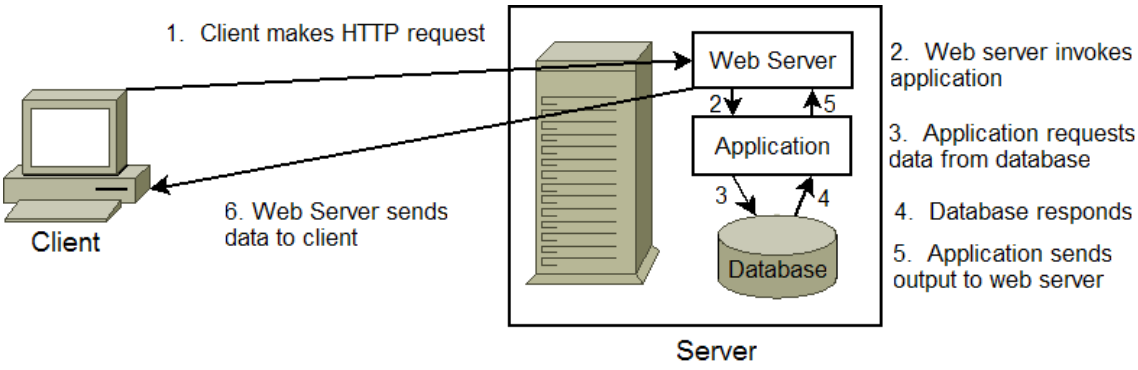


Figure 2.2: A client request to a dynamic website

function of things they have viewed before, and things they have added to their shopping cart. To determine what should be displayed to the client, the web server takes the user request and passes it off to another application. This application is responsible for generating the content to return to the client. In many cases, the application makes use of a database to keep track of system state (for example, the items in a user's shopping cart). An example of this type of request is shown in Figure 2.2.

This system involves three types of servers - a web server, an application server, and a database server. If the system is not in high demand, a single machine will likely have no problems dealing with all the requests. As the website becomes more popular, the processing power of the machine may become a bottleneck, and it may be unable to cope with the load clients are placing on it.

2.2 Scaling Web Systems

The first step towards better scaling of a web system is to "remove" the application server. When a web process passes incoming requests to an application, there is a considerable amount of overhead involved in launching the application. This overhead can be eliminated by embedding the application inside the web server. This is enabled through technologies such as mod_perl, PHP, and ASPs. Although this reduces the overhead for each request, system scaling is still a significant problem as load increases [CDI+04, LR00a].

The next step to scale a web system is to add another machine. This can be done simply by moving the database server to a separate machine. In this way, when a request arrives, the load is spread out over two machines. The application server does its work, but makes requests to the database server for any database accesses. An example of this system can be seen in Figure 2.3.

Although moving the database is fairly easy, it is not a perfect solution. Now that the system is using two machines, it has two points of failure. The system will cease to function if either of the two machines fails. Additionally, assuming the website continues to grow, eventually the capacity of one of the machines will be reached, and the site will need to grow again.

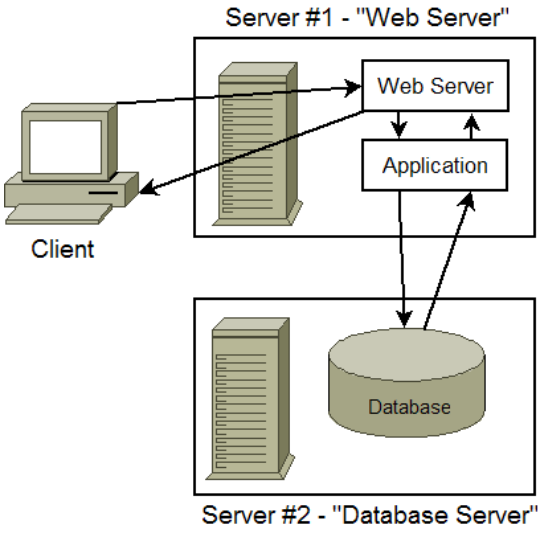


Figure 2.3: A web system with a separate database server

Depending on whether the web server machine or the database machine reaches capacity first, different solutions can be employed. The two components can be scaled independently.

One way to grow is to purchase a bigger, more expensive machine. This has a number of limitations. A machine still represents a single point of failure, although more money can buy more reliability to some extent. The largest problem is that when the capacity of the new larger machine is reached, the only option is to purchase another even bigger and even more expensive machine. Ideally, the system would be able to scale incrementally by simply adding additional computing power as needed. Incremental scalability is a key concept to web system scalability, and has fueled the growth strategies of many companies providing web services, such as Google [BDH03, Bre01, FGC+97, OMG+05]

To scale the web servers incrementally, a number of tricks can be used to make a group of machines appear as one. Software load balancers, hardware load balancers,

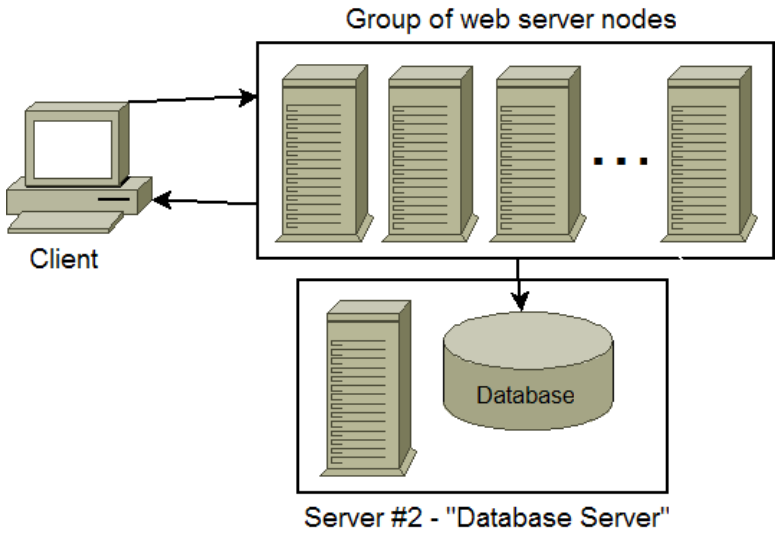


Figure 2.4: A web system with multiple web nodes

or techniques such as DNS load balancing can be used to spread the load out over multiple machines and allow the system to scale one machine at a time. The exact technique used to scale is not important, as for the purpose of this thesis, they all have the same result - all techniques allow multiple web servers to be used to service the requests of clients. An example of a system with multiple web nodes (physical web server machines) can be seen in Figure 2.4.

It is of some importance that the web server machines do not store any state. Regardless of the method of load balancing used, it is possible that the same client will have its requests serviced by two different web server machines on two sequential requests. It is therefore imperative that the machine that services the first request not store any data that is not accessible to the second machine.

It has already been discussed that state is fundamental to many dynamic sites such as a shopping cart. The requirement of not storing any state on the web servers simply means that the state must be stored elsewhere - either on the client

machine itself (likely in the form of cookies), or more commonly, in some sort of shared storage, like the database server.

2.2.1 Database Replication and Clustering

With multiple web server nodes handling requests from more clients, it becomes increasingly likely that the database server will reach its capacity. Fortunately, database replication and clustering are well understood and deployed technologies.

A replicated database consists of one (or more) master servers, with multiple slave servers. All writes must occur at a master database. The master database sends updates to the slave databases with varying degrees of consistency guarantees as required by the application. Replication allows the use of many slave databases that the web nodes can issue read queries against. An example of a system employing this technique is shown in Figure 2.5.

This buys more scalability and redundancy. More slave database nodes can be added to respond to read requests from clients. However, with additional clients using the system, it is likely that more write requests will also be issued against the master database. The master must mirror these write requests to all the slaves to keep them updated. Eventually, the slaves reach a point when their processing time is occupied by these write requests, instead of responding to read requests.

Database clustering is a similar technique that accomplishes roughly the same goal. Whereas replication is typically an asynchronous process, relying on the application (or other supporting utilities) to provide failover, database clustering provides synchronous communication between database copies, and more automated administration [Joo06].

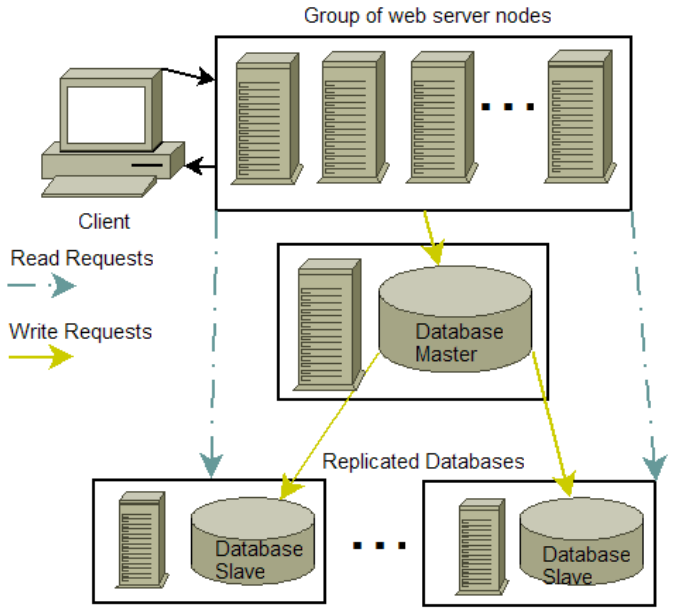


Figure 2.5: A web system with multiple database nodes

2.2.2 Database Partitioning

Even if clustering or replication techniques are employed, as a system grows, it would be beneficial to reduce the number of database accesses. Fewer accesses would result in less hardware to support, a system with more room to grow, or a combination of the two.

One solution to reduce the number of queries against any one server is to partition the data. This does not reduce overall hardware requirements, but rather spreads the load around. This technique is often employed automatically in clustering technologies [Joo06]. For example, there may be little need for user data, accounting data, and authentication data to be stored in the same database, as they may never be accessed at the same time. Splitting them up allows the load on the database to be split into multiple groups, reducing the percentage of write

requests that each slave needs to perform.

2.2.3 Data Caching

Database partitioning does nothing to reduce total hardware requirements. Additionally, each type of partitioning has its limits. Data can only be partitioned so far using vertical partitioning, while horizontal partitioning limits the types of queries that can be performed in a timely manner. Eventually, one must find an alternative solution to reduce database load. This alternative solution often comes in the form of data caching. If database results can be stored somewhere, then the web nodes will not need to query the database as frequently. This assumes that more clients are reading the data than writing it, which, although dependent on the specific application, is very common in web applications [CDI+04, AKR01, ACC+02].

Some Key Caching Concepts

At this time, it is useful to reiterate and explicitly state some concepts that were previously alluded to. Dynamic web environments involve state. All the data composing a system can be thought of as the "state" of that system. Certain types of state, such as a session identifier, can often be stored on client machines. Other types of data, such as a customer list, product list, employee list, or an order history, are often stored on the server for reasons of security and practicality. This type of information could be stored simply as long-living variables in some sort of shared memory, text files on a shared hard disk, or in a database. This data is most frequently stored in a relational database due to a number of advantages offered by modern relational database, including:

- Easy accessibility from multiple machines

- Incremental scalability through clustering or replication
- Referential integrity
- Transactional integrity
- Standard method of inserting/retrieving data (SQL)

When a web server starts caching data for performance reasons, there become two copies of the data: One copy in the database, and another copy in the cache. In a complete system, there may be many levels of caching, but the level of interest for this discussion is the caching implemented by the web application itself (as opposed to proxy caches, data cached by the database in memory before writing it to disk, processor caches, or caches implemented by internet service providers, although database caches are examined later on from a run-time performance perspective). The data in the database will be referred to as the "physical" copy of the data, while the data in the cache will be referred to as the "cached" copy.

Shared storage could be implemented through means other than a database. However, due to the relative ubiquity of databases in web environments, and the advantages stated above, the use of the term "database" will be applied to describe any type of shared storage.

Generating Static .html Files From Dynamic Data

One method of reducing database queries is to generate static .html files each time the database changes. This allows database queries to be eliminated, as only static pages are being requested. Additionally, fewer web nodes are needed, as static content can be served much faster than dynamic content since no application is

invoked. However, this method has a number of drawbacks. Dynamic features such as authentication cannot be enforced. The largest drawback is that many sites (such as shopping carts) generate content based on state (such as "items in the shopping cart" or "user permission level"), and static content cannot be used in this case.

Materialized Views

Materialized views are a caching technique that occurs at the database level. Materialized views generate "virtual tables" from other data in the database. The materialized view is the result of a SQL query that is cached. Materialized views can be updated at specific time intervals, or whenever the data they depend on changes. While they can be useful, materialized views only operate at the database level, and the only benefit they provide is to reduce the execution time of complicated queries. Materialized views do not help out with caching post-processed data, as in the previous method, and thus the scope of their applicability is limited.

Query Caches

Query caches are another database caching technique that can also be used to help an application scale. A query cache is typically a middleware layer that stores the results of one query so that the query does not need to be processed again in the future. Unfortunately, these are difficult to keep updated [ASC05]. Additionally, they also may have a storage capacity limit, as they are typically 32-bit processes running on machines with 4 GB of memory. This limit may be acceptable for smaller websites, but larger systems utilize and cache significantly more than 4 GB of data. While any query cache is usually better than nothing, a query cache can

actually hurt performance due to the overhead of maintaining it when a particular site experiences a high number of updates [Fit06].

2.2.4 Generic Object Caching

Previous methods have looked at caching either HTML pages or database queries. To avoid the discussed drawbacks of these methods, it is desirable to be able to cache any type of data. A desirable solution would be somewhat of a hybrid approach. Caching just HTML is too broad a technique, as it does not allow websites to still be dynamic. However, caching at the database level is too narrow. Often times, the application is responsible for processing query results after they are returned. It would be convenient to be able to perform this processing in the application, then cache the result.

Allowing application designers to cache any type of data adds flexibility. It allows the system to cache the raw result sets of database queries, to cache processed versions of query results, to cache full HTML pages, partial HTML pages, variables, or any object between these extremes. This flexibility can be used to reduce not only database load, but application processing time as well.

To be able to cache any type of data, other caching techniques must be examined. The most obvious way to cache generic data is inside web server processes. It is relatively easy to create a system where each instance of the application caches data, so subsequent requests do not need to issue database queries or perform complex processing. This would be implemented through the use of long-living variables inside the application.

Although easy to implement, this results in each web application process having its own cache. Web server nodes typically run multiple instances of web application

processes per machine, so each machine has multiple copies of a cache.

Multiple copies of a cache on the same machine results in wasted memory, and a higher percentage of cache misses, as each request can only interact with one cache. The obvious next step is to share caches amongst the processes, so that each machine only has one cache.

This approach still results in multiple duplicate caches. In a large system, there are likely many web nodes. Each of these web nodes would have its own cache, and the system again has duplicated caches with low hit rates. The next step is to move to a distributed cache, where every process on all of the machines can share the same cache.

Memcached is a distributed object cache [Fit04]. Running Memcached involves starting a process on one (or more) machines with a specific amount of memory allocated to that process. Applications are made aware of the machines where Memcached is running, and can interact with Memcached through an API provided in many languages (Perl, PHP, Python, Ruby, Java, C#, C, or via an open protocol) [Fit06, Fit04]. Objects are stored and retrieved from the cache with a "key" (an identifying string). The key is hashed to a particular server (as Memcached can run on multiple servers), where actions on that key are performed. Memcached is used on a number of popular websites, such as Slashdot.org, LiveJournal, Digg.com, and Wikipedia.org.

2.3 Producer/Consumer Model of Information Flow

Before continuing this discussion with an examination of existing implementations of Memcached, it is useful to think about the situations in which Memcached could

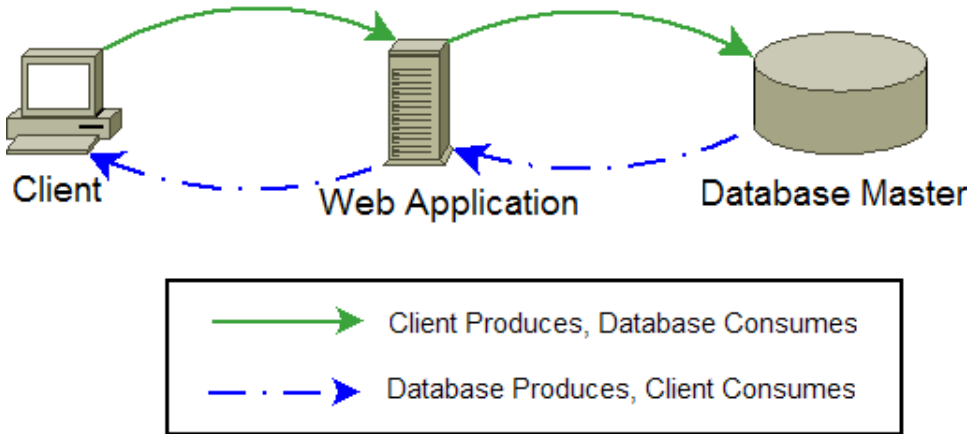


Figure 2.6: Producer/consumer model of information flow

be beneficial. Section 2.2.4 stated that not being able to cache any type of object (variables, full HTML pages, partial HTML snippets, database result sets, etc.) was one of the major drawbacks of other caching implementations. However, upon closer examination, it can be seen that it is not beneficial to be able to cache any type of object - only objects that interact with a database at some level.

In a web application environment, data flows between two entities: clients, and the database. In any given request, one of these entities is the producer of data, and the other is the consumer. For example, when users make posts on a bulletin board, they "produce" the content, and it is "consumed", or stored, by the database. When a customer looks at a shopping cart, the database produces the contents of the shopping cart, and it is presented to the user. Any state maintained by the system must be maintained by the database, as discussed in Section 2.2. An example of this interaction is shown in Figure 2.6.

It should be noted that during any single request, both parties can act as both the producer and consumer. However, these cases are typically two-step processes. For example, when a user adds an item to their shopping cart, this information is

be written to the database (part 1), and then read from the database (as part of the same HTTP request) to display the user's current shopping cart.

Sitting between the producer and consumer is the web application. The web application is responsible for adjusting, processing, or formatting the data. By looking at a web application in this way, it can be seen that any information that the system would want to cache must, in some way, be involved in a database interaction. Data is likely processed before or after it is written to/read from the database, but any data that the application needs to access must either originate from, or terminate at, the database. This is important, as it provides an intuitive view of what Memcached does, and how it is able improve performance.

2.4 Caching for Temporary Storage

One additional way a distributed object cache can be used is to help eliminate database writes. This is quite appealing from a performance and scalability point of view, as writes require database locks, which may result in contention, causing decreased performance. This section discusses the use of Memcached in this manner. The purpose for this discussion will be seen later in Section 4.1.3, but it is useful at this point to explain the issues involved with using Memcached for temporary storage.

There are a number of situations where a database is simply used as a "scratch-pad" for temporary data. The most common use of this is sessions. Sessions are used to track individual client state. Clients are provided with some unique identifying string that is long enough to ensure it is unlikely to be guessed. The client then must provide this string to the server on each request. This often occurs through the use of cookies.

Rather than keeping track of session information (such as "last visited page", "number of requests", or "permission level") through a database as is usually done, a distributed object cache can be used. Individual sessions are not accessed by multiple users, and thus concurrency to the same session data is not a concern. The ACID semantics provided by databases require overhead to implement, and are not necessary for session tracking.

Writing session information to Memcached would save a significant number of database accesses. One problem with this concerns ejecting session data from the cache as part of a cache replacement policy. If the cache were to run out of room, session data could be ejected from the cache while it is still in use. Fortunately, Memcached can be configured such that data is never ejected from the cache unless explicitly deleted. This is not a problem assuming the cache is large enough to hold all the session data (which is not an unreasonable assumption).

A greater drawback to this method is that machines storing session information can crash. Previous discussions involved the use of Memcached as a cache - a "fast duplicate" copy of data. By using Memcached as a session store, it would be the only place this information would be stored. If the Memcached server were to crash, session information would be lost. There are a number of tricks that could be employed, such as duplicating the cache, but Memcached was never designed to be a completely reliable data store - just a fast cache. It does not provide reliability guarantees or any fail-over mechanisms.

That being said, Memcached can be, and is, used as a session store on a number of high-profile sites such as Digg.com, which receives over 400 million hits each day [Tim, Das]. System designers simply need to be aware that if a Memcached server crashes, all sessions stored on that server will be lost. If a crash were to occur, these users would be required to log in again. In the case of Digg, this is not a huge

problem, as all pages can still be read without logging in - it is only for moderating duties that an account is needed. Depending on the application in question, it may be feasible to use Memcached as a store for temporary data, further eliminating both database reads and writes.

2.5 Summary

This chapter provided an overview of how a single website can scale across multiple server machines to deal with increasing client load. Although one could always purchase more machines to help a website continue to scale, it is often significantly cheaper to implement caching techniques to reduce the load on existing hardware (and thus allow that hardware to serve a greater number of client requests). In larger environments where multiple web server machines are needed, it is advantageous to share a single cache amongst all the machines. Memcached is an example of this technique, which is known as Distributed Object Caching. The following chapter examines how Memcached is used.

Chapter 3

Existing Scalability Solutions & Related Work

Chapter 2 discussed how scalability was important in web systems, and how Memcached, a distributed object cache, was able to improve scalability through caching. This chapter serves two purposes. First, it examines existing implementations of Memcached in Section 3.1. During this examination, the problem of cache consistency is discovered. The second purpose of this chapter is to introduce some related areas that could help solve this problem of cache consistency.

3.1 Existing Implementations of Memcached

This section provides an examination of the code from a number of popular websites that make use of Memcached. The purpose of this examination was to identify how existing large websites deal with Memcached, in the hopes that a useful architecture for distributed object caches could be identified.

3.1.1 Slashdot.org

Slashdot.org is a popular technology news website. It is programmed in Perl, and creates a database wrapper object to interact with its database. This database wrapper abstracts the differences between different databases (theoretically; at the current time, only MySQL is supported). It also allows for activities such as application-level database query logging.

Most (but not all) of the Memcached references in the slashdot code are contained in the MySQL wrapper, located in Slash/DB/MySQL/MySQL.pm. There is one global Memcached object defined in this file. This file also contains a function `getMCD()` that is used to return a reference to the global Memcached object. In this way, Memcached can be enabled or disabled all in one location, simply by modifying the `getMCD()` function. References to the Memcached object are interleaved throughout the code.

Usage of the Memcached object happens at various places throughout the code. For example, the function `setStory_delete_memcached_by_stoid()` is responsible for deleting a story from the cache, given a story ID. This piece of code can be seen in Code Listing 3.1.

Of particular interest here is the fact that there are multiple keys that need to be cleared for each story. The three keys to delete from the cache are:

- `$self->{_mcd_keyprefix}:st:$stoid`
- `$self->{_mcd_keyprefix}:stc:$stoid`
- `$self->{_mcd_keyprefix}:str:$stoid`

These three keys represent the story data, the chosen topics, and rendered topics for a particular story. Whenever any aspect of the story changes, these three keys

Code Listing 3.1: Slashdot.org: Removing a story from the cache

```
my @mcdkeys = (
   "$self->{_mcd_keyprefix}:st:",
   "$self->{_mcd_keyprefix}:stc:",
   "$self->{_mcd_keyprefix}:str:",
);

for my $stoid (@$stoid_list) {
   for my $mcdkey (@mcdkeys) {
      # The "3" means "don't accept new writes
      # to this key for 3 seconds."
      $mcd->delete("$mcdkey$stoid", 3);
      if ($mcddebug > 1) {
          print STDERR scalar(gmtime) .
          "$$ setS_deletemcd deleted '$mcdkey$stoid'\n";
      }
   }
}
```

need to be cleared from the cache. The `setStory_delete_memcached_by_stoid` function is called nine times in the Slash/DB/MySQL/MySQL.pm file, indicating that there are multiple locations in the code where aspects of a story can change. If any new features were added that changed story information, the person adding the feature would need to be aware this function existed, and call it at the appropriate time to guarantee the cached data was not inconsistent with the underlying story data.

This method of explicitly clearing individual cache keys is common practice throughout Slashdot's code. Whenever underlying data is changed in the database, programmers must explicitly invalidate the appropriate cache keys.

3.1.2 Wikipedia.org

Wikipedia.org is a website that strives to create a community-driven, collaborative encyclopedia. Similar to Slashdot, Wikipedia uses a wrapper class (includes/-Database.php) around a MySQL database. Unlike slashdot, this database object does not contain any references to Memcached.

Wikipedia provides an easy mechanism to enable or disable Memcached through the use of the LocalSettings.php file. In this file, the variable `$wgUseMemCached = true;` may be enabled or disabled. Unfortunately, this variable must then be manually checked at each instance where Memcached is used.

In contrast to Slashdot, where the keys used in Memcached are selected and not maintained anywhere, Wikipedia has chosen to keep track of the keys that are used in a file called Memcached.txt. A portion of this file is shown in Appendix A. Of particular interest is the fact that the file keeps track not only of where each key is set, but also the functions in which that key is cleared.

The only structure imposed on the use of Memcached is through the text file in Wikipedia's code base. The text file is used to help designers be aware of all the cached elements, so they will be less likely to place the cache in an inconsistent state.

3.1.3 LiveJournal.com

LiveJournal.com is a website where people can write about their lives, or other items of interest. It is especially interesting, as the creators of LiveJournal.com were also the creators of Memcached. They realized that a distributed cache could improve the scalability of their site, and so they created one [Fit04].

LiveJournal's database use is slightly different than Wikipedia and Slashdot. LiveJournal partitions their database into multiple independent "clusters", where each cluster uses replication. The livejournal/cgi-bin/ljdb.pl file is called when access to a database is necessary. It has functions such as `get_cluster_master()` and `get_cluster_reader()` that are responsible for returning the appropriate references.

LiveJournal uses a wrapper object (livejournal/cgi-bin/LJ/MemCache.pm) around Memcached. This wrapper allows for the easy configuration and enabling/disabling of Memcached.

With the exception of the wrapper object, LiveJournal's use of Memcached is similar in style to Wikipedia. It lacks the robustness of a mature, well-developed architecture. Memcached does not appear to fit into the system architecture in any way, only that numerous references to Memcached appear throughout the code. LiveJournal also suffers from the problem of having to delete multiple cached objects at the same time. Numerous examples of this can be found throughout the code, with one example shown in Code Listing 3.2. This example was taken from the

Code Listing 3.2: LiveJournal.com: Deleting elements from the cache

```
# memcache clearing
LJ::MemCache::delete([ $csid, "sasi:$csid" ]);
LJ::MemCache::delete([ $_, "saui:$_" ]) foreach @$uids;
```

`merge_schools()` function in livejournal/cgi-bin/schoollib.pl:

LiveJournal again matches the organization of Wikipedia by also using a text file to keep track of the used keys. The text file used by LiveJournal is called livejournal/doc/raw/memcache-keys.txt. In addition to the name of the key, this text file provides a short description of what each key contains. Again, the only structure imposed on Memcached in the LiveJournal code base is the use of a text file to maintain a list of what cache keys are used.

3.2 Motivation

As was observed by looking at existing implementations of Memcached, it is typically implemented as somewhat of a "hack". Its use is completely unstructured. References to Memcached appear scattered throughout the code, with no formal method used to keep track of when cache elements are set or deleted. The examined implementations would all be easily "broken" (the state of the cache would be inconsistent with respect to the underlying physical data) if the system were to be extended without full understanding of the dependencies between the physical and cached data.

For example, on a bulletin board website, assume one particular cache element stores a list of posts to display on the bulletin board. Now, suppose functionality

was added to the system to allow editing of a post's content. The programmer adding this functionality would need to be aware of the cache, so when the new content is saved, the cache element storing the list of posts can be invalidated, as the underlying content had changed. If the programmer was not aware of this, the cached element would not be invalidated, and the bulletin board would still display the old data even after the content of the post had changed. To further complicate matters, other cached pages could also rely on this data. For example, a "list of recent posts" page may also be cached, and thus would also need to be invalidated.

Although the use of a text file to maintain a list of keys may be beneficial, it also introduces an additional level of complexity around a project. Each developer must be aware of the list, and aware of all the relationships between the cached objects and their physical data representations. Any failure to clear the cache after performing an update to physical data can cause the cache to enter a state inconsistent with the physical data.

In general, there are no formal solutions that solve the consistency problems that are encountered when using a distributed cache. This makes it very difficult to use Memcached in large systems. A quick review of some existing technologies may suggest useful solutions to this problem.

3.3 Related Work

Distributed object caching for web applications is a relatively new area, and thus there is no work directly associated with it. Fortunately, it overlaps with a number of existing areas where extensive study has been conducted. Many of these areas provide invaluable research concepts that can be extended and enhanced to function

with a distributed object cache in a web environment. These related areas, and contributions they provide, are mentioned in this section.

3.3.1 Database Caches

When a web application is viewed as a set of independent components as discussed in Section 2.1, it can be seen that there are three main components: the database, the web server, and the application. Each of these components contributes towards the total time needed to generate the webpage. This section discusses methods that focus on speeding up the database part of the web system, independent of the application or web server.

Query Caches

A query cache is an addition to a database that is transparent to the application, or any other client. Its purpose is to cache the results of database queries so future execution of those queries are faster when run by the application. Query caches can appear either as a middleware layer between the application and the database [LKM+02], or embedded in the database itself [ASC05, MyS06b].

Similar to a distributed object cache, query caches must deal with invalidating cached data when updates occur. Since the underlying data is the same both in a query cache and in a distributed object cache, the methods used to identify what cached elements need to be invalidated are somewhat similar. One popular method is to clear the entire query cache for a complete table any time that table changes [MyS06b]. Clearly this approach often results in invalidating a significant amount of data that is not invalid, which is why finer-grained approaches are more appealing [ASC05].

Materialized Views

Materialized Views are similar to a query cache, in that they cache database result sets. However, a materialized view acts as a standard database view, and not a transparent database cache. A view is a "computed table" - it displays the result of a pre-determined query. A materialized view can be thought of as a cache for a view. Depending on the application, a materialized view can be updated at set intervals, or any time the underlying data changes.

One interesting study implemented materialized views on the application side of a web system, rather than on the "database" side. This improved performance, as it prevented the application from having to connect to the database server [LR00b]. Cache tables are a similar method of database caching, where table data can be stored in multiple caches closer to where the data is needed (for example, directly on the webserver) [ABK+03]. Materialized views and cache tables utilize the same invalidation techniques as query caches, and thus provide the same value to distributed object caching as query caches.

3.3.2 Web Caches

Web caches consist of one or multiple caches sitting in front of the webserver, closer to the client requests. If the web cache contains the full HTML page the client is looking for, then there is no reason for the request to arrive at the web server [YBS99, FCAB00, Wan99]. There are often multiple web caches in a web system. Many times, web caches are operated by ISPs (internet service providers) in an attempt to reduce the amount of network traffic they send outside of their network. If an ISP can cache popular files, they can significantly reduce the number of requests they must make to webservers.

Web caches are most applicable to static content. However, a number of attempts have been made to use web caches for dynamic data, and these are discussed in Section 3.3.3.

3.3.3 Dynamically Generating Static .html Files

The web caches and web caching algorithms discussed in Section 3.3.2 are only useful for webpages that rarely change, since every time a page changes, all the caches need to be updated, or else they are providing outdated information. For this reason, web caches are usually only used for static .html pages or media files, and not dynamic pages. Dynamic pages are usually marked as non-cacheable by the sites that generate them, as they often change very frequently (possibly generating unique data for every request).

A number of websites, such as Wikipedia, make use of their own caching servers [Wik06]. These cache servers are specifically designed to work with dynamic data, and only cache the web pages for as long as the underlying data remains constant. Generating static content and allowing it to be cached is an extension of the work done with standard web caches.

One method that was proved with the website for the 1998 Winter Olympic Games is called DUP (Data Update Propagation) [CDI98, CDI+04]. This method makes use of an ODG (Object Dependency Graph) that relates physical data elements to the cached objects that depend on them. A series of linked lists and hash tables inside of a cache manager (a long-running daemon process) are used to maintain this mapping. Cached objects are registered with the cache manager, and the application informs the cache manager of every update via API calls [CDI98, CDI+04].

Another method of dynamically generating static files from dynamic content is discussed in [CLL+01]. This method is more a passive method that can be used to enhance an existing application. This method consists of a "sniffer" and an "invalidator". The sniffer collects information about the database reads caused by each page, and uses this information to build a "request-to-query" map. The invalidator is another application that monitors the database update logs. When the invalidator recognizes database updates that invalidate existing cached pages, the invalidator is responsible for invalidating those pages.

Both of these methods provide a number of excellent ideas on maintaining cache consistency. However, they both fall short on several key aspects. The major problem with both methods is that static content is being generated and cached. There are a number of applications where generating fully static content is appropriate. One example where this method is appropriate is with the Olympics website. For this site, each person that views the website receives the same content (medal counts, event results, etc).

In more dynamic environments, each user could receive a slightly separate variation of the pages depending on parameters like their permission level, previous spending history, what other users with similar tastes have recommended, or other application-specific parameters. The key difference is that the page is still dynamic. There may be elements of the page that are the same as other requests, but the complete page itself is unique.

Another major issue with existing methods concerns their scalability. Both methods involve some sort of process, such as sniffer/invalidator processes, or a cache manager process. As a site scales to multiple machines, duplicated processes and multiple log files would require significant changes to be made to these architectures. In fact, the method proposed in [CLL+01] specifically states that their

configuration only has one DBMS. Despite these limitations, the ideas presented through these methods provide excellent foundations on which to build a cache consistency model for distributed object caches.

3.3.4 Hybrid Strategies

Rather than trying to generate static .html files from dynamic content, or only relying on database caching techniques, there are a number of existing methods that aim to improve performance through both strategies [YFIV00, SKRP01]. Although combining both caching methods appears to be an attractive approach, it appears that these methods often feature overly complex methods of specifying dependencies, and do not easily scale to environments with multiple servers. Furthermore, it is unclear whether the performance benefits that can be obtained from a hybrid strategy are significantly greater than one of the other approaches. The reason for this is likely due to the overhead that is associated with trying to find which combination of caching strategies provides an optimal solution.

3.4 Summary

This chapter examined how Memcached is used in a number of existing websites. The unstructured nature of these implementations poses a significnat maintainability challenge, as application developers need to be aware of what underlying data each cached element depends on to ensure the cache remains consistent with the underlying data. Existing systems that dynamically generate static .html files provide a number of interesting algorithms for dealing with this consistency problem. In the following chapter, these existing algorithms are examined, and a new method

to deal with cache consistency is presented.

Chapter 4

Cache Consistency

The major problem when using a distributed object cache involves invalidating the data in the cache (either deleting it, or marking it as expired). Once an object is cached, that object typically stays in the cache until it is removed. Objects that are cached can depend on one or more physical data elements (typically a row or column in a database). When one of these physical data elements changes, all the cached elements that depend on that data element must be invalidated. Additionally, if any other cached elements depend on the expired element, they must also be invalidated. Determining what elements in the cache to invalidate is a difficult problem.

A number of existing web caching models were discussed in Section 3.3. Each of these models have an associated consistency protocol [Kaw04]. The models with the most useful consistency protocols were presented in Section 3.3.3. These models were used to generate static .html files. They are the most useful models for use with a distributed object cache because they were also designed to work in a web environment with dynamic webpages. Although generating static .html files

is not sufficient for use in applications making use of Memcached, similar cache consistency concepts can be applied.

This chapter discusses the shortfalls of different aspects of existing models, and presents a new model, the "deploy-time" dependency analysis method, that solves these problems.

4.1 Problems with Existing Models

There are a number of different models that are used to maintain cache consistency in a web environment. The shortfalls of these models lead naturally into the development of the deploy-time method, which is introduced in Section 4.2.

4.1.1 Log Monitoring Systems

One approach to creating an invalidation protocol involves examining database query logs, such as in [CLL+01]. When the application issues a query to the database causing data to be updated, that query shows up in a log that is maintained by the database. This model was originally built under the assumption that caching would reduce the load against the database to a level so low that it could be handled by one database server. Under larger application workloads, this is not possible. Additionally, having only one database node results in a single point of failure, and if possible, such a situation should be avoided for reliability purposes. The ideal type of system setup is shown in Figure 2.5.

With the system shown in Figure 2.5, the master database is the database responsible for performing all "update" requests. As shown, this setup does not appear to cause a problem to the "update log monitoring" technique of invalidation.

However, under a failover scenario, one of the slave machines becomes the master, and all write requests are then be performed on that machine. The log monitoring technique does not implicitly have any notion of "failover recovery". Although it is possible to develop a log monitoring application that supports a failover technique, there is significant difficulty in doing so.

One additional problem in this situation is caused by the possibility of cache delays. Update queries performed by the database would likely be performed before the log entry for that update query is written. As such, there is the possibility that read requests performed very shortly after a write request may return stale data. Although such a delay would be very small, there may be some applications where it would be unacceptable.

4.1.2 A Cache Manager

The next approach that aims to provide a functional invalidation protocol utilizes a cache manager [CDI98, CDI+04]. Rather than using a process that simply monitors query logs, this method requires a process to act as a "Cache Manager". The cache manager stores a dependency graph, linking the cached data elements to the physical data they depend on.

A small example program that uses a cache manager is shown in Code Listing 4.1. With this example, a cache element of "thisKey" is set to a value of "thisValue". The cache manager is informed that the cache element depends on two database values (table.col and table2.col4). Some time later, the cache manager is informed of an invalidation to table.col (either through code, or through a database trigger), and a database query is issued that modifies table.col. The cache manager is responsible for removing "thisKey" from the cache.

This solves the problem in the previous method where a cached element is still present in the cache after the physical data has been updated. Here, the data in the database is not updated until the cached element is removed. Additionally, this is an intuitively attractive system. One may even imagine the case where the cache and cache manager are combined, resulting in a "dependency-aware cache".

Unfortunately, this method has its own problems. If the cache manager is a single process running on a single machine as in [CDI98, CDI+04], reliability becomes the key problem. If the machine running the cache manager process fails, then all the dependency data will be lost. As such, nothing in the cache will ever get invalidated, causing the website to stop displaying updates. The cache manager is now the single point of failure.

If, on the other hand, a new system is developed where a highly available cache manager is designed (which is a challenge with its own set of problems), performance may become an issue. With a highly available cache manager, one could imagine a cache manager process running on multiple machines. Each time a new dependency is set, the cache manager must write that dependency to all nodes. If it does not, and a single node fails, dependency information would be lost, and the cache could be placed in an inconsistent state (since updates would not cause the necessary invalidations). Although such a system could work, the problems outlined and the unnecessary complexity of the system make it a poor choice. As will be seen in Section 4.2.6, the deploy-time method offers a more attractive alternative while still maintaining the benefits of this method.

Code Listing 4.1: Sample API of a cache manager

```
$cacheManager->setDependencies("thisKey", "table.col,
        table2.col4");
$cache->set("thisKey", "thisValue");

#...other application code here..

#This query causes an invalidation of thisKey,
#since thisKey depends on table.col
$cacheManager->update("table.col")
$db->execute("update table set col=5");
```

4.1.3 Distributing the Dependency Graph

To combat the problem of the system's dependency graph being a single point of failure, it is necessary to distribute it. Rather than building a highly available cache manager, one could simply run multiple copies of the cache manager application on different web nodes. If each web node had its own cache manager, then each web node would be able to issue cache invalidations at the appropriate times.

Unfortunately, the fact that web servers must be stateless comes into play. An example of this can be seen in Figure 4.1. In this figure, the client makes a request to server X and sets a cache dependency "thisKey". "thisKey" is stored in the cache, and the data it depends on is noted in the cache manager on server X. Later on, that same client makes a request to server Y that should cause an invalidation. However, since the dependency was set on server X and is stored in server X's cache manager, server Y is unaware of the dependency, and thus would not clear

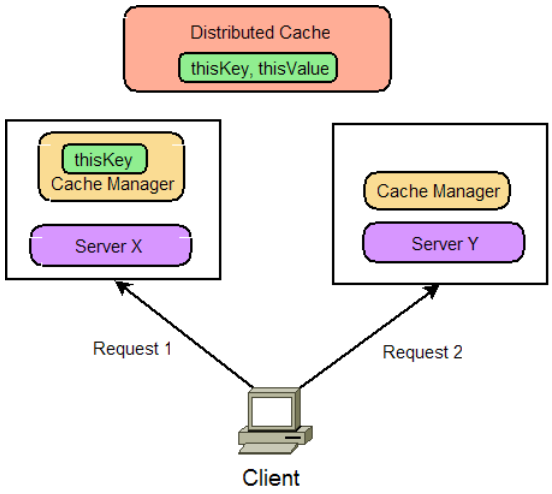


Figure 4.1: Multiple independent cache managers do not work

the cache.

Storing the Graph in Memcached

From this discussion, it can be seen that not only are multiple copies of the cache manager needed, but they each need to be synchronized and they need to store the same information. This sounds similar to the function performed by Memcached. Memcached runs on multiple nodes, and allows for stored data to be accessed by all web nodes. Unfortunately, the data is not stored reliably - if one of the Memcached nodes fails, then all the data stored in that node is lost. The data would be lost because data stored in Memcached is not written to disk, and it is not mirrored across multiple nodes. This lack of redundancy is not a problem if the data stored in the cache is simply a copy that can be fetched quickly, but it is a problem if we are using the cache as a primary data store (similar to Section 2.4).

Storing the Graph in a Database

What is really needed is a reliable distributed area that could be used to store dependencies. For all other requests, a database is used to serve this purpose. However, as one primary purpose of the cache is to reduce database queries, it is counterproductive to use the database to store metadata for information that is stored in the cache.

4.2 Deploy-Time Dependency Analysis Model

A working dependency graph is necessary for proper operation of the cache. There must be multiple copies of the graph to eliminate a single point of failure, and each copy must be complete at all times so that updates to any node cause the proper invalidations. To accomplish these goals, the fact that application source-code is relatively static can be exploited. Additionally, whereas a cache manager builds up its dependency database gradually on a per-request basis at run-time, the deploy-time method has a full dependency graph before the application is started.

4.2.1 Method Overview

This method works on the principle that all possible dependencies can be identified before an application is published to the live web servers. Before the application code is deployed, the code can be analyzed, and the dependencies identified. These dependencies can then be statically written into the source code of each web node. In this way, each web node can be aware of all the dependencies involved by every update, and thus can issue cache invalidations as appropriate. The concept of a "cache manager" still exists, but it is no longer a process - just an object residing

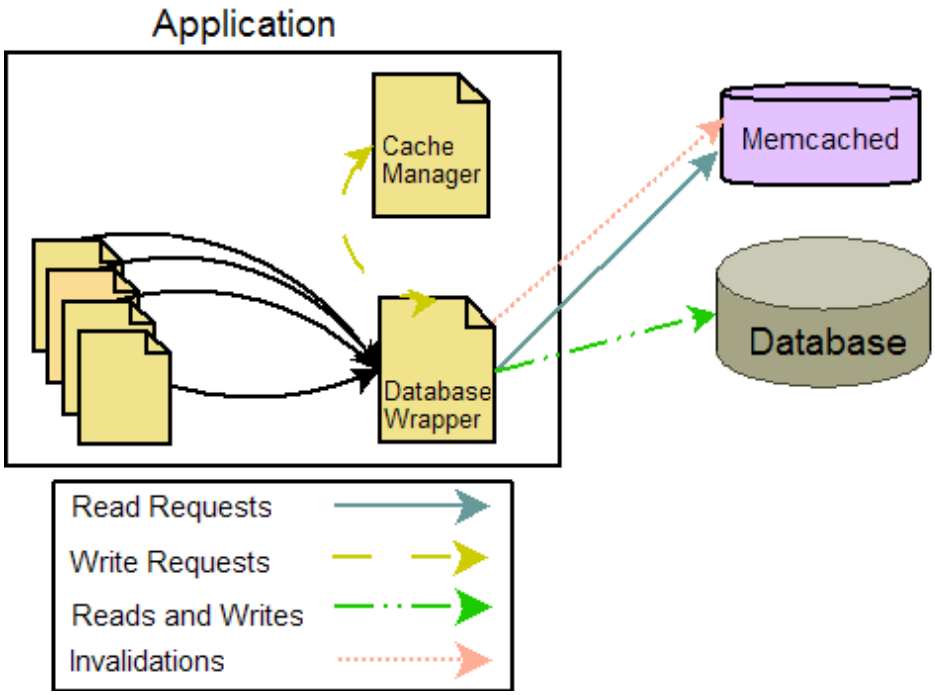


Figure 4.2: Deploy-time method of maintaining cache consistency

on each web node that is aware of all dependencies that are caused by any possible update.

An example of this method is shown in Figure 4.2. This figure shows a single instance of the application with its "cache manager". Assume all database requests are issued through a database wrapper. If the request is a write request, the cache manager is alerted, and is aware if any invalidations are necessary. If they are, the necessary invalidations are performed, and then the writes to the database occur. If the database wrapper receives a read request, it checks Memcached to see if the object is present, and if not, retrieves and processes the appropriate data from the database (then storing it in Memcached).

4.2.2 Limiting the Size of the Dependency Graph

The number of possible cached objects is potentially enormous in a large application. The number of stored objects is usually dependent on the amount of data managed by the system. For example, in a bulletin board environment, there may be one cached object for each thread (a list of posts), in addition to one cached object for each post.

The potential size of the dependency graph presents two problems. First, it is not realistic to expect the cache manager to keep track of all dependencies for all individual objects. This exponential relationship would quickly cause the size of the cache manager to grow to a very significant size and dwarf the rest of the application.

The second problem relates to storing new information in the graph. Each copy of the graph (one copy per web node) contains information on all possible dependencies that can occur in a system. When new data is inserted into the database, such as a new post on a bulletin board, it too has dependencies. The system cannot inform all web nodes that a new dependency has been added, as this would require updating the statically-written dependency information.

The solution to both these problems is to specify the dependency information generically enough so that it can be used to dynamically calculate all possible combinations of cached data. This is best demonstrated with an example. Assume a bulletin board website has a database table "posts" with the information shown in Table 4.1.

The posts table contains five columns: postID, threadID, postedByUserID, content, and date. The postID column is used to uniquely identify each row. The postedByUserID and threadID columns are "foreign keys" - a database term to

postID	threadID	postedByUserID	content	date
...	...	...	...	...
75	12	126	This is a post	2006-10-20 16:20:00
...	...	...	...	...

Table 4.1: Bulletin board site example: posts table

threadID	subject
...	...
12	This example thread probably contains posts
...	...

Table 4.2: Bulletin board site example: threads table

denote that entries in this column depend on entries in another table. In the case of postedByUserID, assume the column is linked to the unique userID field from the "users" table. The last columns, content and date, contain the actual content that the post contains, and the date it was made on.

Also assume that there is a "threads" table that groups together a number of bulletin board posts into a single logical thread. The threads table contains the information shown in Table 4.2. Lastly, assume that a users table exists and contains the information shown in Table 4.3.

A pre-compiled HTML version of the post could be stored in a cache element with a key "post75". A sample version of the value for this key is shown in Code Listing 4.2

The cache element post75 depends on the row in the posts table with ID 75. Whenever that row is modified, possibly due to the user editing the post content, or

userID	name	registrationDate
...	...	...
126	John Smith	2006-09-26 12:00:00
...	...	...

Table 4.3: Bulletin board site example: users table

Code Listing 4.2: Example HTML fragment from the cached element post75

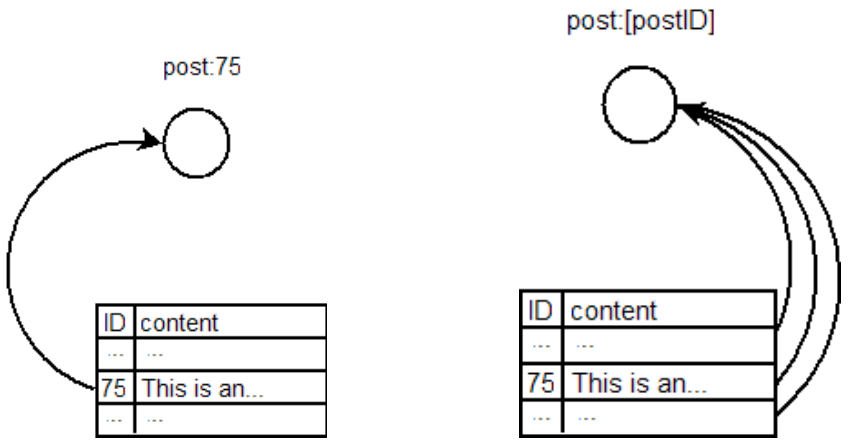
```
<div class="postHeader">Posted By: John Smith<br>
                Posted at: 4:20pm on October 20, 2006
</div>
<div class="postContent">
This is an example post
</div>
```

an administrator deleting the row, then the cached element should be invalidated.

Rather than storing the exact dependencies, we must store dependencies in such a way that the ID value of "75" can be dynamically assigned. For example, the dependency "post[postID]" can be used to denote any cache element of the form "post[postID]", where [postID] is a placeholder for the actual postID. In this way, the cache keys are generic enough to deal with new posts being added at any time, and they do not need to store information for each individual post.

This method requires that extra information is known at the time of invalidation. In the simple example shown in Code Listing 4.1, it was only necessary to know the table/column combination to trigger an update. By allowing the dependency graph to contain variable placeholders, it becomes necessary to fill in these placeholders with actual values at the time of invalidation. This is shown graphically in Figure 4.3. In Figure 4.3(a), a dependency is created directly from the posts table with postID 75 to the cached element with the name post75. In Figure 4.3(b), all post[postID] cache elements are dependent on their corresponding entries in the posts table.

To further complicate this example, an additional cache element can be created. Assume that the cache element thread[threadID] is added. This cache element will contain the pre-compiled HTML for an entire page of posts. This cache element will depend on all of the posts that belong to that particular thread. It will be seen in Section 4.2.3 how this relationship is maintained, but for now, it is simply used as an example to demonstrate that a cache element may depend on other cached elements. This relationship can be seen graphically in Figure 4.4.



(a) A simple dependency graph

(b) A dependency graph with placeholder variables

Figure 4.3: The difference between simple and placeholder dependency graphs

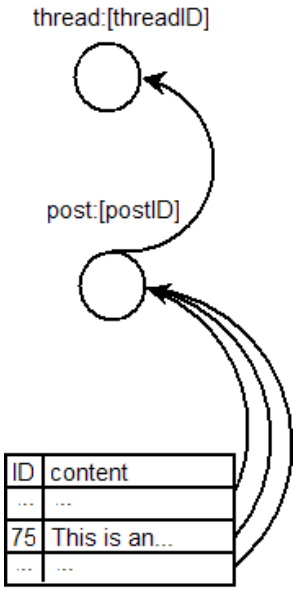


Figure 4.4: A more complicated dependency graph

4.2.3 Invalidating Complex Cached Elements

The more generic a cached element is, the more underlying data it depends on. In some cases, the relationship between an underlying data element and a cache entry is not immediately obvious. This can be demonstrated using the same example as above, and the same HTML code snippet shown in Code Listing 4.2.

In this example, the name of the person posting is cached along with the post content. Not only does the cached element depend on the content, but also the name of the poster. If the poster changes their name, then all cache elements that contain that name must be invalidated.

This relationship needs to be supplied to the invalidation mechanism so that only the necessary cache elements can be invalidated. This relationship takes the form of a SQL query, as will be seen in Section 4.2.4.

4.2.4 Implementation Details

The main requirement for this system is that the code must specify the dependencies that cached elements have on the underlying data. For more complicated dependencies, such as discussed in Section 4.2.3, a SQL query is also necessary to define the relationship between the underlying data and the key of the cached element.

These dependencies are specified using comments in the code. This allows the code to still be executed in a development environment before the distributed cache is used. A simple dependency specification can be seen in Code Listing 4.3. This line would appear when the cache element post[postID] is set.

Code Listing 4.4 shows another line that would appear when the post[postID]

Code Listing 4.3: A simple dependency specification

```
//post[postID] depends on posts.content |
```

Code Listing 4.4: A more complex dependency specification

```
//post[postID] depends on users.name |
select postID from posts where posts.postedByUserID=[userID]
```

cache element is set. This is a more complicated dependency specification, as can be seen by the SQL query after the pipe. This dependency says that the post[postID] cache element depends on the name of the user who made that post. The SQL query is used to describe the relationship between the cache element and the specific posts made by the user.

After the dependencies for the necessary cache elements have been specified, the code can be analyzed using the analyzer application. This application is a two-pass Perl script that uses the Perl "Graph" module [Hie04]. In the first pass, each source file is analyzed for dependency specifications. The dependencies are stored in a directed graph, with the underlying data as source nodes. The source nodes point to any number of cache element nodes, which can also point to other cache element nodes.

Before the application performs a second pass, it creates a function for each intermediate node and sink node in the graph. These are all nodes that correspond to cache elements. This function performs two actions:

1. Invalidates the elements this node is responsible for in the cache
2. Calls invalidation functions for all nodes that depend on this node

Each function is named according to the cache element it invalidates. For example, the function to invalidate the cache element "rubbos:storiesOfTheDay" is named "invalidaterubbosstoriesOfTheDay". This is simply the name of the cache element (with all punctuation removed), with a prefix of "invalidate". (This does create the possibility of name collisions, which currently results in an error, but could easily be fixed by adding a counter or ID number to the function name).

This set of functions, one for each node (representing one for each type of cache element), is written to an output file. This output file is included by all the project files that perform database updates, so that they will be able to call the functions in this file.

Invalidating the Appropriate Cached Elements

The first action performed by the generated function for each node is to invalidate the cache elements represented by that node. As previously discussed, not all cache elements will be invalidated - only the cache elements whose underlying data has changed.

To determine exactly what elements to invalidate, each function accepts an array of hashes as a parameter. Each element in the array specifies the parameters for a cache element to invalidate. The hash is used to specify the parameters. For example, the array could contain the information in Code Listing 4.5.

This array contains two elements, which identifies two cache elements to invalidate. One has a 'postID' parameter with a value of 4, and the other has a 'postID' parameter with a value of 12. This array could be passed into the function generated for the post[postID] cache element. It would cause the cache elements post4 and post12 to be invalidated.

Code Listing 4.5: A sample array passed to an invalidation function

```
[ {postID => 4},
  {postID => 12}
]
```

Code Listing 4.6: Dependency statement for the thread cache element

```
threads[threadID] depends on post[postID] |
select threadID from posts where postID=[postID]
```

Calling Other Invalidation Functions for Dependent Nodes

The second purpose of the generated function is to call the generated functions for nodes that are dependent on the current node. The example displayed graphically in Figure 4.4 shows that the thread[threadID] element depends on the post[postID] element. The dependency statement would be written as shown in Code Listing 4.6.

In this example, the function that invalidates the post[postID] elements would also be responsible for calling the function to invalidate the appropriate thread[threadID] element. It would need to build an array of hashes to pass into the function by using the SQL query shown in Code Listing 4.6. It would substitute the value of the current postID it is invalidating for [postID], and execute the SQL query. The query would return a threadID value that would be passed into the function to invalidate the thread[threadID] cache element.

The Second Pass

After the invalidation functions have been generated, it is time for the application to make its second pass through all the source files. It looks for all update/insert/delete SQL queries. For each query it discovers, it identifies the table and columns that are being affected, and looks at the dependency graph to see if those columns are source nodes in the graph. If they are not in the graph, then the query is ignored (as there is no cached data that depends on it). If the node is in the graph, then an array of hashes is generated, and passed into the appropriate function to perform the necessary invalidations.

4.2.5 Implementation with Database Triggers

Rather than analyzing the application source code to find database changes, it is also possible to identify these changes using database triggers. This has the added advantage of reducing the complexity of the analysis application. However, there are a number of drawbacks that make such a design unattractive.

Although the complexity of the analysis application is reduced, much additional complexity is added to the database. To implement this solution, triggers would need to be created for every database column and table, for each type of action (insert/update/delete). These triggers would then need to call application code that would perform the necessary cache invalidations. The creation of all the triggers would require significant extra work.

Additionally, the system would not scale as easily. If a new cached element was added that depended on a new underlying data column, then a new set of triggers would need to be added to the database for that column. With the proposed

version of the deploy-time method, no additional steps are needed - the dependency would be automatically identified by the analyzer application. New logic would be written into the application, but this would happen at deploy-time, and would be transparent to all parties involved.

Another limitation of database triggers is that they require the use of a database that allows triggers to call outside code. This limitation may be acceptable for some applications, but certainly not all. Methods to avoid this, such as having an "invalidation process" that monitors the database for updates, is not an effective solution for the reasons discussed in Section 4.1.1.

4.2.6 Deploy-Time Method Versus a Cache Manager

It is beneficial to pause for a moment and reflect again on the cache manager method discussed in Section 4.1.2. The main difference between using a cache manager and using the deploy-time method is that with the cache manager, there is a separate process that invalidates the cache, and with the deploy-time method, the application performs the invalidations directly. Both methods require code in the application to set dependencies, and both methods require the application to perform some event when the underlying data is updated (either to inform the cache manager, or perform the necessary invalidations directly). The deploy-time method eliminates the need for building a highly available application (the cache manager) by leveraging the distributed nature of the application source code. The deploy-time method also eliminates the need for the developer to write code to inform the cache manager of updates, since this code will be written automatically at deploy-time.

4.2.7 Method Summary

The use of the deploy-time method frees developers from needing to manually track cache dependencies. Rather, the dependencies are identified at deploy-time, and invalidation code is statically written into the application in the appropriate places. Although the process of identifying dependencies and adding invalidation code is not free, its cost is only experienced once during deploy time. It eliminates the risk that application designers will make code additions or modifications that will place the cache in an inconsistent state. The existing implementations of Memcached discussed in Section 3.1 required invalidation statements to be manually added to various places throughout the code. The deploy-time method automatically identifies dependencies and performs the necessary invalidations.

4.3 Maintainability of the Deploy-Time Method

To measure the maintainability of the deploy-time method, a number of trials were conducted. These trials each consisted of a specific modification that was to be made to a particular web system. The modifications were made to both the system that made use of the deploy-time method, and the system that used the standard, unstructured approach. In Chapter 5, two web applications are used to measure the run-time performance of the deploy-time method. These two web applications are called RUBBoS and RUBiS. These two web applications, and a third web application, iFinance, are used to measure the maintainability of the various methods. More details about the web applications are available in Chapter 5. Summary statistics showing the results from the maintainability trials are provided following the discussion in Table 4.4.

A mix of two general types of tasks were performed, and statistics were recorded on the number of lines of code changed, and in how many different sections modifications or additions were needed. The two basic types of tasks were:

1. Caching a new element - This type of task involves adding a new element to the cache, and all the appropriate supporting code to ensure that the new cached element is never inconsistent with the underlying data.

2. Modifying the application - This type of task involves modifying the application in some way. For example, a new feature could be added, or new functionality could be exposed to the end users. Care must be taken to ensure that information in the database is not updated without invalidating the appropriate cache elements.

4.3.1 RUBBoS Task 1: Cache Post Comments

Currently with RUBBoS, post comments are not cached. A comment consists of the name of the person who made the comment, the comment date, a score (assigned by moderators), and the actual comment content. Similar to slashdot, once a comment has been posted, it cannot be edited or deleted.

With both methods, the cache elements were added in two locations - when viewing the main story, and when viewing comments in detail. (These files, ViewStory.php and ViewComment.php, correspond to Slashdot's article.pl and comments.pl files).

The next step was to identify all the places in the code where the underlying data changes. In these places, code needed to be added to invalidate the appropriate

cache elements. The only time the cached data can change is when it is moderated (StoreModeratorLog.php). With the deploy-time method, this dependency is automatically recognized, and invalidation code is added accordingly.

4.3.2 RUBBoS Task 2: Allow Users to Edit their Post Comments

Digg.com is another website that uses Memcached. In many ways, it is seen as a competitor to Slashdot. One of Digg's features is the ability to edit comments (with Digg, this feature is limited to a short time period after posting). Currently, Slashdot and RUBBoS lack this ability.

To add this feature, StoreComment.php was edited to allow for the updating of comments, and ViewComment.php was updated to provide an interface to do so.

Once this functionality was added, care was used to ensure that editing a comment did not place any cached elements that use that comment in an inconsistent state. In this case, the element of concern was the cache element set in Section 4.3.1. This step is handled automatically by the deploy-time method, but needs manual coding for the unstructured method.

4.3.3 RUBiS Task 1: Cache the "About Me" Page

The RUBiS "About Me" page is very similar to the "My eBay" page provided by eBay. It contains information about the recent items the user has bid on, any recent items won, any recent items sold, in addition to any feedback left or received. Caching this page would result in a significant amount of savings, as it takes a

significant amount of time to generate all the information for that page. Unfortunately, keeping the cache element up-to-date is more difficult than the previous examples, as the cache element relies on a larger amount of underlying data.

Adding the cache element was easy enough, and only required minor modifications to the AboutMe.php file. However, invalidating this cache element required hunting around for all the locations where the underlying data is modified. This involved adding code to the StoreBid.php, StoreBuyNow.php, and StoreComment.php files. Using the deploy-time method, this step was not necessary.

4.3.4 RUBiS Task 2: Allow Users to Edit Feedback

Currently with eBay, if you leave feedback for a user, you can later adjust that feedback. For example, if you left a terse message after being frustrated with a seller, you could later adjust that feedback once the issue was resolved. Currently, this is not possible with RUBiS. Once feedback has been left, it cannot be edited. To better mirror eBay, RUBiS could be adjusted to allow the editing of feedback.

After adding the code for the interface, and the code to update the database table, all the cache elements must be checked to see if they depend on the updated information. This step is not necessary for the deploy-time method, as it is handled automatically. There are a few cache elements that depend on the updated comment - the "View User Information" comments page, and the "About Me" page. Both the poster and postee need to have their cache elements invalidated.

4.3.5 iFinance Task 1: Caching Course Structure Elements

The deploy-time system was used in a real-world environment with the "iFinance" product offered by Wilfrid Laurier University and Tempic Inc [Tem06]. This is a fully online textbook with an integrated algorithmically-generated question database for self-test quizzes, and a bulletin board for TA support.

In the iFinance product, course structure elements are the "fundamental building blocks" of a course. The overall course structure is composed of chapters, sections, and subsections. Each page of a textbook belongs to one of these elements, as does every practice problem, every quiz question, and every bulletin board post.

Course structure elements are used in many places, and certain pieces of them are already cached in various places (For example, the textbook chapters are already cached, as are categories for quiz questions). Adding the course structure cache element was fairly straightforward, as many methods already called an existing function to get course structure information directly from the database. However, different pieces of this cached information were being changed in many different places throughout the code. For example, making bulletin board categories visible/hidden was an action that occurred through the bulletin board, while enabling/disabling sets of questions for self-test quizzes was an action performed through the self-test quiz module. Identifying all these locations resulted in a huge amount of work in the unstructured case, while none of this effort was necessary when using the deploy-time method.

In the unstructured case, the amount of work was so unreasonable that the only way it could reasonably be performed was by first using the deploy-time method, and then using the "diff" utility to identify all the places in the code that had been modified. Much effort was then spent on updating all the identified sections

of the application to invalidate the necessary elements in the cache. Even though all the necessary locations were identified by the deploy-time method, it was still a very time consuming and manual process to write the proper invalidation code in all appropriate locations. Without using this approach, it is unlikely that all the appropriate places in the code could be identified and updated accordingly.

4.3.6 Summary

Table 4.4 shows a summary of the work that was performed to measure the maintainability of these projects. The number of lines of code added or modified was measured, as was the number of locations in the project that were affected. A "location" was a group of lines that were added or modified together. Many times, multiple lines of code were needed in one place to accomplish a task. When those lines were together in one file or area in a file, they were treated as a single location.

For small projects, the benefits of the deploy-time method are noticeable, but not great. When looking at iFinance, the real benefits of the deploy-time method can be seen. As projects grow larger, each change has the ability to impact a greater number of facets in the system. Thus, larger applications stand more to gain by using a structured method (like the deploy-time method) to manage the cache dependencies. The benefits of the deploy-time method are most clearly seen with large projects, as the deploy-time method frees the developer from needing to be aware of all the data interdependencies.

With the unstructured method, the number of locations modified depends on the number of places in the code that touch data that any cached elements depend on (either directly or indirectly). Each of these locations will need to be modified to ensure the appropriate cached elements are invalidated. With the deploy-time

Trial	Lines Added/Modified		Locations Modified	
	Unstructured	Deploy-Time	Unstructured	Deploy-Time
RUBBoS - Task 1	13	10	3	2
RUBBoS - Task 2	31	28	3	2
RUBiS - Task 1	22	4	5	1
RUBiS - Task 2	42	22	6	2
iFinance - Task 1	237	14	28	2

Table 4.4: Maintainability summary

method, these locations are identified automatically, and the invalidation code is added at deploy-time. This frees the developer from needing to be aware of these dependencies, and it can greatly reduce the number of modifications to the code that are required.

This chapter discussed the deploy-time method and its implementation details. The chapter concluded with an examination of the maintainability improvements offered by the deploy-time method. The following chapter discusses run-time performance tests that were conducted to measure how the deploy-time method performed at run-time.

Chapter 5

Run-Time Performance

In Section 4.3, it was seen that the deploy-time method was far more maintainable than the unstructured method, especially for large systems. This chapter discusses run-time performance characteristics for the deploy-time method. Additional maintainability is no boon if it comes at too high a run-time performance penalty (in the form of increased response time for users, or a system that can handle a fewer number of users). In this section, the deploy-time method is compared to the unstructured Memcached method, systems that do not use a distributed cache, and systems that use a database cache.

Two websites were used in a simulated environment to produce detailed results for analysis. The two websites were first used in [ACC+02] to study the effect of load on dynamic websites. Both websites and testing frameworks are available online under the GNU Lesser GPL license [CM02b, CM02a]. The two websites are named "RUBBoS" (Rice University Bulletin Board System) and "RUBiS" (Rice University Bidding System). RUBBoS is a slashdot-like bulletin board, and RUBiS is an eBay-like auction website.

From a code perspective, RUBiS is more complicated than RUBBoS. RUBiS has more complicated pages, which require more processing on the web server. For example, the "View Item" page contains information about the item in question, in addition to the bid history for that item. RUBiS also contains an "About Me" page that requires significant processing, as it contains a list of the items a user has sold, all items that user has bid on, and all the feedback left by/to the user.

The RUBiS database employs transactions to guarantee ACID semantics, which are necessary for a website that deals with money. Conversely, RUBBoS does not need or use any transactions. Although transactions are necessary to ensure that multiple users do not interfere with each other when bidding, ACID semantics do not come for free. This cost will likely be observed through high database CPU usage.

Each test environment consisted of one web server node, one database node, and multiple client machines to generate load. All machines were running Linux (kernel 2.6.15-27-386). The web server and database server were 3.2 GHz Pentium 4 machines with 3 GB of RAM. All machines were connected via a 1 Gbps full duplex switch on an isolated network.

Additional "real world" test results are also reported in Section 5.3. These results were obtained from a web system that was used for a high-demand online learning finance website [Tem06]. The following sections provide more detail about the tests and the results.

5.1 RUBBoS

RUBBoS is a bulletin board system very similar to slashdot. The server application is programmed in PHP and makes use of a MySQL database. For the test, stories and comments are generated using a dictionary. Their length varies between 1KB and 8KB. A zipf-like distribution is used to determine the exact length, as short stories/comments are more common than long ones. A total of 500,000 users are simulated, of which 50,000 (10%) have moderator access [CM02b]. Each day has an average of 15-25 stories, and each story has between 20-50 comments [Mal01]. More information about the benchmark is available in [ACC+02].

5.1.1 RUBBoS Load Generating Tool

The RUBBoS system is packaged with a load generator tool. This tool assumes some sort of shared storage (for example, using the "sshfs" filesystem) that is used by all the client machines to read configuration information, and write test results. The application is started on one client machine, which then uses "rsh" to launch copies of itself on all the remote clients. Each client machine generates a set number of client threads, which generate load by making queries to the web server. Each client thread operates on a "session" principle. A request is made, and the client waits for a reply. After a response is received, and after a think time (average of 7 seconds [WDS00]), the client thread determines with pre-determined probability what type of follow-up request to make, or whether to end its session.

5.1.2 Results - Overview

The load generator application ran tests with the number of clients ranging from 25 to 1000. The number of completed requests for each test with a different number of clients is shown in Figure 5.1. Shown in the figure is the base case (labeled "No Memcached"). This is a standard setup without any types of additional caching. Next, a trial with the database query cache enabled was performed. In general, the database query cache did not improve performance in any noticeable manner. This is likely due to the overhead of maintaining the cache, and the fact that the query cache for each table is cleared whenever any row in that table is changed [MyS06a]. Both of these methods responded to the maximum number of requests when 400 clients were used to generate load. More than 400 clients resulted in a decrease in the number of successfully completed requests, due to the servers being excessively loaded.

The other two trials were performed with Memcached enabled. The first method used Memcached in the "standard" unstructured way, and the second trial used Memcached with the deploy-time method previously discussed. There is very little difference in performance between these two methods. The maximum number of responses was reached with 500 clients generating load, and the throughput decreased as more clients were used to generate load. As more clients were added, the Memcached trials were still able to process more requests than the trials without Memcached.

Figure 5.2 shows the average response time (as measured by the clients) of the different methods. In general, when Memcached is used, the response time is faster. Again, there is very little difference between the manual method of using Memcached and the deploy-time method.

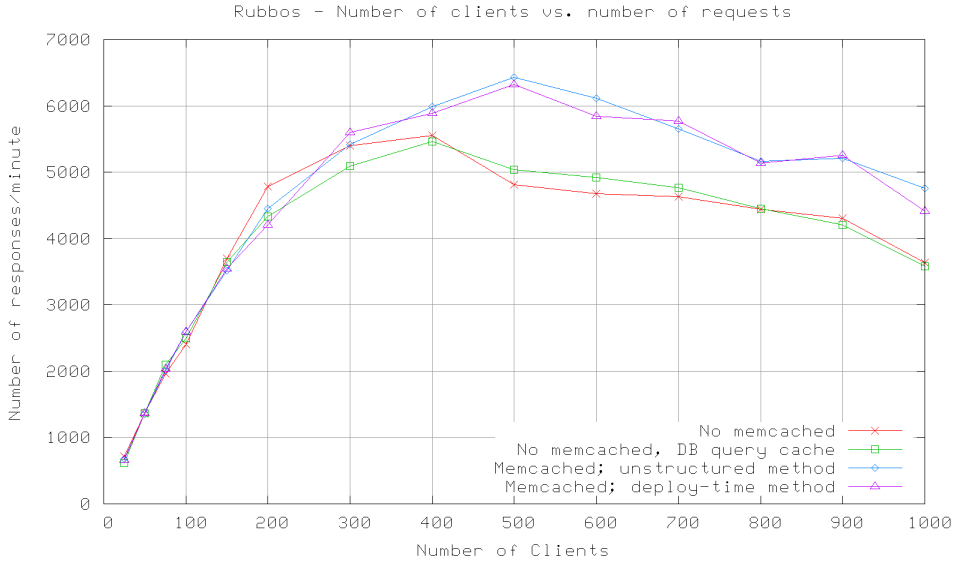


Figure 5.1: RUBBoS - number of clients vs. number of completed requests

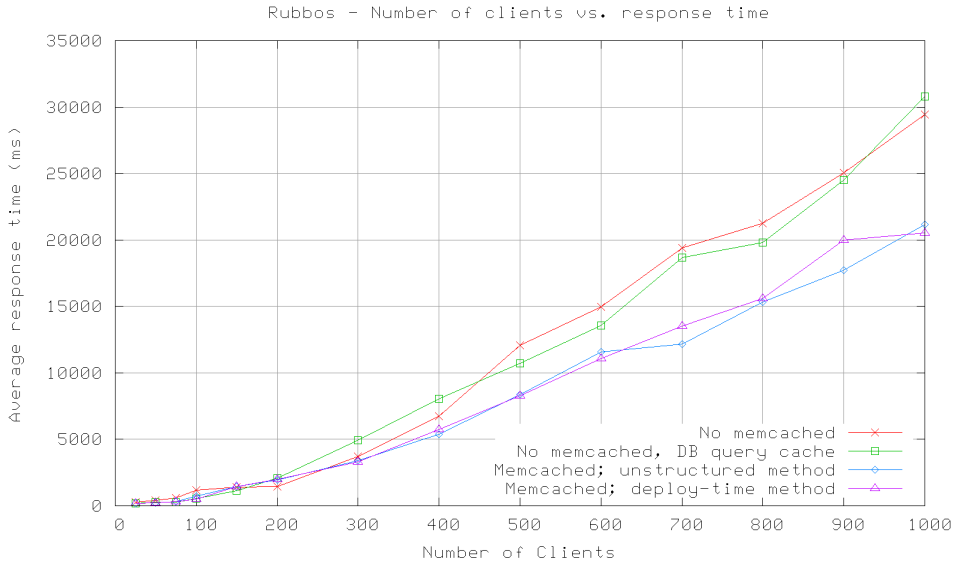


Figure 5.2: RUBBoS - number of clients vs. response time

5.1.3 Results - Detailed Analysis

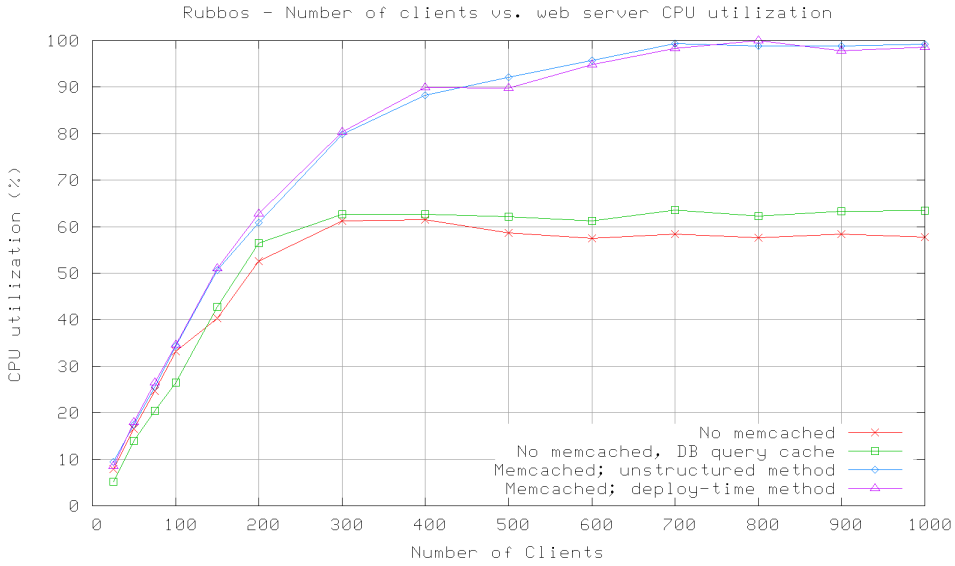
Section 5.1.2 showed that using Memcached (either manually, or with the deploy-time method) can result in faster response time, and more clients being able to use the web application. However, the use of Memcached also has additional benefits and consequences that can only be seen when looking at the memory and CPU utilization of the server machines. This section identifies those differences.

CPU Utilization

Figure 5.3(a) shows the CPU utilization of the web server under the different trials, and Figure 5.3(b) shows the CPU utilization of the database server. The first observation that is immediately apparent is that the limiting factor switches from the database server's CPU when Memcached is not used to the web server's CPU when Memcached is used. The benefits of this may not be immediately apparent, but this is a huge boon to system designers. The use of Memcached reduced the burden on the database server, and shifted much of the load to the web servers. Database servers are often more costly (both in terms of hardware, and maintenance due to replication) than web servers. Web servers are simply stateless nodes that can be very easily replicated. As a web application grows, Memcached allows fewer database nodes to be used.

It can also be seen that without Memcached, the database server reaches saturation with approximately 300 clients generating load. With Memcached, the web server (which is now the point of contention) reaches saturation with between 600-700 client nodes.

Another observation that can be made is that there is very little difference in CPU utilization whether the database query cache is used or not. There is also no



(a) Web server CPU utilization

(b) Database server CPU utilization

Figure 5.3: RUBBoS - server CPU utilization

difference in CPU utilization depending on what Memcached method is used.

Memory Utilization

Figure 5.4(a) shows the memory utilization of the web server under the different trials, and Figure 5.4(b) shows the memory utilization of the database server. It can be seen that memory was never a limiting factor in these trials, as both the web server and database server were equipped with 3 GB of RAM. Looking at the web server memory utilization in Figure 5.4(a), we can see that Memcached causes an additional 100 MB of memory to be used, regardless of the number of clients. This is due to the way Memcached operates. When the Memcached process is initialized, it immediately reserves a fixed amount of memory. The maximum amount of memory it can use is configurable, but in these trials, it did not use more than the initially allocated amount.

Memcached usually resides on web nodes. Although the extra memory usage could be seen as a disadvantage, web nodes are typically CPU hungry and memory light, meaning they often have extra memory to spare for Memcached processes. This figure simply shows that Memcached does consume some memory.

A number of interesting observations can be seen by looking at the database memory utilization shown in Figure 5.4(b). Firstly, it can be seen that enabling the database query cache causes the database server to consume significantly more memory than all other trials.

When compared to the "base case" (where no caching is used), it can be seen that using Memcached causes a reduction in the amount of memory used by the database server. This is due to the fact that the database server is being queried far less frequently, and thus does not have as many open connections at any given

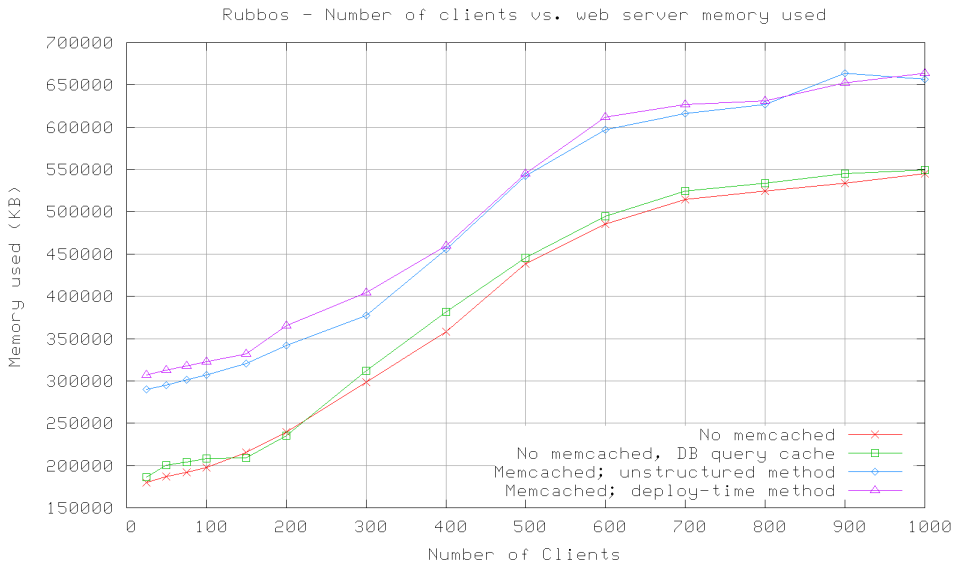
time (as compared to the base case).

5.1.4 Cache Analysis

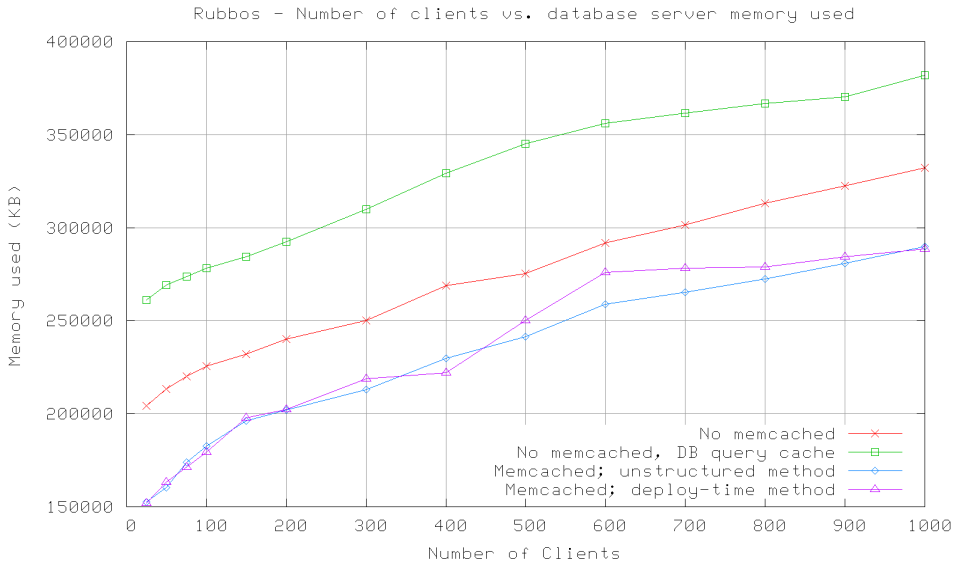
By analyzing the contents of the cache during runtime, a picture of what the cache was actually doing could be obtained. For this reason, modifications were made to introduce a logging layer between the application and Memcached. Four types of operations were logged: Cache hits, cache misses, cache writes (when a new object is added to the cache), and deletes (invalidations).

It should be noted that the method used to perform this logging serialized all operations through the file system, by writing all results to a log file. This decreased the performance of each run by approximately 10%. For this reason, the obtained results are not representative of the test trials presented earlier. However, the results shown here still provide an excellent window into the operation of the cache.

Table 5.1 shows the number of each type of operation that was recorded during the 30-minute duration of this test, with both 50 and 500 clients. Of particular interest with this website is the very low number of cache deletes (invalidations). This is due to the simplicity of the workload, and the type of information that was cached. For this test, the only types of cached elements that could be invalidated were cached user names, and the main "Stories of the Day" page. It is very rare for a member to change their user name, and the "Stories of the Day" page is only updated when a site administrator approves a new story for the front page (in a similar manner to Slashdot). For this reason, the number of cache invalidations is very low. This allows for a very high number of cache hits compared to the number of invalidations.



(a) Web Server Memory Utilization



(b) Database Server Memory Utilization

Figure 5.4: RUBBoS - server memory utilization

	50 Clients	500 Clients
Cache Hits	38187	1309324
Cache Misses	11223	112993
Cache Writes	11223	112993
Deletes	62	588

Table 5.1: RUBBoS cache statistics

One consequence of this is the impact Memcached has on the system: it is quite large. As observed, Memcached eliminates the database as the bottleneck in the system. As will be seen in Section 5.2, under different websites and workloads, the effects of Memcached are different.

A more detailed view of these operations can be seen in Appendix B. Figure B.5 and Figure B.6 show these operations, and how they vary with time. In general, the number of cache hits outnumbers the number of writes and cache misses, while the number of invalidations remains minimal.

Figure B.1 and Figure B.3 show the size of the cache, and the number of elements in it at any given time. When 50 clients are used to generate load, the maximum cache size only reaches about 100 kB (meaning only 100 kB of data is cached at any given time). New data is being added to the cache very slowly, as a limited number of clients are making requests for new data. With 500 clients, the cache grows to store approximately 120,000 cache elements, consuming 2 MB. From these numbers, it can be calculated that the average size of a cached object is 16.7 bytes.

5.2 RUBiS

RUBiS is a bidding website, similar to eBay in style and function. Members can post items for sale, as well as bid on items that other members have posted for sale. Each auction lasts a specific period of time (typically one week). During this time, members can browse for and view items, placing bids on an item if it is something they are interested in. After the close of an auction, members can leave feedback for other members, and rate how the sale proceeded. Sample data was obtained by observing statistics on eBay's website. The database contains approximately 33,000 items in eBay's 20 categories and 62 regions, and 1,000,000 members to bid on these items. Each item had an average of 10 bids, and it was assumed 95% of users left feedback for transactions they were involved in. The database was approximately 1.4 GB.

The same load generator tool that was used to test RUBBoS was also used to test RUBiS. The tool worked in the same way - independent clients initiated request sessions with the server, following links according to pre-determined probabilities to mimic a typical user [ACC+02].

5.2.1 Results - Overview

Similar to RUBBoS, the load generator application ran tests with the number of clients varying from 25 to 1000. The number of completed requests for each test with a different number of clients is shown in Figure 5.5. Shown in the figure is the base case (labeled "No Memcached"). This is a standard setup without any types of additional caching. Next, a trial with the database query cache enabled was performed.

For this website, the database query cache slightly reduced performance. The base case reached its peak number of responses when 300 clients were used, while the trial using the database query cache reached its peak number of responses when 200 clients were used to generate load. By looking at the response time experienced by the users in Figure 5.6, it can be seen that the response time was longer for almost all numbers of clients when the database query cache was enabled. This reduction in performance by the database query cache is due to this site's particularly intensive database workload. The use of transactions increases the amount of work necessary for each query. Coupled with the overhead needed to maintain the database's query cache, this resulted in the reduction in performance.

The other two trials shown in the figures were performed with Memcached enabled. The first method used Memcached in the standard unstructured way, while the second trial used Memcached with the deploy-time method. Both methods provide very similar improvements in both performance and response time. The number of clients that results in the maximum number of responses is 500 for both methods. It can also be seen that as more clients are added, the response rate and response time are more desirable when Memcached is used, compared to the cases where Memcached is not used.

5.2.2 Results - Detailed Analysis

CPU Utilization

Figure 5.7(a) shows the CPU utilization of the web server under the different trials, and Figure 5.7(b) shows the CPU utilization of the database server. For all trials, the database server was the bottleneck. Unlike RUBBoS, the bottleneck was not shifted from the database server to the web server. This demonstrates how

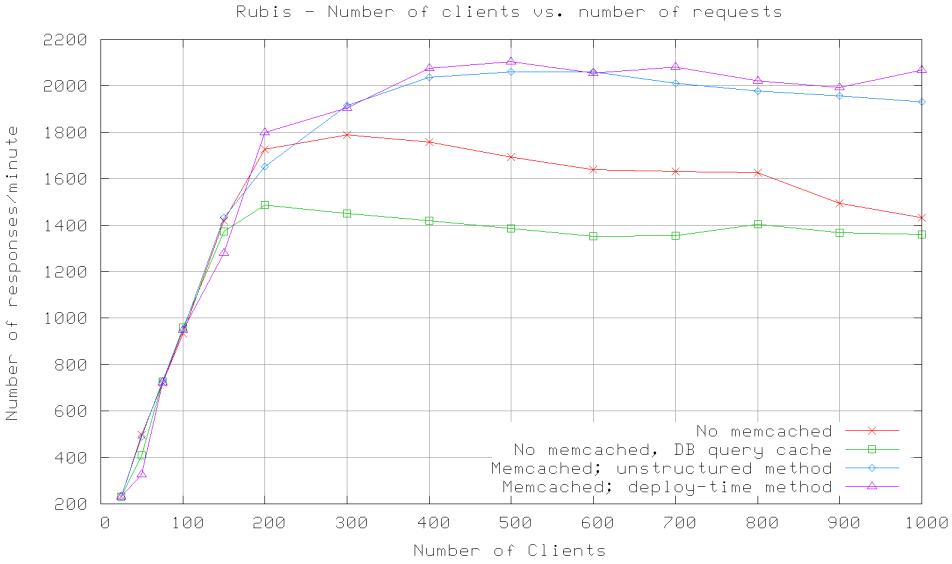


Figure 5.5: RUBiS - number of clients vs. number of completed requests

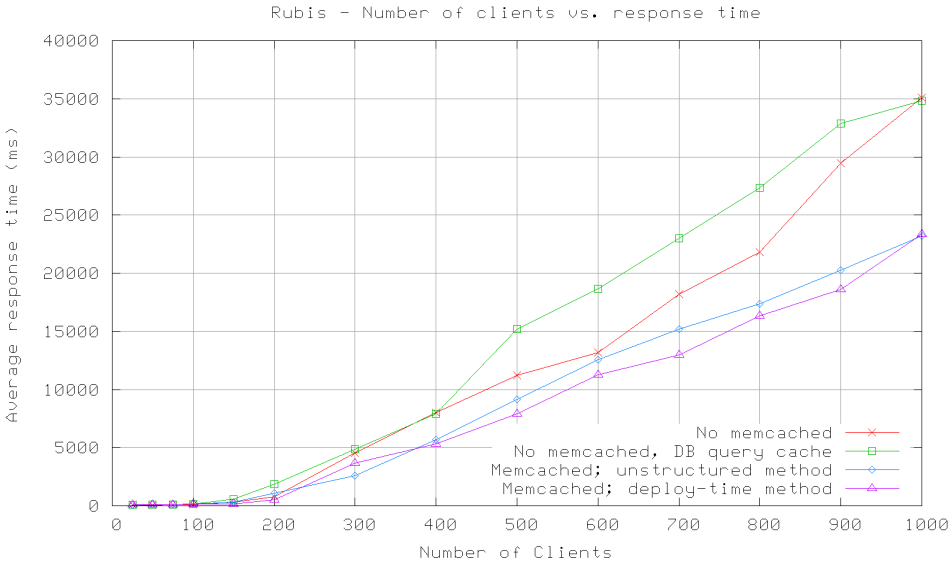
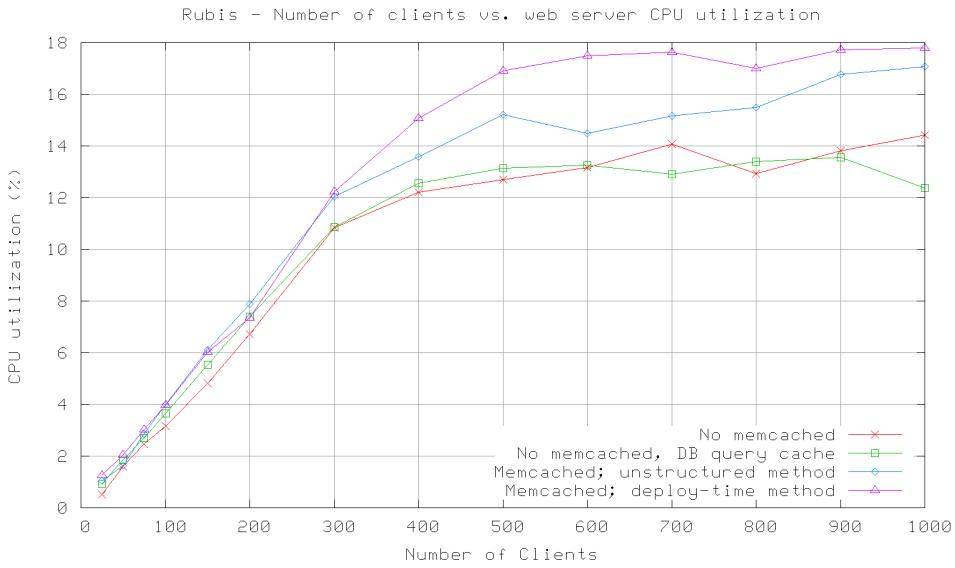
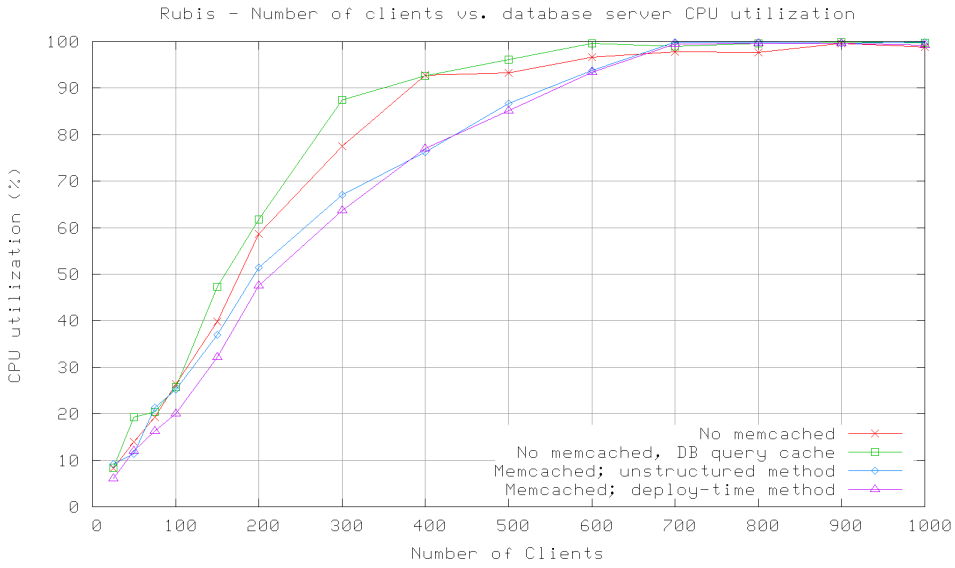


Figure 5.6: RUBiS - number of clients vs. response time



(a) Web server CPU utilization



(b) Database server CPU utilization

Figure 5.7: RUBiS - server CPU utilization

database-intensive the workload for this website was.

Memcached reduced the CPU utilization on the database server between 10-25%, depending on the trial. Without Memcached, the database server reaches saturation when between 600 clients are used to generate load. This compares to 700 users with Memcached enabled. There is no great difference in CPU usage on the database for either method of using Memcached. It does seem that enabling the query cache on the database server causes a slight increase in CPU utilization compared to the base case of no query cache, but this increase is not dramatic.

By looking at the CPU utilization of the web server, it can be seen that the web server was hardly utilized regardless of the number of clients. The maximum utilization of the web server approached 18%. Similar to the RUBBoS trials, using Memcached increased the CPU utilization. However, in this case, the increase in utilization was not nearly as great as with RUBBoS. Although Memcached increases the CPU utilization, in this case, that cost is offset by some of the savings achieved. RUBBoS cached mainly very simple things (such as the username associated with an account). Conversely, RUBiS caches more complex objects (such as a "bid history" table for an item). Items like the bid history table take time and processing power (on the web server) to generate. By caching these more complex objects, processing time on the web server is saved. It is this difference that accounts for the small (less than 5%) increase in CPU utilization.

Also of interest is the fact that the the deploy-time method utilized slightly more of the CPU than the manual method. This slight penalty is a result of extra "invalidation queries" that the deploy-time method executes that the manual method does not. For example, when a comment is inserted into the database, both the fromUserId and toUserId values (who the comment was from/to) are known. In the manual case, these values are used to invalidate the appropriate cache elements.

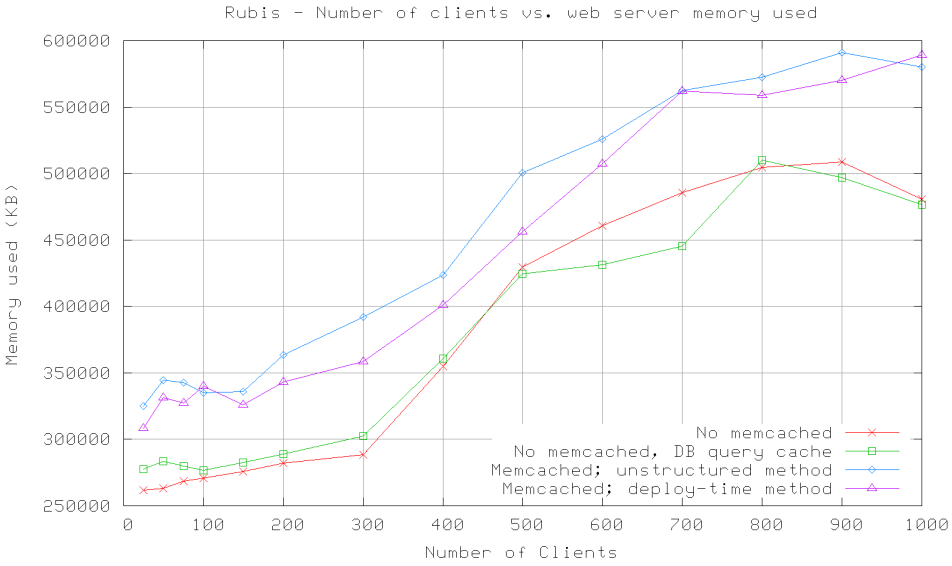
However, the deploy-time method is not smart enough to utilize these existing variables. It notices that a row is inserted into the comments database, and then issues another query to obtain the fromUserId and toUserId. This extra step is the cause of the increased CPU utilization.

Memory Utilization

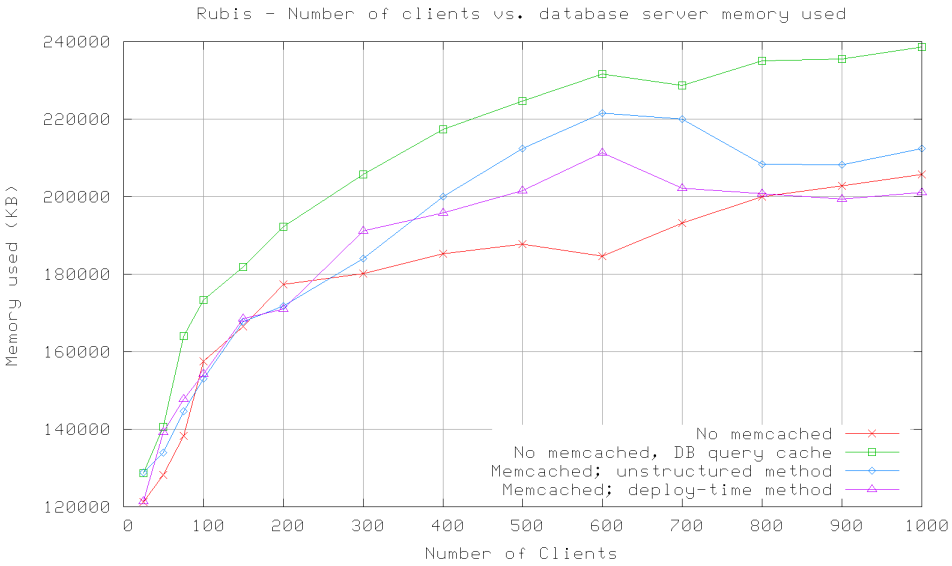
Figure 5.8(a) shows the memory utilization of the web server under the different trials, and Figure 5.8(b) shows the memory utilization of the database server. Similar to RUBBoS, memory was never a limiting factor in these trials.

The web server memory utilization in Figure 5.8(a) shows that using Memcached causes an extra 50 MB to 100 MB of memory to be used. This is due to the memory consumed by the Memcached process.

By looking at the database memory utilization shown in Figure 5.8(b), it can be seen that enabling the database query cache causes the database server to consume significantly more memory than all other trials. Of particular interest on this graph is the increase in memory consumed when Memcached is running on the web server. Rather than reducing the memory requirements of the database server, the use of Memcached increased the memory use on the database server. Examination reveals that the data set used in this test is large enough to not be completely accessed (by the clients, and thus the database), and stored in memory. Memcached allows more clients to be serviced, which results in additional queries being generated, which leads to more data being accessed. It is this phenomenon that leads to the increase of memory on the database server.



(a) Web server memory utilization



(b) Database server memory utilization

Figure 5.8: RUBiS - server memory utilization

5.2.3 Cache Analysis

The same method to monitor cache operations described in Section 5.2.3 was employed to monitor cache operations with the RUBiS test. Table 5.2 shows the number of each type of operation that was recorded during the 30-minute duration of the test, with both 50 and 500 clients. It can be seen that this test had significantly less cache activity than the previous test, with the exception of invalidations. The RUBiS test had fewer cache hits, writes, and misses, but many more invalidations. This, of course, has to do with the nature of the information that was cached. Whereas the RUBBoS test cached small pieces of information like usernames, the RUBiS test cached larger pieces of information that took longer to generate, such as an items "bid history" table.

This limited amount of caching is a result of the application. It is more difficult to cache information on an eBay-like website, as more information is constantly changing. The bulletin board represented a more "read only" website, with user comments being restricted to replies to a particular story, and only administrators being able to post new stories to the main page. RUBiS requires not only more user writing, but also time-sensitive data, such as auction expiry times. The increased complexity of the site meant that more cache invalidations were necessary to cache content. The smaller amount of content that was cached, in combination with the use of transactions, caused the performance improvements not to be as great as in the RUBBoS trials.

A more detailed view of the operations in Table 5.2 can be seen in Appendix B. Figure B.7 and Figure B.8 show the frequency of these operations, and how they vary with time. Similar to the RUBBoS case, the number of cache hits outnumbers the number of writes and misses. However, with RUBiS, there is a noticeable

	50 Clients	500 Clients
Cache Hits	29230	115615
Cache Misses	17530	33333
Cache Writes	17525	33313
Deletes	3206	10940

Table 5.2: RUBiS cache statistics

number of deletes that take place. It can also be seen that as the system starts up, there are a greater number of cache writes. As time progresses, the number of writes decreases. As the test is run, more elements become cached, so the chances of having a cache hit become greater.

Figure B.2 and Figure B.4 show the size of the cache, and the number of elements in it at any given time. When 50 clients are used to generate load, the maximum cache size reaches approximately 5 MB, with 15000 elements cached. It also seems that this is near a "peak". At the 30-minute mark, the size of the cache seems to be leveling off. With 500 clients, the cache grows to store approximately 22,500 cache elements, consuming 10.5 MB. From these numbers, it can be calculated that the average size of a cached object is 467 bytes - almost 30 times larger than in the RUBBoS trials.

5.3 Real World Results

iFinance is a fully online textbook where the deploy-time method made its real-world debut. During the second year offering of iFinance at Wilfrid Laurier University's "Financial Management" course, load problems were experienced during

the few days leading up to the midterms and final exam. Additional servers were needed to handle the load. For the offering in the following year, Memcached was employed to reduce hardware demands.

Since this was a live, real-world environment, detailed analysis statistics are not available. For example, the number of simultaneous clients accessing the system varied at any given time from 0 to over 700. Additionally, the year two enrollment was higher than the year three enrollment by approximately 150 students. Since students were actively using the site as a study tool, each student likely had slightly different usage characteristics and refresh delays. However, the hardware requirements from year two to year three decreased quite dramatically. The time needed by the server to generate each page also decreased significantly when Memcached was deployed. These results are presented here for a number of different actions. These are aggregate results based on an average of over 3 million page requests, obtained from the web server and database logs. The results are shown in Table 5.3.

The "Time" value takes into account the time to generate the page. This accounts for both the time the web server spends generating the HTML, and the time the database spends executing queries.

As can be seen from Table 5.3, the textbook and quiz status pages each issued a very large number of database queries. Many of these queries were from simple scripted queries (often in loops) that relied on the course structure information (textbook chapters and settings). Memcached was able to cache these very effectively, greatly reducing the average number of queries for those pages. Although the time to generate each page decreased in every case, the decrease was not as dramatic as one might have suspected given the dramatic decrease in the number of database queries. Although the database server had far fewer queries to deal with, the end user response time was not greatly improved. This suggests that

Page	Year 2		Year 3	
	Time (ms)	DB Queries	Time (ms)	DB Queries
List of Categories	360.4	617	114.5	42
View a textbook page	559.1	1718	380.0	29
View a question page	361.8	44	302.1	29
Quiz status page	398.4	924	274.4	28
View a quiz page	380.9	51	332.7	28

Table 5.3: Real-world run-time performance results

there are other time consuming steps in the process of generating a webpage, such as formatting and writing the HTML output.

5.4 Summary

In all cases, performance was improved when Memcached was used. The amount of the improvement (in response time, or ability to handle load), and the cost (in CPU or memory consumption) was very dependent on the nature of the website, and the types of information Memcached was used to cache. Memcached resulted in increased memory and CPU utilization on the web server, although in some cases, this increase was offset by Memcached's ability to cache full HTML page fragments.

In all cases, the use of Memcached reduced the CPU utilization of the database server. This effect was most noticeable when the number of clients stressed the non-Memcached trials, but the trials with Memcached were still able to perform well. Memcached also reduced the memory usage of the database server in the first test. In the second test, when the available set of data was larger than the working

set for any given trial, Memcached allowed the web server to handle queries from more clients, increasing the working set of the test, and thus also increasing the memory consumed by the database server.

No major differences were observed between the deploy-time method and the unstructured method of using Memcached, although in certain cases the deploy-time method did incur a slight increase in CPU utilization on the web server. Both methods allow the same performance benefits to be recognized.

This study is significant because it is the first formal study of its kind that examines methods of maintaining cache consistency in a distributed cache environment. It shows that Memcached can be used to increase the scalability of a system without hurting performance or maintainability.

Chapter 6

Conclusions

6.1 The Problem

As a website grows larger, caching becomes an increasingly more desirable way to reduce client load. In such high-demand environments, multiple machines are used to run the web server. Memcached, a distributed object cache, allows the application to share one single cache among all web servers. Memcached is most frequently used in a very unstructured way. Application developers cache elements at various places throughout the code, and then manually search through the code to identify all places this cache element is invalidated. This approach results in making the application more difficult to maintain. Many popular websites that use Memcached resort to keeping a separate text file to describe the keys that are cached, and what their dependencies are.

6.2 Problems with Existing Solutions

A number of existing methods attempt to solve this problem by generating static .html files from the dynamic data, or using some sort of cache manager or log analysis application to identify updates. Generating static .html files is only an acceptable method for particular websites where all clients receive the same page. It is not an apropriate solution for sites that need to provide dynamic data to clients based on a variety of parameters, such as permission level, previous actions, or the contents of their shopping cart.

Cache manager and log analysis systems do not easily scale past one machine. Existing solutions implement the cache manager as a single long-running process that is responsible for invalidating the appropriate cache elements. This process acts as a single point of failure. Log analysis applications rely on reading database query logs, and using those logs to perform invalidations. This is also a single point of failure, and depends on only a single machine generating query update logs.

6.3 Contributions of the Deploy-Time Method

The deploy-time method solves the aforementioned problems by recognizing the dependencies between the cache elements and the underlying data at deploy-time. The application source code can then be analyzed to identify all locations where data updates occur. At these locations, cache invalidation routines are statically written into the source code of the web application to perform the necessary updates.

Using this method, developers no longer need to be concerned about manually invalidating cache elements to ensure the cache is consistent with the underlying

data. Tests demonstrate that using the deploy-time method decreases the number of lines of code that need to be modified when making a change by at least 10% in small applications, and up to 95% in a large application. Larger applications show greater maintainability improvements as they often maintain larger, more complex webs of dependencies.

Run-time performance tests prove that using a distributed cache can offer significant performance benefits. Tests also prove that using the deploy-time method incurs very little or no performance penalty (compared to the unstructured method), and provides all the added benefits of having a more maintainable application. This was the first formal study of its kind.

6.4 Limitations and Future Work

As presented, this method is sufficient for many applications. However, there are some specific cases where additional work would be beneficial. There is a small delay between when Memcached receives an invalidation statement, and when it will accept a new object for the invalidated key. During this short time period, any requests that arrive (that are looking for the recently invalidated key) will need to be served without obtaining that element from the cache. This is similar to a problem discussed in [CDI+04], and would have an identical solution. Rather than performing simple invalidations, new updated data could be generated and immediately placed in the cache.

Another limitation is due to the fact that invalidation functions are written directly into the application. Although this method provides numerous benefits, it also limits the usability of any implementation of this method, as it is language

dependent. Additional code generating routines would need to be supplied for each target language.

Currently, the analysis program does not identify the appropriate updates in all SQL syntax. Currently, only queries that reference a table's primary key fields will be automatically identified as dependencies. If other columns are needed, two queries can be used instead of one. (The first query could return the primary keys, and the second query could use the primary keys to perform the necessary updates). While this is a possible workaround, there are likely particular applications that perform updates on rows using columns other than the primary key. These applications would benefit from full SQL parsing support.

Lastly, this work can be viewed as a starting point for a further, more thorough investigation into the role of caches. Traditional web caches do not apply to dynamic data, since each person receives a different copy of that data. However, it may be beneficial to augment traditional caches (which are often geographically distributed) using the methods discussed here. Such a system could reduce response time, thus improving the overall user experience by making use of geographically distributed "smart" web caches. Such a system could also reduce bandwidth requirements across the network, and may become increasingly beneficial as the general populous begins to demand more dynamic content on resource-constrained handheld devices.

Bibliography

[ABK+03] Mehmet Altinel, Christof Bornhovd, Sailesh Krishnamurthy, C. Mohan, and Hamid Pirahesh. Cache tables: Paving the way for an adaptive database cache. In *Proceedings from the Twenty-Ninth International Conference on Very Large Data Bases*, September 2003.

[ACC+02] C. Amza, E. Cecchet, A. Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *5th Workshop on Workload Characterization*, 2002.

[AKR01] Martin Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Trans. Inter. Tech.*, 1(1):44–69, 2001.

[ASC05] Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 264–273, New York, NY, USA, 2005. ACM Press.

[BDH03] Luiz Andre; Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[Bre01] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Com-*

puting, 5(4):46–55, 2001.

[CDI98] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–30, Washington, DC, USA, 1998. IEEE Computer Society.

[CDI+04] James R. Challenger, Paul Dantzig, Arun Iyengar, Mark S. Squillante, and Li Zhang. Efficiently serving dynamic data at highly accessed web sites. *IEEE/ACM Trans. Netw.*, 12(2):233–246, 2004.

[CLL+01] K. S. Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 532–543, New York, NY, USA, 2001. ACM Press.

[CM02a] E. Cecchet and J. Marguerite. Rice university bidding system "rubis". `http://rubis.objectweb.org/`, 2002. [Online; accessed 27-November-2006].

[CM02b] E. Cecchet and J. Marguerite. Rubbos: Bulletin board benchmark. `http://jmob.objectweb.org/rubbos.html`, 2002. [Online; accessed 27-November-2006].

[Das] Anil Dash. memcached. `http://www.lifewiki.net/sixapart/memcached`. [Online; accessed 11-October-2006].

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *SOSP*

'97: Proceedings of the sixteenth ACM symposium on Operating systems principles, pages 78–91, New York, NY, USA, 1997. ACM Press.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004.

[Fit06] Brad Fitzpatrick. Memcached. `http://www.danga.com/memcached/`, 2006. [Online; accessed 10-October-2006].

[Hie04] Jarkko Hietaniemi. Graph. `http://cpan.org/modules/by-module/Graph/Graph-0.80.readme`, 2004. [Online; accessed 24-November-2006].

[Joo06] Vinjay Joosery. High-availability clustering: How mysql supports 99.999% availability, Sep 2006. Webcast.

[Kaw04] Jalal Kawash. Consistency models for internet caching. In *WISICT '04: Proceedings of the winter international synposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004.

[KD02] Madhukar R. Korupolu and Michael Dahlin. Coordinated placement and replacement for large-scale distributed caches. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):585–600, 2002.

[LKM+02] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, New York, NY, USA, 2002. ACM Press.

[LR00a] Alexandros Labrinidis and Nick Roussopoulos. Generating dynamic content at database-backed web servers: cgi-bin vs. mod_perl. *SIGMOD Rec.*, 29(1):26–31, 2000.

[LR00b] Alexandros Labrinidis and Nick Roussopoulos. Webview materialization. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD inter-*

national conference on Management of data, pages 367–378, New York, NY, USA, 2000. ACM Press.

[Mal01] Rob Malda. Handling the loads. `http://slashdot.org/article.pl?sid=01/09/13/154222&mode=thread&tid=124`, 2001. [Online; accessed 27-October-2006].

[MyS06a] MySQL. How the query cache operates. `http://dev.mysql.com/doc/refman/5.0/en/query-cache-how.html`, 2006. [Online; accessed 20-December-2006].

[MyS06b] MySQL. Mysql. `http://www.mysql.com`, 2006. [Online; accessed 14-October-2006].

[OMG+05] Christopher Olston, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, and Todd C. Mowry. A scalability service for dynamic web applications. In *Proc. Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 56–69, January 2005.

[SKRP01] M. Schrefl, E. Kapsammer, W. Retschitzegger, and B. Pröll. Self-maintaining web pages: an overview. In *ADC '01: Proceedings of the 12th Australasian database conference*, pages 83–90, Washington, DC, USA, 2001. IEEE Computer Society.

[Tem06] Tempic. Tempic's online courseware used by wilfrid laurier university. `http://www.tempic.com/iFinance-2006.html`, 2006. [Online; accessed 10-July-2007].

[Tim] Timeless. Memcached as a session store. `http://lists.danga.com/pipermail/memcached/2006-June/002384.html`. [Online; accessed 11-October-2006].

[Wan99] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, 1999.

[WDS00] Intel Corporation Wayne D. Smith. Tpc-w: Benchmarking an ecommerce solution, 2000.

[Wik06] Wikimedia. Wikimedia servers. `http://meta.wikimedia.org/wiki/Wikimedia_servers`, 2006. [Online; accessed 16-October-2006].

[YBS99] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 163–174, New York, NY, USA, 1999. ACM Press.

[YFIV00] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *The VLDB Journal*, pages 188–199, 2000.

Glossary

ACID Atomicity, Consistency, Isolation and Durability

ASP Active Server Page

Data node A "data node" is a physical machine that a database server is running on

DUP Data Update Propagation

HTML HyperText Markup Language

ISP Internet Service Provider

kB Kilobyte

MB Megabyte

ODG Object Dependency Graph

PHP PHP: Hypertext Preprocessor

RDBMS Relational Database Management System

RUBBoS Rice University Bulletin Board System

RUBiS Rice University Bidding System

SQL Structured Query Language

TA Teaching Assistant

Web node A “web node” is a physical machine that a web server is running on

Appendix A

Wikipedia.org: A portion of memcached.txt

```
== Keys used ==

User:
    key: $wgDBname:user:id:$sId
    ex: wikidb:user:id:51
    stores: instance of class User
    set in: User::loadFromSession()
    cleared by: User::saveSettings(), UserTalkUpdate::doUpdate()

Newtalk:
    key: $wgDBname:newtalk:ip:$ip
    ex: wikidb:newtalk:ip:123.45.67.89
    stores: integer, 0 or 1
```

```
    set in: User::loadFromDatabase()
    cleared by: User::saveSettings() # ?
    expiry set to 30 minutes

LinkCache:
    key: $wgDBname:lc:title:$title
    ex: wikidb:lc:title:Wikipedia:Welcome,_Newcomers!
    stores: cur_id of page, or 0 if page does not exist
    set in: LinkCache::addLink()
    cleared by: LinkCache::clearBadLink()
        should be cleared on page deletion and rename
MediaWiki namespace:
    key: $wgDBname:messages
    ex: wikidb:messages
    stores: an array where the keys are DB keys and the values
     are messages
    set in: wfMsg(), Article::editUpdates() both call
     wfLoadAllMessages()
    cleared by: nothing

Watchlist:
    key: $wgDBname:watchlist:id:$userID
    ex: wikidb:watchlist:id:4635
    stores: HTML string
    cleared by: nothing, expiry time $wgWLCacheTimeout (1 hour)
    note: emergency optimisation only
```

```
IP blocks:
    key: $wgDBname:ipblocks
    ex: wikidb:ipblocks
    stores: array of arrays, for the BlockCache class
    cleared by: BlockCache:clear()
```

Appendix B

Run-time Cache Statistics

This section contains various graphs that were obtained by analyzing cache operations during test run-time. More information and discussion is available in Section 5.1.4 and Section 5.2.3. Tests were run with both 50 and 500 clients. The 50 client test was chosen to analyze as it is a relatively small number of clients, and would allow more detail to be seen in a low-load environment. The 500 client test was chosen for analysis as it is a heavier test. It allows observation of the cache in a high-load environment.

B.1 Cache Size and Contents

Figure B.1 and Figure B.3 show the RUBBoS test with both 50 and 500 clients. The figures show the size of the cache, and the number of elements in the cache respectively. Figure B.2 and Figure B.4 show the same information with the RUBiS test suite.

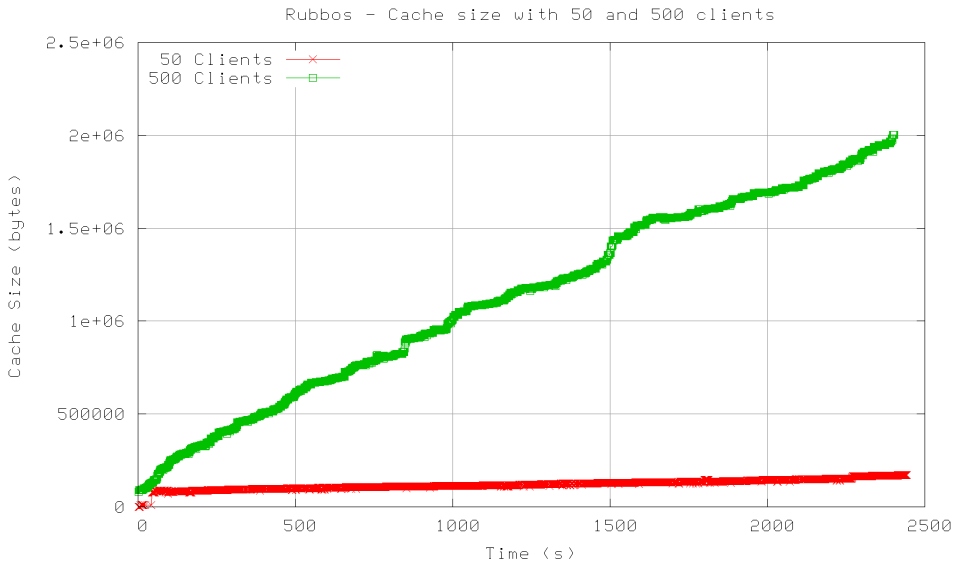


Figure B.1: RUBBoS - cache size vs. test time (50 and 500 clients)

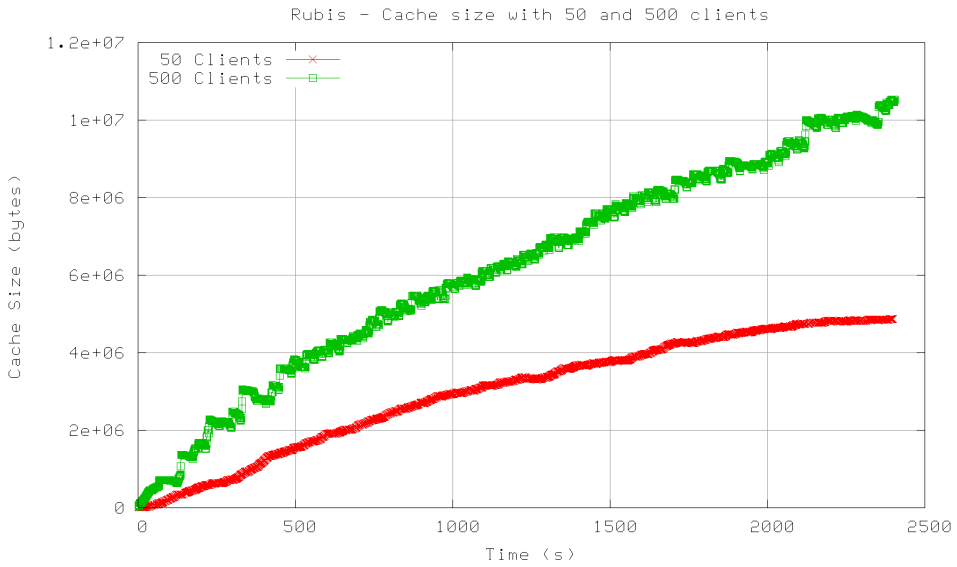


Figure B.2: RUBiS - cache size vs. test time (50 and 500 clients)

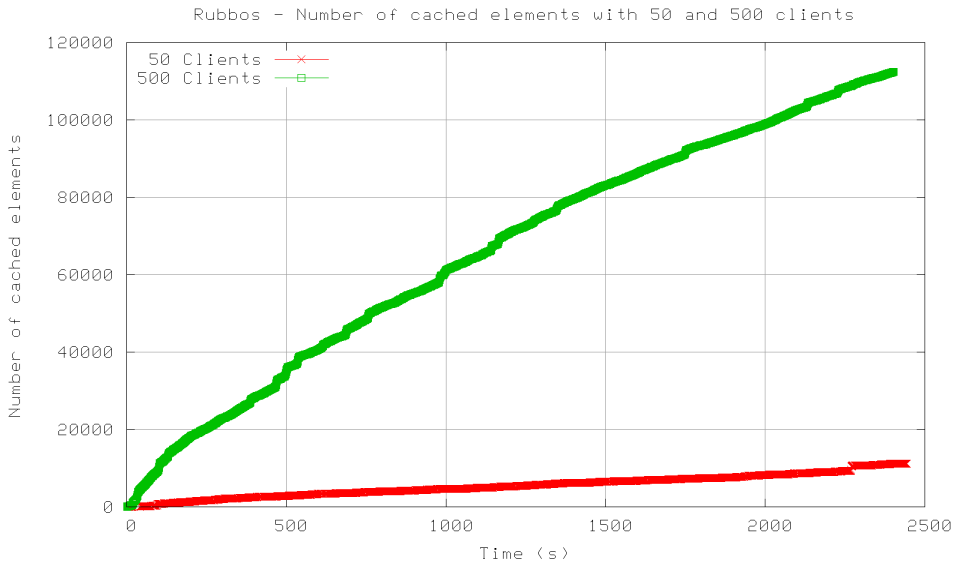


Figure B.3: RUBBoS - number of cached elements vs. test time (50 and 500 clients)

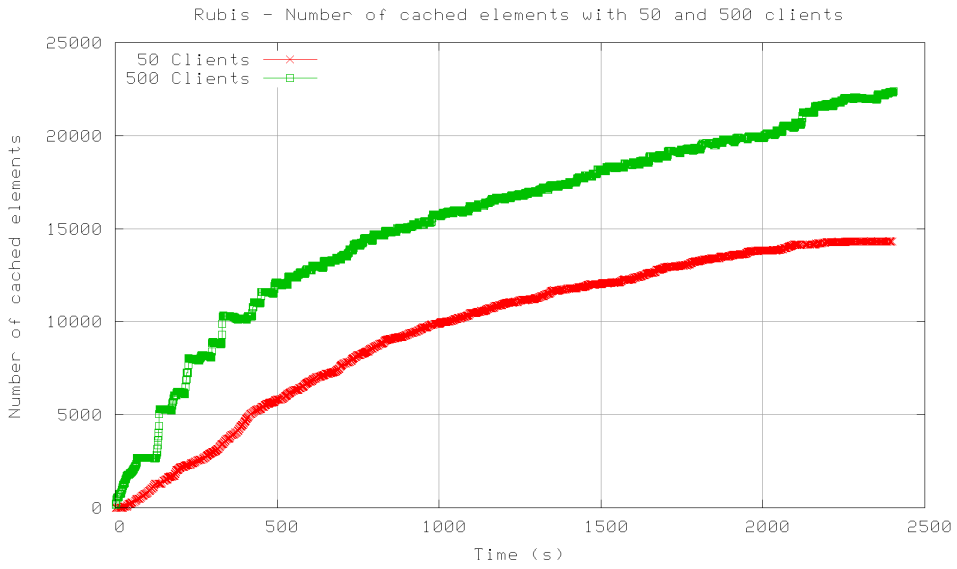


Figure B.4: RUBiS - number of cached elements vs. test time (50 and 500 clients)

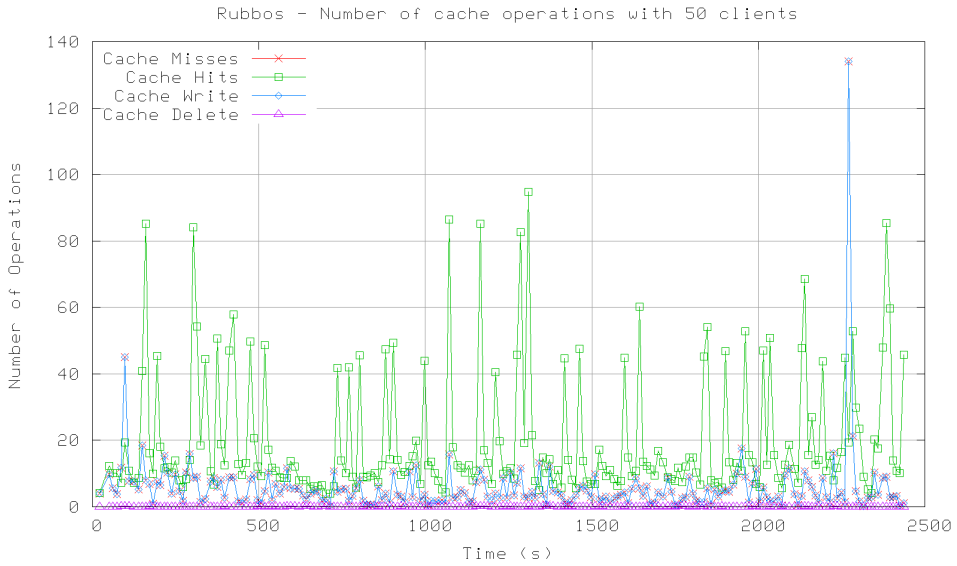


Figure B.5: RUBBoS - cache operations vs. test time (50 clients)

B.2 Cache Operations

This section graphically displays the types of operations that were executed against the cache, and how those types of operations vary with time. Four types of operations were tracked: Cache misses, cache hits, cache writes (where an object is stored in the cache), and cache deletes (where a stored object is invalidated).

These operations are tracked for both 50 and 500 clients, for each of the RUBBoS and RUBiS tests. Figure B.5 displays the results of the RUBBoS test with 50 clients, while Figure B.6 displays the results for the RUBBoS test with 500 clients. Figure B.7 displays the results of the RUBiS test with 50 clients, while Figure B.8 displays the results for the RUBiS test with 500 clients.

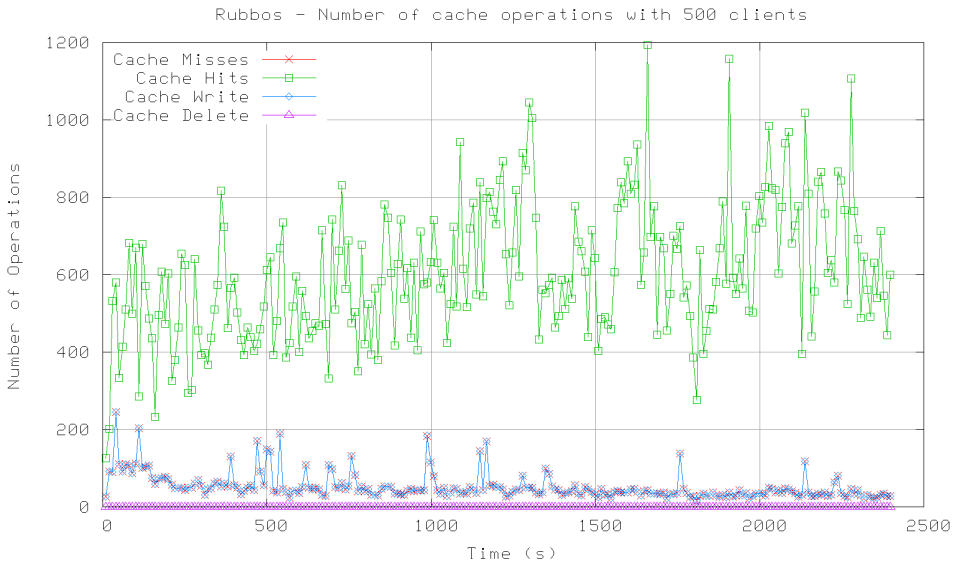


Figure B.6: RUBBoS - cache operations vs. test time (500 clients)

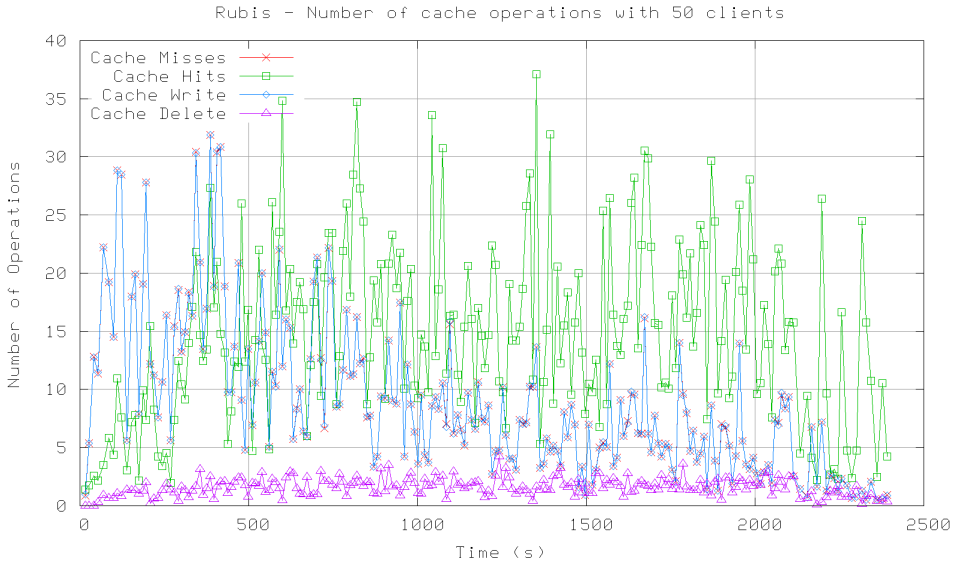


Figure B.7: RUBiS - cache operations vs. test time (50 clients)

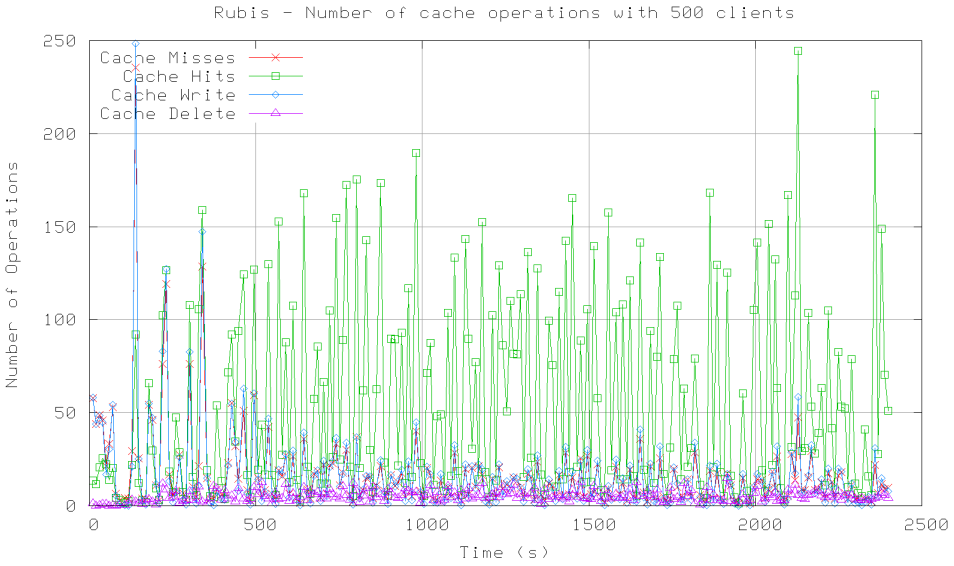


Figure B.8: RUBiS - cache operations vs. test time (500 clients)