Caching Personalized and Database-related Dynamic Web Pages

Yeim-Kuan Chang, Yu-Ren Lin and Yi-Wei Ting
Department of Computer Science and Information Engineering
National Cheng Kung University, Taiwan R.O.C.
{ykchang,p7893113}@mail.ncku.edu.tw

Abstract

In this paper, we use web session objects and database-related dynamic web cache to implement the dynamic web cache system in Tomcat web server. We show how to build the dependency between dynamic web pages and the underlying database fields and session objects. Our experimental results show that Tomcat with proposed dynamic web cache can increase the stability of web server and improves web server throughput by up to 290%

Keywords: dynamic web pages, cache, consistency, Tomcat, Session Objects.

1. Introduction

Today many personalized web pages are created dynamically based on the data stored in the database system. This characteristic requires web servers to generate and serve users the requested page content dynamically. When the web server receives a request for dynamic content, it queries the database to extract the relevant information needed to generate the requested content dynamically. Even when the underlying data retains the same value in the database, web servers query the database and regenerate the same content of dynamic web pages every time a request is received. To improve the web server performance, one solution is to cache the result of requested content. The cached copy may exist in client/browser, proxy, CDN (Content delivery networks), and the web server itself.

There are various ways to accelerate dynamic web pages. Oracle Database Cache [3] is a database-replicated cache directly associated with Oracle9i Database. A copy of the requested data is loaded from the database into the database cache, which is on the middle tier and, therefore, closer to the application for faster accesses. Queries are made locally to the database cache instead of across the network to the original database. It requires heavy database-cache synchronization overhead to maintain the consistency between the data cache and the original database.

Another Oracle Web Cache [4] is a HTTP-level cache, maintained outside the application. It is a content-based cache, capable of caching static or dynamic data with time-based, application-based, or trigger-based invalidation of the cached pages. However, it does not provide a mechanism through which updates in the underlying data can be used to identify which web pages in the cache to be invalidated. The use of triggers is inefficient and may cause a large overhead on the underlying database system.

DUP [1] maintains data dependence information manually between cached objects and the underlying data which affect their values in a graph. In [2], they improved the system to general multitier architectures and add a set of algorithms that have been deployed for high-performance serving of dynamic content to many clients at highly accessed Web sites. The disadvantage is that it maintains the dependence graph manually, such that each time a new web page adds to the web site, the administrator should rebuild the graph.

In this paper, we find all relevant underlying data to dynamic web pages precisely. We explore the database, find all table names and field names. Then check if the MD5 of the requested content changes when the column of one field is changed. If the MD5 changes, it means the database field is relevant to the dynamic web pages. This is a brute-force method, but with some tricks we can make the process faster. In particular, most personalized web site stores data in session objects. We use the data in the session objects to generate dynamic personalized web pages. These kinds of web pages are session related. We also solve the problem of session related dynamic web pages by signing in before accessing the MD5 of the requested content. Mapping session related dynamic web pages to database fields can be done by this method. When we emulate signing action, the data has already loads from the database to the session objects. Later, these session objects can be shared by the subsequent requests.

Our proposed cache system is implemented in Tomcat web server and the dynamic web pages are JSP or servlet pages. Therefore, we name it Tomcat Dynamic Web Pages Caching System, or Tomcat Cache in short. One task of Tomcat Cache is to build the relationship between URIs and database fields is called the URIs to Database Fields Mapping Table. In our experiment, we found the Tomcat Dynamic web Pages Caching System improves Tomcat performance up to 290%.

The rest of this paper is organized as follows. In Section 2, we address the design and implementation of our web caching system. In Section 3, we present the performance results. Last, we give our conclusion in section 4

2. System design and implementation

The design idea of *Tomcat Cache* is to develop a general caching system inside Tomcat, not for any particular web application. The caching system is independent from web applications, such that we don't have any application-related code in Tomcat. The architecture of *Tomcat Cache* is shown in Figure 1. The *Core Caching System* and the *Invalidator* are main components.

At the initial state, *Core Caching System* loads mapping_table.txt, target_files.txt and session_files.txt into *Target URI list*, *Session URI list* and *URIs to Database Fields Mapping Table* separately. When Tomcat serves a request, *Core Caching System* would see if the request URI is in the *Cached URI list*. If yes, it returns the cached web page. Otherwise, it generates the dynamic web page and copy it into the cache.

The *Invalidator* runs periodically to check the update log file of the database called *Binary-Update-Log*. If it finds some fields are changed, it tells *Core Caching System* to remove related URIs from *Cached URI list*.

The main challenge *Tomcat Cache* faces is to build a mapping among the cached web pages and database fields. DUP [1] builds the mapping manually. Our goal is to construct the *URIs to Database Fields Mapping Table* automatically by programs.

First of all, we extract the relevant information from the web access log to know which URIs is accessed in this web server. We exclude all static content (e.g. image files, html files) and extreme dynamic content (e.g. verify.jsp which be used to verify if user name and password are valid). We find the URIs of the

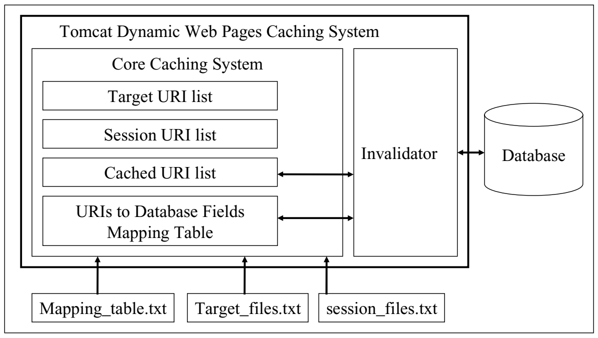

Figure 1: Architecture of Tomcat Dynamic web Pages Caching System.

dynamic web pages that is suitable to cache and save them to target_files.txt. We map them by the following steps:

Start the Tomcat web server and MySQL database server. Backup all database tables.

Get one URI from the target_files.txt.

Get one field from the database fields, e.g. digital_camera.max_price that is a combination of table name and a field name.

Access the URI and record the MD5 of the requested content.

Change all values of this field. If the type of the field is String we set it to null then recover it back to the original value next time. If the type of the field is int or float, we add 10 to it.

Access the URI again and get the MD5 of the requested content. Then compare the new MD5 with the former MD5 and see if they are different. If yes, it means the field we changed is related to the URI.

Go back to step 3, until all database fields are examined.

Go back to Step2, until all URIs are examined.

Restore the database and save the result to mapping_table.txt.

Using the steps described above, we can find the relationship between URIs and database fields. However, some web pages, especially personalized web pages use session objects to save data grabbed from the database and create dynamic web pages by the information in the session objects. These web pages are designed to load personalized data from the database after user's login, then save these data to session objects for future use. This scheme is widely used and largely improves web server performance, because the web server doesn't need to access the database each time it creates a personalized web page but uses data stored in session objects to create personalized web pages. Therefore, web pages depend on the session objects and session objects depend on some database fields. We call the kind of relationship *Indirect Mapping*.

When a user signs in, verify.jsp checks if the user is valid. If the user is valid, verify.jsp loads user related data from the database into the session objects. Then in the same session all requests can use the data in the session objects.

When we want to find the indirect relationship between URIs and database fields, we should emulate our web client logins to the web site then access the URI content. But how does Tomcat maintain session connection? We use *Web Performance Trainer 2.7* to record the process of the web pages access actions. We use W*eb Performance Trainer* only for recording HTTP request header and response header. When *Web Performance Trainer 2.7* starts the benchmarking process, it launches Internet Explorer. Then records each step what Internet Explorer does, including the session connection, the cookies, and the form inputs. After records the web pages you want to benchmark, *web Performance Trainer 2.7* displays each state of request and response. We use this tool to find out Tomcat session scheme.

After we know how Tomcat maintains a session, we can emulate our clients to use sessions and find out the *Indirect Mapping*. We map them by the following steps:

Start the Tomcat web server and MySQL database server. Backup all database tables.

Get one URI from the target_files.txt.

Get one field from the database fields, e.g. digital_camera.max_price that is a combination of the table name and a field name.

Emulate login process by one of the user/password then get the value of JSESSIONID.

Access the URI with the JSESSIONID cookie and record the MD5 of the requested content.

Change all values of this field. If the type of the field is String we set it to null then recover it back to the original value next time. If the type of the field is int or float, we add 10 to it.

Access the URI with the JSESSIONID cookie again and get the MD5 of the requested content. Then compare the new MD5 with the former MD5 and see if they are different. If they do, it means the field we changed is related to the URI.

Go back to step 3, until all database fields are examined.

Go back to step 2, until all URIs are examined.

Restore the database and save the result to mapping_table.txt.

With session detection, we can build the relationship of *Indirect Mapping*.

2.1 Implementation of Core Caching System

We implement the *Core Caching System* inside Tomcat web server. When *Core Caching System* starts

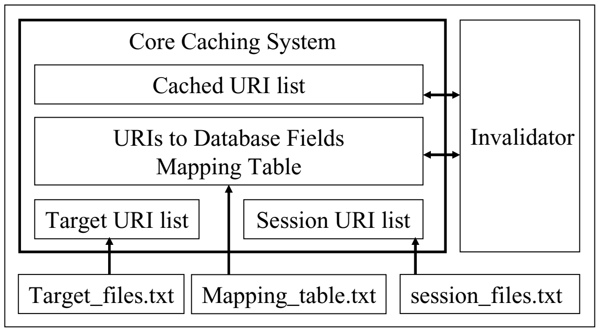

Figure 2: Core caching system architecture.

up, it loads target_files.txt to *Target URI list*, mapping_table.txt to *URIs to Database Fields Mapping Table*, and session_files.txt to *Session URI list*. *Target URI list* is used to check if the requested URI is our target URI. If it isn't, the request breaks the Caching System checking condition, decreases the overhead of Caching System. *Session URI list* is used to check if the request URI is session-related. If it is a session-related request, *Core Caching System* adds the session information to the id of cached object. *Cached URI list* is used to check if the request URI is cached. If the request URI is in the *Cached URI list*, Core Caching System redirects the request URI to the cached object URI. When the invalidator detects some database

fields changed, it notifies *Core Caching System* to remove the associating URI from the *Cached URI list*. *URIs to Database Fields Mapping Table* is static and immutable, and provides the mapping information between URIs and database table fields. When a request is received, the *Core Caching System* checks if the content of the requested page is cached. If it finds a valid cached copy, it redirects the request URI to the URI of the cached content. Therefore, we implement the *Core Caching System* after Tomcat parsed the request URI and before Tomcat maps the request to associating context, wrapper. This location is in the org.apache.coyote.tomcat5.CoyoteAdapter.service() method. In particular, we need to find out where we implement the Core Caching System. When a request is received, the Core Caching System checks if the response content of the request is cached. If it finds a valid cached copy, it redirects the request URI to the URI of the cached content. Therefore, we implement the Core Caching System after Tomcat parsed the request URI and before Tomcat maps the request to associating context, wrapper. This location is in the org.apache.coyote.tomcat5.CoyoteAdapter.service() method.

Now we explain how to use *Binary Update Log* to implement invalidator. The *Binary Update Log* contains all SQL statements that update data. MySQL logs only statements that actually change the data. For example, a delete statement that fails to affect any rows is not logged. Update statements that set column values to their current values are also not logged. MySQL logs updates in execution order. Such that when the *Invalidator* parses the log, it skips all processed data and only parses new ones.

Launching MySQL with the --log-bin argument starts the *Binary Update Log*. If you don't specify the filename, MySQL uses the hostname-bin as the default filename, in our example it's ren-bin. MySQL appends a numeric index to the end of the filename so that the file looks like ren-bin.001. When MySQL server restarted, refreshed or the log reaches the maximum log size, MySQL creates another log file like ren-bin.002 to log. MySQL also creates an index file that contains a list of all used binary log files. By default, the file is named something like ren-bin.index. You may change the file name or path of the index file with the --log-bin-index=file option.

We implement the *Invalidator Thread* as a thread inside Tomcat so that it's easy to communicate with the *Core Cache System*. The thread starts to run when the RenTomcat.getRenTomcatInstance() is called in the first time. The *Invalidator Thread* runs periodically to check if some data changed by the *Binary Update Log*. *Binary Update Log* logs all update SQL statements in execution order. Therefore, the *Invalidator Thread* don't need to parse through the whole log file, instead it starts to parse from the last time it checked. The incremental parsing improves performance a lot. The Invalidator Thread runs periodically, using the period saved in valiable TomcatConstant.POLLING_TIME. When it wakes up, it reads the Binary Update Log from last pared into a string. Then it finds the table name and field name from the update statements by regular expression. Actually, we should have more regular expression patterns to extract table name and field name from all possible update SQL statements. However, it depends on how the programmers write SQL statements code.

3. Performance evaluation

We use one PC and one laptop as the web clients and another PC as the web server. All PCs are connected by the SMC switch. The system and software configuration for each component are as follows:

1. DBMS: MySQL 4.0.15 is used as the database system and it runs on the same machine as Tomcat web server runs, AMD 2500+ with 1G Byte main memory running Windows XP SP2.
2. Switch: SMC SMC2804WBR.
3. JVM: Sun Java(TM) 2 SDK, Standard Edition 1.4.2_03.
4. Server: Tomcat 5.0.24 is the web server.
5. We run it with the parameter `JAVA_OPTS = -server - Xms128m -Xmx384m` which makes it runs faster. `-server` means it runs in server mode, JVM ignores keyboard, mouse events. `-Xms128m` means JVM runs with initial 128M Byte main memory and `-Xmx384m` is the maximum available memory JVM can use.
6. Laptop web client: We run WebBench controller and client on this laptop machine, Intel Pentium 4 1600MHz with 768M Byte main memory.
7. PC web client: We run WebBench client on this PC machine. It communicates with laptop WebBench controller and runs web load at the same time. Intel Pentium 4 3000MHz with 1G Byte main memory.

WebBench [5] is used to emulate many clients that send requests to web servers. It is a closed-loop benchmark tool and such that clients do not send requests faster than the server can respond. One controller is responsible for cooperating these clients to run one mix. After one mix finished, all statistic data are collected to the controller and presented as a Microsoft Excel document.

We set the *Ramp Up*, *Ramp Down*, and *Length* to 5 sec, 5 sec and 30 sec, respectively. It means in each mix we record only 20 seconds. Because our web site is session-related, we check the *Enable Cookie Support* option, such that WebBench treats each mix of a client as single session and they can share the same session objects. In the mix configuration, we configure WebBench to run each mix three times, each mix use two clients, i.e. PC client and laptop client. Engine per clients adds one incrementally until total engine per clients reach 10. Such that we have 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 clients tested.

The Figure 4 presents the result of performance comparison. We can see that Tomcat with proposed web cache system can serve more requests, up to 800 requests per second, while the original Tomcat can only serve 270 requests per second. Thus a 290% throughput improvement over the original web server is obtained. With web cache enabled, Tomcat doesn't need to request database each time a database-related request comes in and doesn't need to regenerate the dynamic web pages which contain the same data. Another thing we found is that *web cache* performs much smoother than the *original*. Because *web cache* serves dynamic web pages request by cached copy, it uses less system resource, like socket connections, process context switch, etc. While the *original* generates database-related dynamic web pages, it queries the database then recomputes the dynamic web pages, spends much more CPU time.

Figure 5 shows the performance comparison in average response time. Note that *web cache* system is still more stable than the *original*. When 20 clients request the *original* web server, the average response time is worse then 90 milliseconds. Figure 6 shows that the *original* curve of response time standard deviation grows much faster when clients increase. We can conclude *web cache* system is more stable than *original*.

4. Conclusion

We present a method to solve the problem of mapping request URIs to the underlying database fields. Furthermore, with session objects detection mapping, dynamic web pages that use numerous session objects to save personalized data can also be detected and cached. This important feature helps web applications that offer personalized service. In addition to the framework design, we implement the system in Tomcat. Our experimental results show that our proposed caching system improves the Tomcat throughput on dynamic web pages by up to 290%.

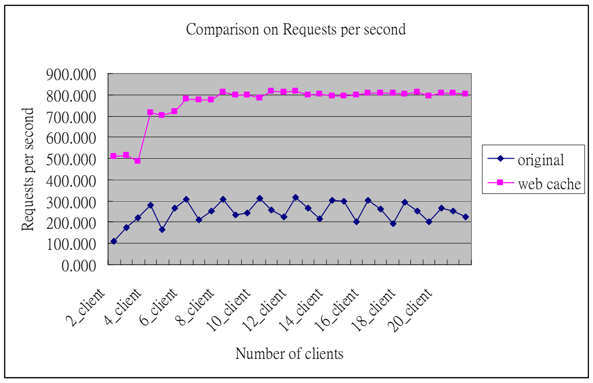

Figure 4: Performance comparison on Requests per second.

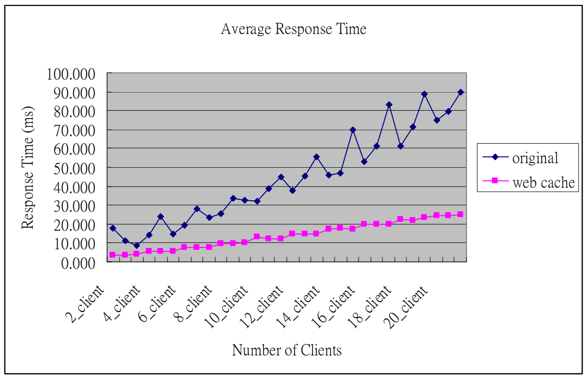

Figure 5: Response Time Comparison.

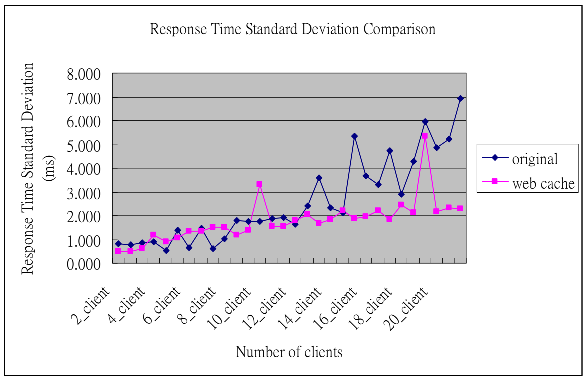

Figure 6: Response Time Standard Deviation Comparison.

5. Reference

[1] Arun Iyengar, Jim Challenger, Paul Dantzig: A Scalable System for Consistently Caching Dynamic web Data. INFOCOM 1999: 294-303

[2] Jim Challenger, Paul Dantzig, Arun Iyengar, Mark S. Squillante, Li Zhang: Efficiently serving dynamic data at highly accessed web sites. IEEE/ACM Trans. Netw. 12(2): 233-246 (2004)

[3] Oracle Database (iCache) Cache FAQ http://www.orafaq.com/faqicach.htm

[4] Oracle Web Cache http://www.oracle.com/technology/products/ias/web_cache/index.html

[5] http://www.veritest.com/benchmarks/webbench/home.as Introduction and related works, 170-177, Chemnitz, Germany, Dec. 2000