Web Caching for Database Applications with Oracle Web Cache

Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, Tie Zhong
Oracle Corporation
500 Redwood Shores, CA 94065
{jesse.anton, lawrence.jacobs, xiang.liu, jordan.parker, zheng.zeng, tie.zhong} @oracle.com

ABSTRACT

We discuss several important issues specific to Web caching for content dynamically generated from database applications. We present the techniques employed by Oracle Web Cache to address these issues. They include: content disambiguation based on information in addition to the URL, transparent session management, partial-page caching for personalization, and broad-scope invalidation with performance assurance heuristics.

Keywords

Caching, dynamic content, disambiguation, session, personalization, partial-page caching, template, fragment, consistency, performance, invalidation, heuristics.

1. INTRODUCTION

Caching has been used to speed up content delivery since the early days of the World Wide Web, a prime example being proxy caches deployed by many organizations. For many years, most of the Web caching was done on static content, such as images and text. Dynamic content in contrast is normally generated from executing business logic in application servers and querying business data in databases. A typical dynamic content Web site configuration is shown in Figure 1.

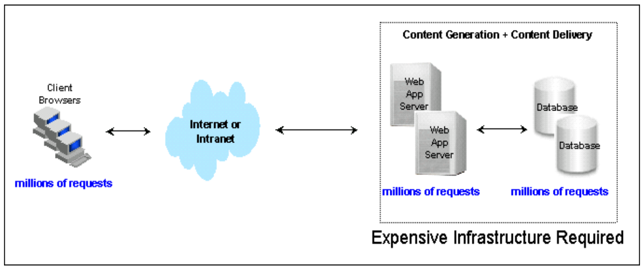

Figure 1. Typical Dynamic Content Web Site Configuration

Only recently has dynamic content caching become available and begun to attract attention. This coincides with the move of Web content becoming more dynamic and personalized with the aim of providing the users a more personal and rich Web experience. Generating personalized content is computationally expensive. This has driven the need for dynamic content caching that can achieve higher cache hit ratios and thus improve performance and scalability in today's Web servers. The need for this scalability is a result of the underlying framework of Java engines and databases having been designed for tens to hundreds of concurrent users rather than the hundreds of thousands of users the Web has introduced.

From Web servers and databases to browsers, caching is done at almost every stage of a request's lifetime (Figure 2).

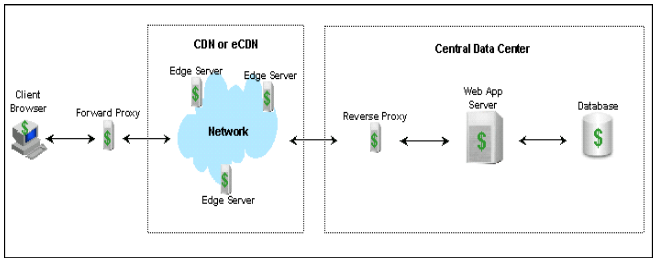

Figure 2. Caching on the Web

In the course of request, database caches can reduce database query cost. Caches within application servers can eliminate database queries and reduce application execution cost. Reverse proxy caches can eliminate database queries and all application execution cost. CDN (content delivery network) or eCDN (enterprise content delivery network) caches not only eliminate database queries and application execution cost, but also reduce network transmission cost. Proxy caches and browser caches offer further server and network cost reduction, but their cache content is controlled by end users and tends to have a low cache hit ratio for dynamic content.

Oracle Web Cache is one of the first reverse proxy caches designed for dynamic content. Web Cache is also used within CDNs/eCDNs and may also be used for caching application data such as SOAP responses. Many customers have successfully deployed it to address their performance issues with dynamic content.

Compared to static Web content caching, dynamic content caching has many unique issues. This paper focuses on how Oracle Web Cache addresses disambiguation, session management, personalization, and consistency.

Disambiguating content is essential for properly storing and delivering documents and maintaining high cache hit rates. URLs, normally the unique identifiers for static content caches, are often insufficient for dynamic content due to request-specific content generated by applications. For example, a browser-aware application may generate different responses for different browsers for the same URL.

Web applications often need to maintain session state (such as shopping carts) for every user. Some applications embed the session state within the URL and hyperlinks as an alternative to cookies. A naïve cache would interrupt the session establishment during cache hits and prevent new session creation. Additionally, pages with embedded session state become unique for each session and defeat simple caches.

Personalized pages are often unique aggregations of common content. A single page may combine content fragments with different caching properties. Some are cacheable, some are not, some expire soon, and some hardly expire. Some content expires at prescribed intervals, while other content expires as the result of an external event. The combination also creates "cache space explosion" for traditional full-page caches due to the redundant caching of common fragments. There is little value in caching a personalized page as a whole page since the page can only be delivered to the user or users for whom it was designed, and only while all the content is valid. In addition, some personalized pages contain personalized information such as personal greetings or addresses, further limiting the hit rate for the page.

Compared to static content, dynamic content may change unpredictably upon certain events, such as changes in the underlying data. The dependencies between cached content and the underlying data are complex and difficult to maintain efficiently. Previous works ([1], [2], [3], [5]) have focused on calculating, monitoring and maintaining such dependencies in caches. However, the cost of analyzing ad hoc queries, triggering on data updates, and integrating with third-party databases and applications can often be prohibitively expensive. Therefore, instead of making fine-grained but expensive data mappings, we choose not to maintain data mappings at all, but to support coarse, "dumb", and conservative invalidation that is cheap and optimized in both invalidation processing and cache update penalty. This has proved the most scalable and convenient approach in most of our commercial deployments.

2. DISAMBIGUATION

One of the challenges with dynamic content caching is identifying the correct document to deliver to a particular request. Each cached document requires a cache-wide unique key. If the key matches multiple versions of a document, the cache will deliver the wrong content. If multiple keys match the same document, the effectiveness of the cache is diluted with unnecessary cache misses and cache storage is wasted on duplicate content.

The primary component of the cache key is simply the URL corresponding to the document. However, in many cases the URL is either not sufficient or contains too much information to be the key. Applications are capable of generating distinct HTML content in response to requests with distinct header content. HTTP headers that often affect content include application-provided cookies and user agents (browser model and version). All the HTML documents that can be generated from such applications share the same URL, yet they need to be stored as distinct objects and delivered to the appropriate recipients.

Some applications embed session-key or user-tracking data in URLs as an alternative to cookies. This technique requires the data be embedded in hyperlinks in the pages delivered, enabling the application to identify the session or user in each interaction. The result of this system is URLs with superfluous data for generating a cache key.

With the above in mind, Oracle Web Cache composes the cache document key by augmenting the URL with HTTP request header data or stripping data from the URL as appropriate. When the URL is augmented with header data, the Web Cache stores and delivers different versions of the page in response to the same URL.

One may instruct the Web Cache to use the existence of the header data or the value of a portion of the header as the disambiguating factor. With the existence option, the cache will store and deliver one version of the page when the request contains the header, and one version of the page when the request does not contain the header. With the value option, the cache will store and deliver a distinct version of the page for each header value (including the absence of the header.)

The disambiguation factors can be statically configured for the cache or dynamically specified in the response. With configuration, Web Cache can be used with unmodified applications. With the response specification, application content is self-described for Web caching and obviates the need for deployment-time cache configuration.

3. SESSION STATE MANAGMENT

Many Web applications utilize sessions to track users' state and collect user data. A session between an HTTP client and the backend HTTP server typically involves some state information of the client being stored on the server. The server sends the response to a client's request based on its current state. Information about the state is sent between the client and the server using either cookies or URL encoded parameters, usually in the form of a unique session identifier.

We designed the Web Cache to support session management. When required, requests lacking specified session keys pass through the Cache to allow the application and application server to perform unfettered session state establishment. The response from the application will now contain the session key in either the HTTP set cookie header or embedded within the hyperlinks of the page. Subsequent requests contain the session key and may be satisfied by the Web Cache.

If the application embeds the session key in URLs and hyperlinks, each page becomes unique for each session. This leads to a cache miss for every new session view of every page and a cache populated with session-specific pages.

Oracle Web Cache can cache and deliver pages with session-embedded data. The Cache stores the pages with placeholders replacing the session-specific strings. When the cache receives requests with a session-embedded URL, it composes the cache key by removing the session string from the URL. The cache will then substitute the incoming session information for the placeholders in a cached response, and then deliver the page. This preserves session state across multiple cache hits.

The Web Cache also employs this same string substitution technique for some personalization, such as shopping cart contents displayed on otherwise static catalog pages. We present this in Section 4.2.

Another example of Web Cache session support is a situation where pages delivered for established sessions are not sharable and not feasible to cache. However, content for "anonymous users" without sessions is very much sharable, and a session is not required to access such content. In this case, the cache does the opposite of the above. If it detects the existence of the session in a request, it passes it to the back end server for the non-cacheable content; and if it does not find the session, it can then satisfy the request with a cached, sharable, common page for the anonymous users.

These are the primary examples of how Web Cache handles session management. There are other variants on above scenario. The objective is to perform the correct actions based on the different session requirements/settings and to cache as much shareable data as possible.

4. PERSONALIZATION

Personalization is common in dynamic content. There are at least three common challenges when caching personalized content. First, many personalized pages cannot be cached for long or at all. Personalization often creates pages that each consists of "fragments" with different caching properties (volatility, cacheability, etc.). For example, a Portal page may include stock quotes that expire in 20 minutes, news that expires in 3 hours, and rotating ad banners that should not be cached. To serve consistent content, traditional caches need to update the entire page at the highest change frequency of all its fragments. Second, the customizable combination of fragments creates a vast number of unique pages. Cache hit ratios will be low even if these unique pages are all cacheable. Third, personalized information often appears in Web pages, making them unique for each user.

4.1 PARTIAL-PAGE CACHING

To solve the first two challenges, Oracle Web Cache operates in a partial-page model, in which each Web page can be divided into a template and multiple fragments that can in turn be further divided into templates and lower level fragments. Each fragment or template is stored and managed independently; a full page is assembled from the underlying fragments upon request. Fragments can be shared among different templates, so that common fragments are not duplicated to waste precious cache space. Sharing can also greatly reduce the number of updates required when fragments expire. Depending on the application, updating a fragment can be cheaper than updating a full page. In addition, each template or fragment may have its own unique caching policies such as expiration, validation, and invalidation, so that each fragment in a full Web page can be cached as long as possible, even when some fragments are not cached or are cached for a much shorter period of time.

Oracle Web Cache uses Edge-Side Includes, or ESI ([6]), with our unique extensions ([8]) to achieve flexible partial-page caching. ESI is a simple markup language for partial-page caching. Applications can mark up HTTP responses with two different kinds of tags to define the fragment-template structure in the response:

- The "include" tag: An include tag (<esi:include>) is a reference to an independently obtained fragment. Web Cache will insert the fragment, whose URL is specified in the "src" attribute of the tag, when the cache assembles the full response. A fragment itself may contain other ESI tags, including the include tags.

- The "inline" tag: This tag is intended for applications that cannot support separate HTTP requests for each fragment. The inline tag (<esi:inline>) demarcates a fragment embedded in an HTTP response. The embedded fragment does not need to be fetched or assembled but will be cached separately from the template enabling the fragment to be shared with other pages, and reduce the fragment's update frequency.

Note that ESI processing may be performed in numerous locations. ESI processing is typically performed in a reverse-proxy cache or a CDN cache. It may also be performed in an application cache, a proxy cache, or even a browser cache. Templates and fragments can be retrieved from the cache itself, from an origin Web server, or from a downstream cache (Figure 3). The source and cacheability of fragments and templates are orthogonal to their assembly, so that cacheable and non-cacheable fragments can co-exist in any template.

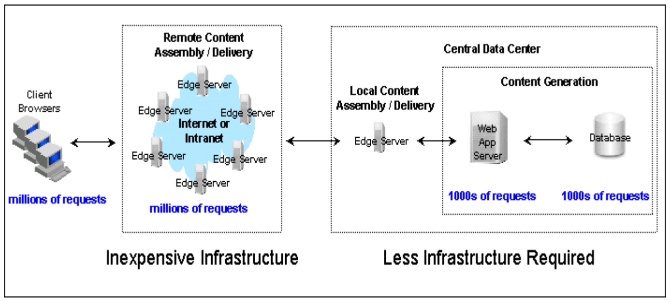

Figure 3. ESI separates content delivery from content generation for greater scalability and cost savings

4.2 REQUEST AND RESPONSE VARIABLES

Another problem in caching personalized pages is the user-specific information appearing in these pages, making them unique at the full-page level. Although partial-page caching can reduce the uniqueness of each page by sharing as many fragments as possible, some small but personal strings in a page are still unique to every user. For example, many Web pages contain tens or hundreds of hyperlinks embedding application session IDs.

In order to avoid storing and retrieving too many small fragments, we allow application developers to use variables in an ESI template. Because variables can be resolved to different pieces of request information or response information, the uniqueness of templates and fragments can be significantly reduced when personal information abounds.

There are two kinds of ESI variables: *request variables* and *response variables*. When an ESI template is assembled, a request variable is instantiated to a piece of request information such as a query string parameter, a cookie, or an HTTP header. For example, when a request for a dynamic page carries an application session ID in a query string parameter, this page may contain many hyperlinks with ESI request variables accessing this session ID, so that generated hyperlinks can carry the session ID into the next clicked page.

A response variable is similar to a request variable, except that its value comes not from the request, but from a special fragment called ESI *environment*. An ESI environment is essentially a special type of fragment whose response defines a set of variables that can be accessed by response variable occurrences in the enclosing template. For example, a dynamic page with a calendar may need to present personal appointments that cannot be stored in browser cookies due to cookie size limits. The application can instead reference a "profile" environment fragment in the template, and refer to all appointments in the environment without making separate requests and cache objects for each appointment. In addition, an environment may be used to merge multiple small fragments into one environment by which each fragment can be referenced through response variable instantiation. This reduces storage and retrieval overhead similarly.

4.3 PERSONALIZATION IN ALTERNATIVE CACHES

ESI enables Oracle Web Cache to perform in-cache personalization as a reverse-proxy cache or a CDN/eCDN cache. Similar partial-page caching ideas have been explored in alternative cache locations.

[2] and [3] describe an integrated Web publishing system that uses a similar partial-page caching model. The primary difference between their partial-page model and ours is that theirs requires tight and proprietary application integration to define the template-fragment relationship.

Chutney and BEA ([4]) built partial-page caches within application servers that save partial application execution in cache hits. The template-fragment definition is also through tags. However, these caches depend on application integration on different platforms such as servlet, JSP and ASP. Oracle Web Cache communicates with any application through HTTP. The performance gains of application caches also suffer more because the cost to invoke expensive applications that need performance improvements can never be entirely eliminated, while Oracle Web Cache handles cache hits without contacting an origin Web server or invoking a JVM. The independence from more expensive application execution environments such as a JVM allows us to develop a much more efficient and dedicated system.

4.4 PERFORMANCE IMPROVEMENT

We will use a simple example to illustrate the performance impact of caching and assembling ESI templates and fragments. This example also illustrates another component of ESI, ESI for Java (JESI). JESI is a JSP tag library with two roles. First, JESI generates the appropriate ESI tags and headers in the JSP output that instruct ESI processors to cache (or not) templates and fragments for the appropriate duration. Second, JESI facilitates the partial execution of JSPs when an ESI processor requests fragments and templates.

This application provides the user with a personalized page of five stock quotes, three news topics, three sports scores, and the weather forecast for one city. In Table 1, we present the quantity and volatility of data we use in this example.

This application is a JSP and collection of EJBs. The JSP performs all the HTML formatting while the EJBs provide a simple access method for the data. The JSP would begin processing by calling an EJB that provides the identity of the user's selection of five stocks, a city, three news topics, and the sports teams. This EJB is essentially generating the template. The JSP then calls a series of EJBs that provide the current data for the JSP to present. Each EJB may obtain the source data from an external stream, a database, or some other source, and the EJB would likely cache the data in memory for optimal performance.

Table 1. Example Application Content Assumptions

Content	Number of Objects	Time to Live
Stock Quotes	10,000 Securities	15 minutes
Weather	1,000 Cities	One Hour
Sports	500 Teams	One Hour
News	50 Topics	One Hour

The performance data assumes the application and Web Cache run on a 933 MHZ Intel Pentium III processor running LINUX, with enough memory to cache all the content. Without Web Cache we assume the system has a capacity to generate 50 pages per second. On such a system we have measured Web Cache assembling and delivering 2140 pages per second of similar content with a dozen ESI fragments. In Table 2, we present the performance comparison of the application with and without a populated Web Cache delivering a uniformly distributed million pages per hour. The first row presents execution data for the basic application. The second row presents data for the same application where the JSP has been tagged with JESI tags and the application is run with Web Cache.

Table 2. Web Cache and Application Execution Cost

	Web Cache	Application
Without Web Cache		1,000,000JSP 1,000,000Template EJB 5,000,000Stock EJB 1,000,000Weather EJB 3,000,000Sports EJB 3,000,000News EJB *Requires 20,000 cpu seconds*
With Web Cache	1,000,000 Pages *Requires 468 cpu seconds*	41,550JSP w/JESI 0Template EJB 40,000Stock EJB 1,000Weather EJB 500Sports EJB 50News EJB *Requires <831 cpu seconds*

With ESI processing, Web Cache delivers the bulk of the replies without invoking the application. Web Cache only invokes the application when it processes a request for an expired fragment. In this example we present the worst-case scenario for caching, where all 10,000 stock quote fragments are accessed throughout the hour, requiring all the fragments to be replaced four times, thus generating 40,000 JSP and EJBs invocations. The same applies to all the other content. We assume each EJB invocation requires a separate JSP request, so the total number of JSP requests represents the sum of all EJB invocations: 41,550.

In this example Web Cache provides a performance gain greater than a factor of 15. The application without Web Cache consumes approximately five and one-half hours of CPU time in the one-hour period. The application with JESI and Web Cache needs less than 22 minutes of CPU time in the same period.

5. CONSISTENCY MANAGEMENT

On the Web, dynamic content is usually more volatile than static content, changing more frequently and less predictably. Since dynamic content is often based on underlying data stored in databases, events external to the Web application can trigger changes in the Web content. While we support expiration and validation techniques to maintain consistency, invalidation is our key technology in consistency management due to its ease of control by applications.

The key challenge in invalidating dynamic Web content is to establish mappings between underlying relational data and the cached Web data. With such mappings, changes in the underlying data can be translated to an invalidation of the affected cache content. Previous works ([1], [2], [3], [5]) aim at maintaining accurate mappings and calculating affected cached objects to invalidate from such mappings ([1], [2]). The mappings are maintained by monitoring traffic between databases and applications and between applications and Web listeners, by explicit API calls by applications ([2]), or by run-time and compile-time query analysis ([5]). However, in large Web sites, the cost of monitoring, triggering, and analysis can often be very expensive. Although the goal is to invalidate less if possible so that performance does not suffer, the dilemma is that maintaining accurate mappings automatically can not only consume more resources, but it also can lead to more frequent but narrower-scoped invalidations, which may be much more expensive to process than fewer, wider-scoped invalidations. Application integration of such automatic maintenance is also complicated in third-party applications and databases.

An alternative to fine-grained (e.g., a single cached object by its exact URL) invalidation is coarse, broad-scope, conservative invalidation (e.g., all cached objects matching a broad URL prefix). While each ensures full consistency, the two methods involve a trade-off between the cost and ease of generation, and "*cache choking*" – the sudden cache hit ratio decrease after invalidation. Fine-grained invalidations are hard and expensive to generate with less cache choking, while coarse invalidations are easy and cheap to generate with more cache choking.

However, if we control not only the granularity but also the consistency of invalidations, we can have coarse invalidations with loose consistency, which then allow us to achieve both minor cache choking and easy-to-use invalidations. In Oracle Web Cache, a user can request that invalidated objects be refreshed within a grace period, during which the invalidated stale content may still be served as cache hits. Because we measure the popularity (from their access patterns) and the validity (from how soon they will have to be refreshed) of all cached objects, our Performance Assurance Heuristics prioritize all updates and only select a small set of the most popular and least valid objects to refresh first. Since most Web content access patterns follow the Zipf 80-20 distribution, the percentage of fresh responses that the Web Cache delivers increases quickly. There is very little cache choking because stale content is being served from the cache during the grace period. The loss of consistency is minor and controlled within the grace period.

These different invalidation options have proved to be convenient and efficient in many applications. Since deploying Oracle Web Cache, Oracle's online store has used coarse, high-consistency invalidations and coarse, loose-consistency invalidations on different occasions, and achieved dramatic improvements in performance and ease of use.

6. SUMMARY

Oracle Web Cache is a reverse-proxy, partial-page Web cache specializing in dynamic content caching. The Cache utilizes flexible document identification, intelligent session management, convenient and flexible invalidation messages, and fast page assembly to drastically reduce application execution and database query cost.

7. CONCLUSION

Dynamic content caches are providing performance boosts to traditional data systems as well as enabling new distributed computing paradigms to scale to levels once exclusive to esoteric transaction processing systems.

The marriage of dynamic content caching with traditional databases provides the manageability of the database with the performance of a fast cache. Referential integrity is no longer relegated to the raw data of financial applications as more data may be managed in databases without a performance or scale penalty.

HTTP has become the standard for computing communication, for client-server application communication, for enterprise application integration, and for enterprise-to-enterprise communication with SOAP. Dynamic content caching of HTTP traffic has the potential for broad impact on the performance and scalability of many systems beyond the traditional Web sites where Oracle Web Cache started. Systems built without similar technology will scale poorly and cost an order of magnitude more to run and develop. Once dynamic content caches become tightly integrated with development environments, security infrastructure, and network infrastructure, most applications will benefit from the increased performance. Standards such as ESI are essential for cross-industry interoperability.

8. ACKNOWLEDGMENTS

We would like to thank all members of the Oracle Web Cache team, past and present, for their contribution to the ideas covered here, and for successfully implementing all the techniques in this product. Our thanks to Mark Tsimelzon and the rest of the Akamai ESI team, for their contributions to ESI. We would like to express our special thanks to Roger Bamford, Shehzaad Nakhoda, Rajiv Mishra, and Sachin Desai for their original contributions to the techniques discussed in this paper.

In Memoriam: We would also like to acknowledge the contributions of Danny Lewin, founder of Akamai Technologies

and one of the pioneers of ESI. Danny perished tragically in the attacks of September 11, 2001, but his inspiration lives on.

9. REFERENCES

[1] K. Candan, W. Li, Q. Luo, W. Hsiung, D. Agrawal. *Enabling Dynamic Content Caching for Database-Driven Web Sites.* In Proceedings of ACM SIGMOD 2001, Santa Barbara, California.

[2] J. Challenger, A. Iyengar, P. Dantzig. *A Scalable System for Consistently Caching Dynamic Web Data*. In Proceedings of IEEE INFOCOM'99, New York, New York, March 1999.

[3] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, Paul Reed. *A Publishing System for Efficiently Creating Dynamic Web Content*. In Proceedings of IEEE INFOCOM 2000, Tel Aviv, Israel, March 2000.

[4] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, K. Ramamritham, D. Fishman. *A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration*. In Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[5] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. *A Middleware System Which Intelligently Caches Query Results*. In Proceedings of ACM/IFIP Middleware 2000, Palisades, New York, April 2000.

[6] *Edge-Side Includes*, http://www.esi.org.

[7] Inktomi Traffic Edge, http://www.inktomi.com.

[8] *Oracle9iAS Web Cache Administration and Deployment Guide, Release 2.0.0*. http://otn.oracle.com/docs/products/ias/content.html

[9] Server Accelerator, http://www.cacheflow.com.