Web Caching With Dynamic Content

(only first 5 pages included for abstract submission)

George Copeland - copeland@austin.ibm.com - (512) 838-0267
Matt McClain - mmcclain@austin.ibm.com - (512) 838-3675
IBM

Abstract

The web application server (WAS) is becoming the transaction monitor of the internet world. A WAS is a web server extended with support for application development like a transaction monitor. The problems faced are similar, including servicing a large number of clients, handling high throughput with reasonable response times, as well as the need for security, correctness and high availability. The main difference is the type of client, which is more numerous but less intensive and less controllable. The client uses HTTP for communication and HTML for presentation.

This paper describes the issues involved in caching dynamic content in a web application. It describes when caching is a good idea, caching data vs. rendered pages, caching granularity, where to put the cache in a system, what metadata is needed for caching and how a web application developer might specify it. It also discusses implementation issues, including throughput scaleup issues.

Many of the ideas in this paper were taken from the pioneering work of [Iyengar and Challenger 1997], [Challenger, Iyengar, and Dantzig 1998], and [Challenger, Dantzig, and Iyengar 1998] on the Olympic web sites, as well as experience with Net.Commerce.

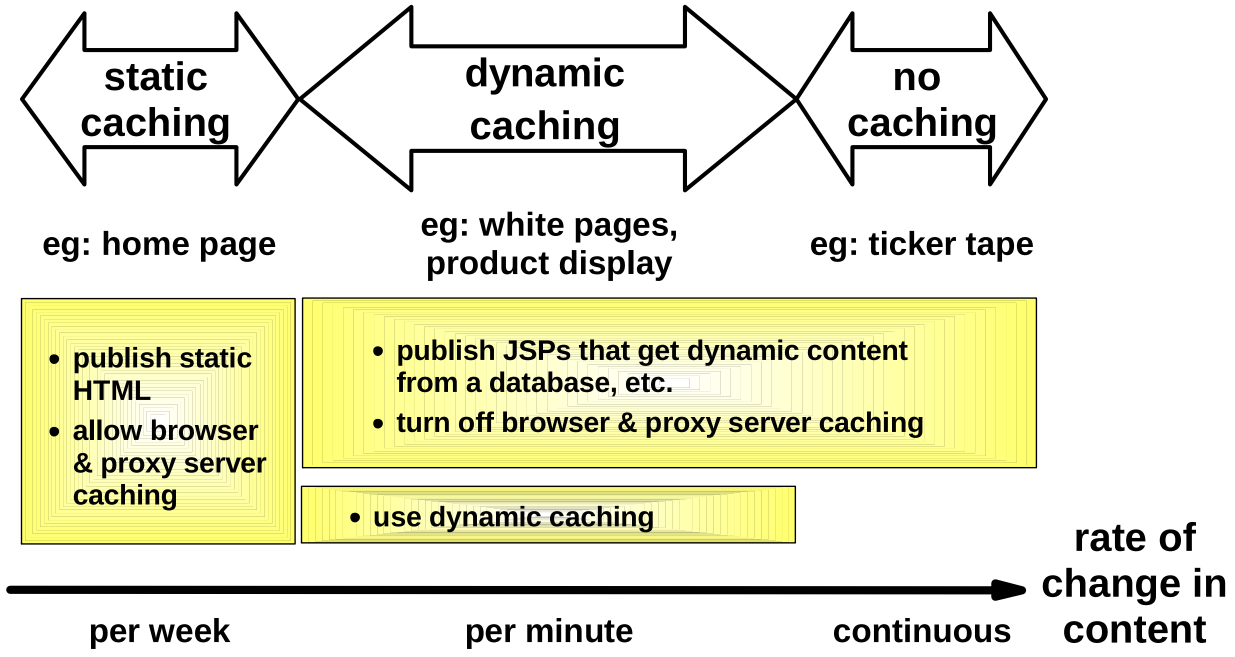
When caching is a good idea

No content is completely static because everything changes eventually. Figure 1 illustrates how content in an HTML page should be treated differently depending on how often it changes:

- If content changes very infrequently (eg, a typical home page), then it is convenient to republish the web site whenever its content changes. It is usually safe to enable browser and proxy server caching.
- If content changes so often that it is unreasonable to republish the web site every time it changes, then JSPs should be used that dynamically get the content from a file or database, and then render (ie, format) it into HTML. In this case, static caching in browsers and proxy servers should be disabled. In this case, dynamic caching may or may not be useful:
    - ✓ If the content is constant over a large number of requests (eg, products in e-commerce, white pages), then performance can be significantly improved by using dynamic caching. With dynamic caching, either time limits or an event-driven invalidation mechanism can be used to keep the content in the cache up to date.
    - ✓ If the content changes continuously (eg, a ticker tape), then any form of caching is a bad idea. JSPs should be used without any caching.

One way to view caching is that it automates the publishing process. For updates that are too frequent for manual republishing each time some underlying dynamic content changes, JSP/servlets with dynamic content can be used along with caching to incrementally and automatically republish.

Figure 1: When To Cache



static caching
dynamic caching
no caching
eg: home page
eg: white pages, product display
eg: ticker tape
• publish static HTML
• allow browser & proxy server caching
• publish JSPs that get dynamic content from a database, etc.
• turn off browser & proxy server caching
• use dynamic caching
rate of change in content
per week
per minute
continuous

Caching HTML vs. data

Caching rendered HTML offers the following performance improvements when the underlying dynamic content has not changed:

- Avoiding access to backend servers (eg, database, transaction monitor, internal application, news service) to get the dynamic content.
- Avoiding the rendering the dynamic content into HTML.

Caching the underlying dynamic content (ie, data) instead of HTML requires rendering the data into HTML during the fast path. However, the fast path occurs more often. When the same data is rendered in multiple ways, caching rendered HTML requires accessing the backend server once for each rendering. The tradeoff is difficult to make because it is between always avoiding a typically less expensive activity (ie, rerendering) vs. infrequently avoiding a typically more expensive activity (ie, backend access). In this paper, we concentrate on caching rendered HTML.

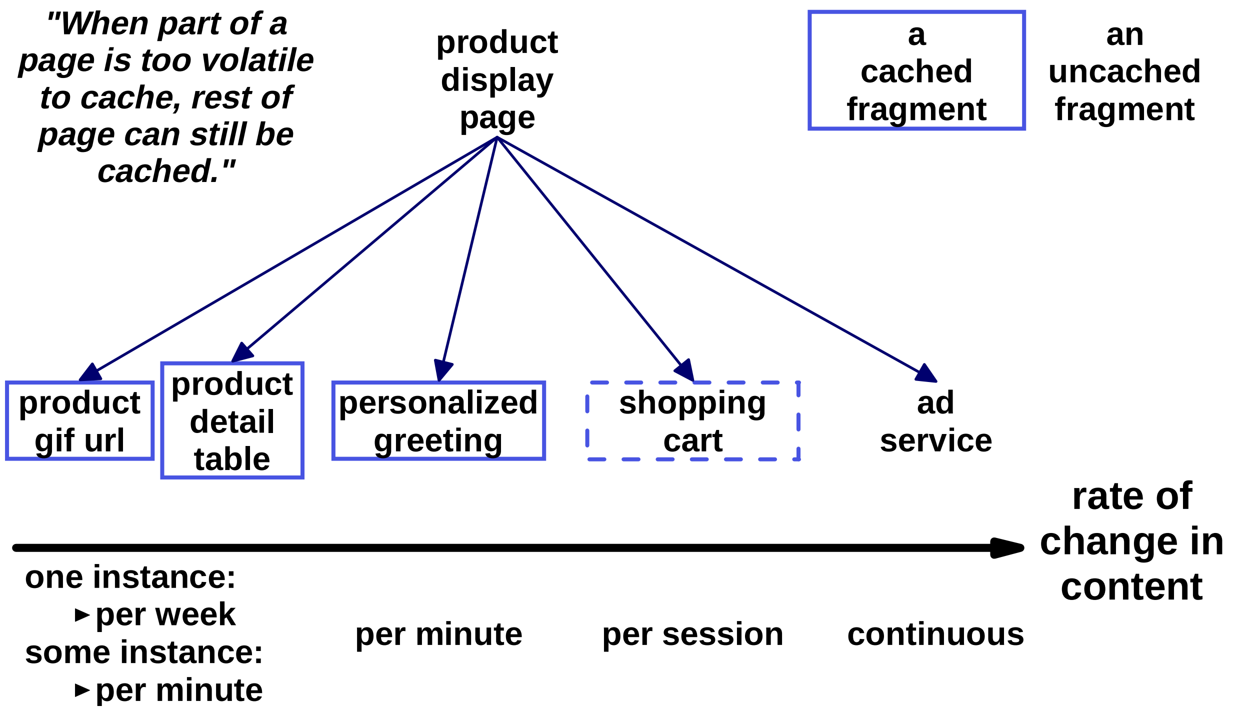
Caching granularity

Caching rendered HTML with dynamic content requires flexibility in the granularity of the cache. A *fragment* is a part or all of a rendered HTML page which can be cached. A fragment can contain 0 or more child fragments, and can be contained by 0 or more parent fragments, forming a directed acyclic graph (DAG). Figure 2 illustrates a returned page fragment (the product display page) made up of the following 5 child fragments in order of increasing rate of change in the underlying data:

- The href for an image that shows what the product looks like. The underlying database record for the product contains this URL.
- A formatted table that includes the detailed description of the product (eg, product order number, name, options and price).
- A personalized greeting (eg, “Hello, John! Welcome to AcmeCorp.”).
- A formatted shopping cart, including the order number, name, quantity and price of the products that have been chosen for possible purchase.
- A href for an image that displays an advertisement.

The advertisement href is different each time a page is sent to a shopper. This makes the overall page too volatile to cache. However, fragment granularity still allows the rest of the page to be cached. The href to the product image and the detailed product description are excellent candidates for fragments to be cached, because the underlying data describing a particular product changes infrequently. However, the underlying data describing some product changes far too frequently for static publishing. The personalized greeting has the lifetime of a user session but only for a particular shopper. It may be used several times within a fairly short time interval, so it is a good candidate for dynamic caching. The shopping cart changes multiple times within a user session (every time something is added or the quantity changes), so it is not as good a candidate for dynamic caching as the personalized greeting. However, if it is included on every page returned to the shopper, then it is typically returned several times between

Figure 2: Fragment Granularity



changes, so there is a reasonable case for caching it. The advertisement href is a very poor candidate for caching because the hit ratio would be zero and caching has its own overhead (ie, storing it in the cache and invalidating it).

Where to put caching

Two places to put the caching in a system are the following:

- **External cache**: There are a wide variety of simple caches that are external to the web application server (eg, web server, sprayer).
- **Integrated cache**: Integrate the cache into a web application server. When a template page containing requests for dynamic content is executed, cache the resulting rendered fragment.

The advantage of an external cache is:

- These offer better cost-performance than a web application server which supports the flexibility and protection required for application code. A simple cache can even be supported in machines that do not have the overhead of an operating system (eg, multiple processes/threads, memory management). These systems can have a cost-performance advantage of as much as 5x.

The advantages of an integrated cache are:

- Fragment granularity can be exploited, so that if part of a page is too volatile to cache, the rest can still be cached. With an external cache, only whole pages (ie, top-level fragments) can be cached, so that the pages cannot have anything on them that is too volatile to cache.
- Access control can be enforced, so a page can be accessible to only a selected group (eg, the product description pages may be different for different shopper groups). With an external cache, authentication is not done except for firewall enforcement, so the pages must be accessible to everyone within that security domain.

The best of both worlds is a web application server with an integrated cache that can also push results of selected template pages to a external caches. This would allow a decision at a template page granularity to be made as to whether that template would be pushed to a simple cache vs. an integrated cache. Those templates that satisfy the above conditions could exploit the cost-performance advantage of an external cache. Other templates could use the flexibility of the integrated cache.

Options when caching a fragment

For each fragment that is cached, the FragmentCache supports the following options:

- Allow LRU replacement to be applied to it or not (ie, pinned). Reasons for this might be realtime requirements or the fact that the web developer knows better than an LRU algorithm.
- Set invalidation ids that represent external events which cause the fragment to be invalidated. When a fragment is invalidated, all of its parent fragments are invalided. A database trigger might be used to create the external event that initiates the invalidation.
- Set a time limit in second granularity. When the time limit expires, the fragment is discarded from the cache. For the web developer, this is simpler than setting invalidation ids because there are no triggers to write. However, it causes many fragments to be discarded and rerendered when they are actually still valid.
- Aggressively rerender a fragment when it is invalidated or when its time limit expires, instead of waiting for an external request that needs the fragment. Aggressive rerendering can exploit the idle time of the server by rendering pages in a background mode. A daemon

thread would look for rerendering work at low priority. When it finds work, it executes the rerendering at normal priority, so that external requests that need the fragment are not kept waiting. This is similar to automated publishing.

- Set external caches that can be written to when they are rendered. These are subject to the constraints described above for external caches.

Supplying caching metadata

Figure 3 describes the information that is needed to cache a fragment. Each of these is discussed below:

- A fragment id identifies the fragment within the FragmentCache. It must be unique within the scope of the FragmentCache. This scope could be a process within a server (if each server process has its own FragmentCache instance), a server (if a FragmentCache instance is shared across the processes within a server), or a cluster of servers (if a FragmentCache is shared across multiple nodes in a cluster).
- The maximum time interval in seconds that the object should be cached. A negative value implies that there is no time limit.
- Pinned indicates whether the cache entry should be exempt from LRU replacement. If a time limit expires, the entry will be marked to be removed when no longer pinned.
- The URL relative to the server for this fragment. For a top-level fragment (eg, a JSP/Servlet that is externally requested), this could be obtained from the HTTP request object's URL. For a contained fragment, this is the JSP/Servlet file name URL. The id can be the URL, the URL plus some request attributes or not directly related to the URL.
- The list of invalidation identifiers. Each invalidation id is a string that identifies the underlying dynamic content (ie, the raw data). A fragment can use zero or more pieces of raw data, so a fragment can have zero or more invalidation ids. A piece of raw data can be used in one or more fragments, so an invalidation id can have one or more fragments. When the raw data changes, then its invalidation id is used to invalidate all the fragments that depend on it, as well as their parent fragments recursively. It must be unique within the same scope as the fragment id. When a piece of data is used in only one fragment, the invalidation id of the data can be the same as the fragment id. In our implementation, we chose to make this common case simple to configure by having the fragment id always be one of the invalidation ids. When a piece of data is used in multiple fragments, its invalidation id would be different from either of the fragment ids.
- The parent fragment id of the fragment that caches this fragment. This is null if the cached fragment is externally requested (ie, the "parent" is the server) or if the parent is not itself included in the cache. This is used to propagate an invalidation up to containing parents. For a fragment with multiple parents, a union of the Fragment's parent ids is maintained.
- The aggressiveRerender option is a boolean indicating that the fragment should be aggressively rerendered when either its time limit expires or it is invalidated. If aggressiveRerender=true, an expiration or invalidation causes the fragment's value to be set to null and the fragment id put on a queue to be rerendered when idle time permits. If false, an expiration or invalidation causes the fragment to be removed from the cache. It will be rerendered and put back in the cache only when needed by an external HTTP request.
- The set of external caches that are written to when the fragment is rendered.