AN ANALYSIS OF GENERATIONAL CACHING IMPLEMENTED IN A PRODUCTION WEBSITE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Marc Zych

June 2013

COMMITTEE MEMBERSHIP

TITLE:	An Analysis of Generational Caching Implemented in a Production Website
AUTHOR:	Marc Zych
DATE SUBMITTED:	June 2013
COMMITTEE CHAIR:	Alexander Dekhtyar, Ph.D., Associate Professor, Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D., Associate Professor, Computer Science
COMMITTEE MEMBER:	Chris Lupo, Ph.D., Assistant Professor, Computer Science

Abstract

An Analysis of Generational Caching Implemented in a Production Website

Marc Zych

Website scaling has been an issue since the inception of the web. The demand for user generated content and personalized web pages requires the use of a database for a storage engine. Unfortunately, scaling the database to handle large amounts of traffic is still a problem many companies face. One such company is iFixit, a provider of free, publicly-editable, online repair manuals. Like many websites, iFixit uses Memcached to decrease database load and improve response time. However, the caching strategy used is a very *ad hoc* one and therefore can be greatly improved.

Most research regarding web application caching focuses on cache invalidation, the process of keeping cached content consistent with the permanent data store. Generational caching is a technique that involves including the object's last modified date in the cache key. This ensures that cache keys change whenever the object is modified, which effectively invalidates all relevant cache entries. Fine-grained caching is now very simple because the developer doesn't need to explicitly delete all possible cache keys that might need to be invalidated when an object is modified. This is particularly useful for caching arbitrary fragments of HTML without increasing the complexity of cache invalidation.

In this work, we describe the process of implementing a caching strategy based on generational caching concepts in iFixit's website. Our implementation yielded a 20% improvement in page response time by caching fragments of HTML and results of database queries.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

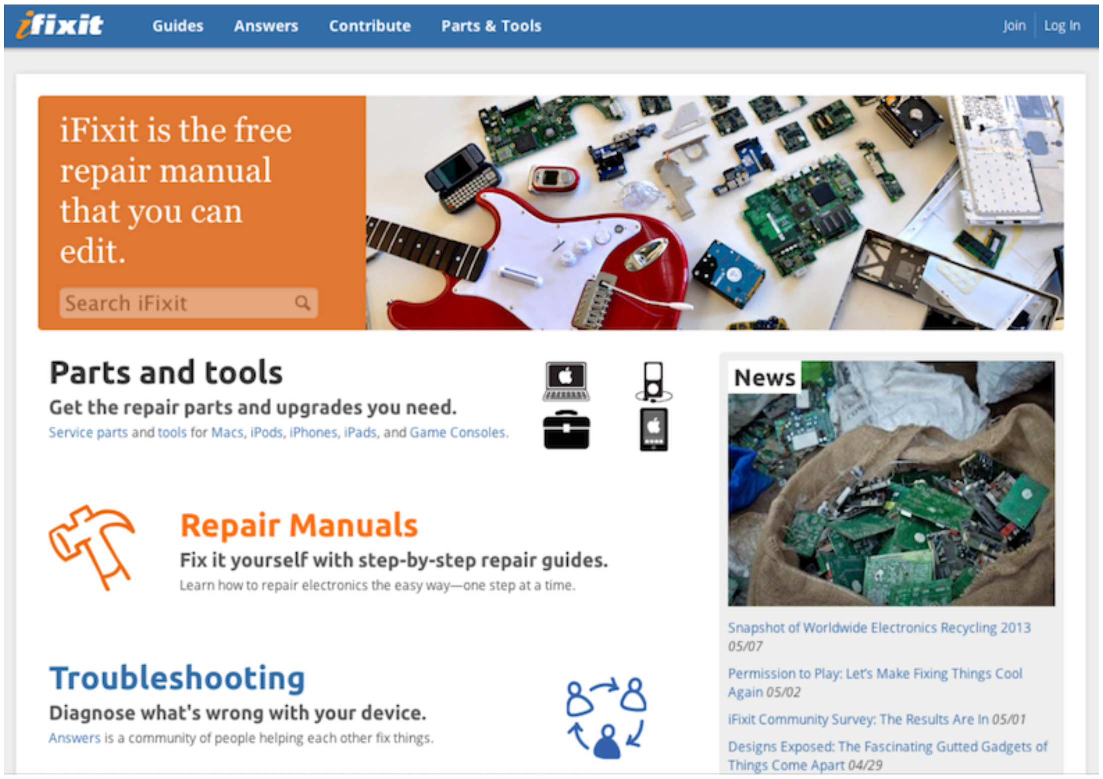
Web scalability has been an issue for websites since the dot-com boom of the late nineties [15]. All moderately sized websites experience decreased web server stability as their breadth of content and active user base grow. If not properly addressed, the website's performance suffers and in some cases the site can get overloaded by requests and stop responding altogether. Websites need to be able to scale with increasing site traffic to continue providing a good user experience.

Some aspects of website scaling are very straightforward. Serving static assets such as Cascading Style Sheets (CSS), JavaScript (JS), and images can be offloaded to Content Delivery Networks (CDN) that handle requests separately from other resources such as the page's HTML. Scaling application machines can be accomplished by load balancing all incoming requests amongst a cluster of application machines. The request is forwarded to the selected machine which eventually returns the HTML of the page to the load balancer which then returns it to the user. Although scaling application machines is trivial, each one still needs to access the same underlying data which is typically stored in a single database machine. The database is the hardest resource to scale because of the inherent problems in distributing data while maintaining consistency.

The ubiquitous solution to scaling websites is some form of caching. Although most web servers employ generous amounts of caching, the methods and approaches used vary widely. Client side browser caching is popular because it avoids a large amount of network overhead. This is typically done by the server checking `E-Tag` and/or `Last-Modified` headers sent by the web browser to determine whether or not the content has changed since the client's previous request. Proxy caching is very similar to browser caching except that it happens at proxy servers located in between the client and the primary host server. This thesis, however, focuses on caching strategies and techniques in web application code that are aimed to reduce database load and decrease page response time.

Although application caching is used widely in industry, it is not a solved problem. Websites still struggle with determining what data to cache and how fine-grained to make the cache entries. Additionally, cache invalidation is the most important aspect of caching strategies and commonly is the driving force behind new solutions. This is primarily because websites are becoming less tolerant of serving stale data especially when their content is constantly changing. Current research is focused on maintaining a high cache hit rate while maintaining cache coherency.

Modern websites are interested in caching as a way to reduce response times and handle increasing volumes of traffic. Traditional website scaling techniques aren't effective because of the vast amounts of user-generated content and dynamic nature of websites. Nearly all web pages need to be generated on demand from content out of the database. Because of this, server load becomes a real issue especially for the database machine.



iFixit is the free repair manual that you can edit.

Parts and tools

Get the repair parts and upgrades you need.

Service parts and tools for Macs, iPods, iPhones, iPads, and Game Consoles.

Repair Manuals

Fix it yourself with step-by-step repair guides.

Learn how to repair electronics the easy way—one step at a time.

Troubleshooting

Diagnose what's wrong with your device.

Answers is a community of people helping each other fix things.

Snapshot of Worldwide Electronics Recycling 2013 05/07

Permission to Play: Let's Make Fixing Things Cool Again 05/02

iFixit Community Survey: The Results Are In 05/01

Designs Exposed: The Fascinating Gutted Gadgets of Things Come Apart 04/29

Figure 1.1: iFixit's home page

One such website currently facing server scaling problems is www.iFixit.com, a moderately sized website that is "the free repair manual that you can edit" [3] (Figure 1.1). iFixit's step-by-step guides and question-and-answer platform are driven largely by user-generated content. All repair guides on iFixit are publicly editable and contain rich revision history. Consequently, all of the content is stored in a database that must be queried in order to respond to requests for data. To reduce server load and decrease response time, iFixit uses Memcached, an open-source in-memory caching system [4], to cache the results of expensive operations. However, since the caching strategy used is an *ad hoc* one, the effectiveness of the cache can be greatly improved by using a different caching technique.

This thesis investigates web application caching techniques in the context of iFixit. We focus on reducing page response time and server load by caching the results of database queries and HTML rendering. Our work makes the following contributions:

1. A survey of existing caching solutions applicable to web applications.
2. An in-depth description of implementing an adapted generational caching solution in iFixit's code base.
3. An analysis of performance improvements by running experiments using real-world traffic.

The rest of this thesis is organized as follows. Chapter 2 provides background on caching and outlines related work, Chapter 3 describes the design and implementation of the caching system implemented in iFixit's code base, Chapter 4 validates our work, and Chapter 5 concludes.

Chapter 2

Background

2.1 Terminology

2.1.1 Cache

At a high level, a cache is used to store results of operations so future requests can be processed faster. A cache is typically not the primary storage location for data. In most cases there is a permanent backing store such as a database which is used to retrieve information and populate the cache. A cache entry is a single piece of data stored in the cache that can be uniquely identified. A cache hit occurs when the requested cache entry is in the cache and the value can be used by the caller immediately. A cache miss occurs when the requested cache entry is not in the cache and the caller must retrieve it from its primary location.

Caching is effective because cache lookups are much faster than retrieving data from its persistent location. In order to make use of the fast cache lookups, caching exploits locality. Temporal locality is the idea that an item that has been referenced recently will likely be referenced again in the future. Spatial locality is the idea that items in nearby storage locations are likely to be referenced in close proximity to one another. As of late, spatial locality has been broadened for websites to include access patterns such as recommended and related content.

Caches can be either static or dynamic. A static cache is one that is not updated as a result of the application running. The application can request data from the cache but it cannot add, modify, or remove cache entries. However, the cache is periodically updated outside of the application to keep it mostly up to date. The timeframe for this differs from application to application but is largely a function of how often the underlying data changes. In contrast, a dynamic cache is one that can be updated as a result of the application execution. The application can freely add, modify, and remove cache entries based on the operations it is performing due to requests.

Dynamic caches have a limited size and may eventually fill to capacity. The eviction policy determines which cache entries to remove when the cache is full and new items are being added. A common eviction policy is Least Recently Used (LRU) which evicts the cache entry that has been accessed least recently. Additionally, some caches allow entries to be set with an expire time. After the specified time has passed, the entry is expired and future requests for it will result in a cache miss.

Cache coherence, also referred to as cache consistency, is the concept of keeping the data in the cache consistent with the data in the primary backing store. One such method is to perform cache invalidation upon data modification which removes any modified items from the cache so the data must be generated upon the next request. A cache stampede can occur when a cache entry that is frequently requested is invalidated. There are then a series of cache misses followed by a series of duplicated requests to the backing store followed by a series of cache sets. These should be avoided if possible because of the redundant work done by each request. If not handled properly, a single cache stampede can crash websites.

2.1.2 Database

Database tables can specify a primary key which declares which columns of the table must have unique values for all tuples stored in the table. Primary keys are useful for uniquely identifying tuples because `SELECT` queries for explicit values of the primary key return at most one record.

The ActiveRecord pattern is a common technique used in web applications to interact with the data stored in a database. In the host language, the developer creates classes that represent tables in the database. This becomes an object-relational mapping (ORM) in that the properties of the class represent columns in the database. Rows can be inserted, updated, and removed by calling the `insert`, `update`, `delete` methods on the object, respectively. This is demonstrated in Figure 2.1.

In this example, `User` is a class that implements the ActiveRecord pattern. A new user is inserted into the database by creating a new `User` object, assigning the properties of the object corresponding to the database columns, and calling `insert`. This `User` can later be retrieved by calling `find`.

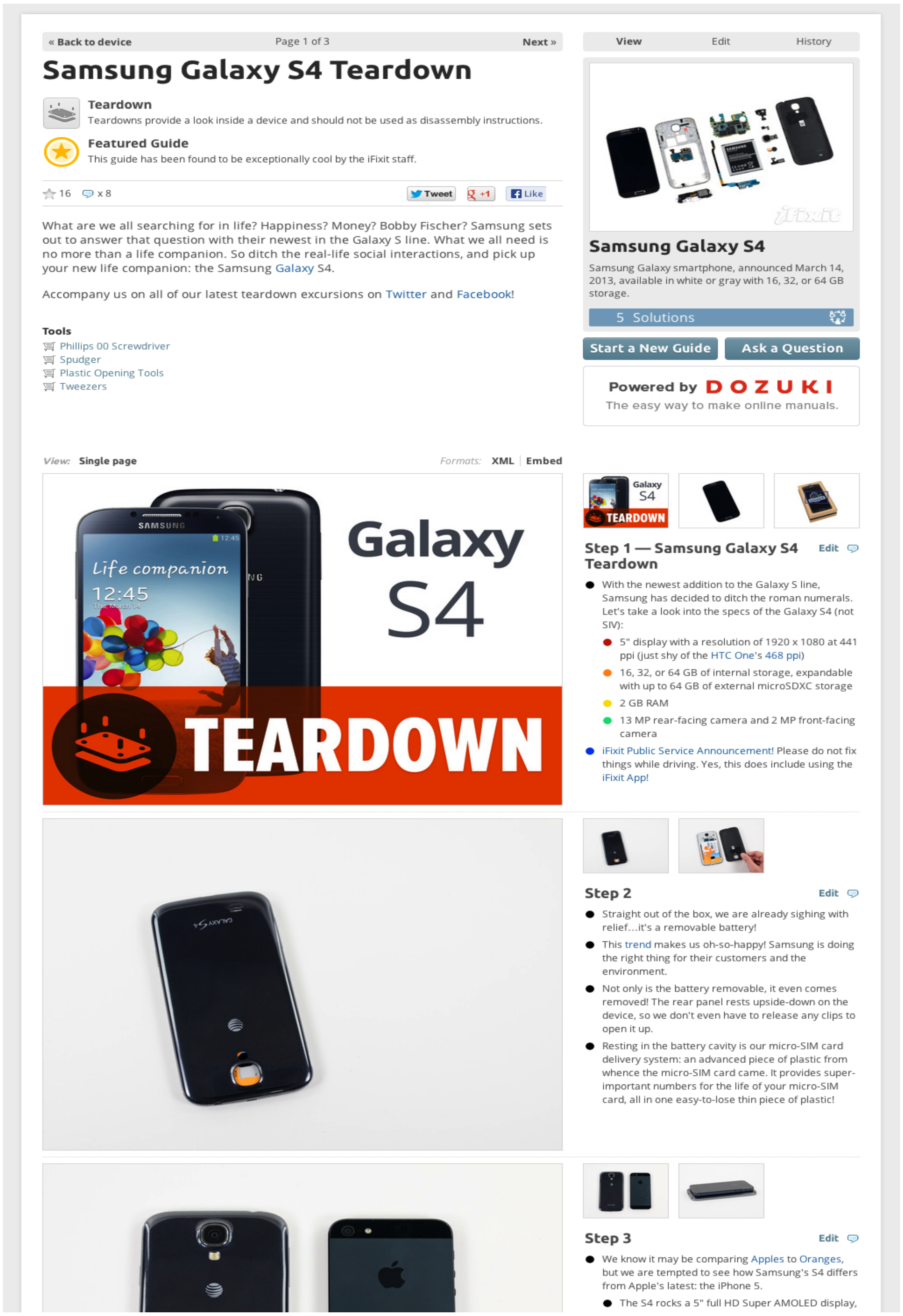
```php
<?php
class User extends ActiveRecord {
   public $userid;
   public $name;
   public $email;
}

function createUser($userid, $name, $email) {
  $user = new User();
  $user->userid = $userid;
  $user->name = $name;
  $user->email = $email;
  $user->insert();
  return $user;
}

function getUser($userid) {
  $user = User::find($userid);
  return $user;
}
?>
```

Figure 2.1: Basic ActiveRecord example

2.2 iFixit

2.2.1 Data and Site Usage

iFixit's most popular content is their step-by-step repair guides, an example of which can be seen in Figure 2.2. An iFixit guide consists of an ordered list of steps as well as meta data about the guide such as title, summary, parts, and tools. Each step consists of a list of bullet points with text, indentation, and color as well some sort of media such as a list of images or a video. To avoid duplicating content, guides can also be included in other guides as prerequisites. The guide is displayed to the user with its prerequisite guides' steps before the primary guide's steps.



« Back to device   Page 1 of 3   Next »

View   Edit   History

Samsung Galaxy S4 Teardown

Teardown
Teardowns provide a look inside a device and should not be used as disassembly instructions.

Featured Guide
This guide has been found to be exceptionally cool by the iFixit staff.

16   x 8   Tweet   +1   Like

What are we all searching for in life? Happiness? Money? Bobby Fischer? Samsung sets out to answer that question with their newest in the Galaxy S line. What we all need is no more than a life companion. So ditch the real-life social interactions, and pick up your new life companion: the Samsung Galaxy S4.

Accompany us on all of our latest teardown excursions on Twitter and Facebook!

Tools
- Phillips 00 Screwdriver
- Spudger
- Plastic Opening Tools
- Tweezers

Samsung Galaxy S4

Samsung Galaxy smartphone, announced March 14, 2013, available in white or gray with 16, 32, or 64 GB storage.

5 Solutions

Start a New Guide   Ask a Question

Powered by DOZUKI
The easy way to make online manuals.

View: Single page   Formats: XML | Embed

Step 1 — Samsung Galaxy S4 Teardown   Edit

- With the newest addition to the Galaxy S line, Samsung has decided to ditch the roman numerals. Let's take a look into the specs of the Galaxy S4 (not SIV):
  - 5" display with a resolution of 1920 x 1080 at 441 ppi (just shy of the HTC One's 468 ppi)
  - 16, 32, or 64 GB of internal storage, expandable with up to 64 GB of external microSDXC storage
  - 2 GB RAM
  - 13 MP rear-facing camera and 2 MP front-facing camera
- iFixit Public Service Announcement! Please do not fix things while driving. Yes, this does include using the iFixit App!

Step 2   Edit

- Straight out of the box, we are already sighing with relief...it's a removable battery!
- This trend makes us oh-so-happy! Samsung is doing the right thing for their customers and the environment.
- Not only is the battery removable, it even comes removed! The rear panel rests upside-down on the device, so we don't even have to release any clips to open it up.
- Resting in the battery cavity is our micro-SIM card delivery system: an advanced piece of plastic from whence the micro-SIM card came. It provides super-important numbers for the life of your micro-SIM card, all in one easy-to-lose thin piece of plastic!

Step 3   Edit

- We know it may be comparing Apples to Oranges, but we are tempted to see how Samsung's S4 differs from Apple's latest: the iPhone 5.
  - The S4 rocks a 5" full HD Super AMOLED display,

Figure 2.2: Example iFixit guide

Most guide fields, such as introduction, conclusion, and step bullets, contain *wiki text*. This text is parsed and rendered into HTML before being returned to the user. An example of this transformation can be seen in Figure 2.3.

```
This is '''a lot easier''' than opening the [guide|1158|iPod touch].
```

(a) Raw wiki text

```
<p>This is <strong>a lot easier</strong> than opening the
<a href="/Teardown/iPod+Touch+Teardown/1158/1">iPod touch</a>.</p>
```

(b) Rendered wiki text

Figure 2.3: Example wiki text transformation

All of iFixit's content is dynamic which means that handling requests involves gathering data from a database, rendering the page in the application language, and finally returning the resulting HTML to the user. A dynamic website allows users to have rich, modifiable content that is always up-to-date.

iFixit certainly has a read-heavy workload; however, writes are still fairly common. Most notably, iFixit is known for its device "teardowns." These events consist of opening a brand new device as soon as it becomes available on the market. The opening process is documented and updated on iFixit's website in realtime. These events attract thousands of visitors to the site in a short period of time. Because of this, the website must be able to handle a read-heavy workload while simultaneously writing to the same data.

In addition to their main website, iFixit runs Dozuki, a Software as a Service (SaaS) business that provides technical documentation to anyone who needs step-by-step instructions. The same code base powers both iFixit and Dozuki which means that we are only interested in performance improvements to both platforms. In particular, the general solutions to web scaling, such as full page caching and web server proxy caching, cannot be used because much of Dozuki's content is private.

The primary bottleneck is the database machine because it has to retrieve data for every incoming HTTP request. In contrast, application machines can easily scale horizontally by load balancing requests between them because they are stateless and operate independently of one another. Sharding the data between multiple database machines to distribute the load is a common solution. However, consistency and race conditions become major issues that must be addressed.

2.2.2 Web Architecture

In this section we describe iFixit's production architecture, depicted in Figure 2.4, at the time of this writing. All of iFixit's traffic goes directly to a load balancer which selects an application machine to handle the request. At any given time there are four to eight application machines that run iFixit's core software which is written in PHP. They get all of their data from one master database machine running the Percona build of MySQL which handles all read and write operations. There is also one slave database machine that is asynchronously replicated from the master database using MySQL's built in replication. However, this machine is used purely as a backup and does not improve the website's performance. All application machines access a single Memcached instance with 4GB of space allocated.

All of this is built using Amazon Web Services (AWS) including Elastic Cloud Compute (EC2) for server instances, S3 for storing static assets and images, CloudFront for serving content out of S3 through a Content Delivery Network

(CDN), and Elastic Block Store (EBS) for raw block storage on each server instance.

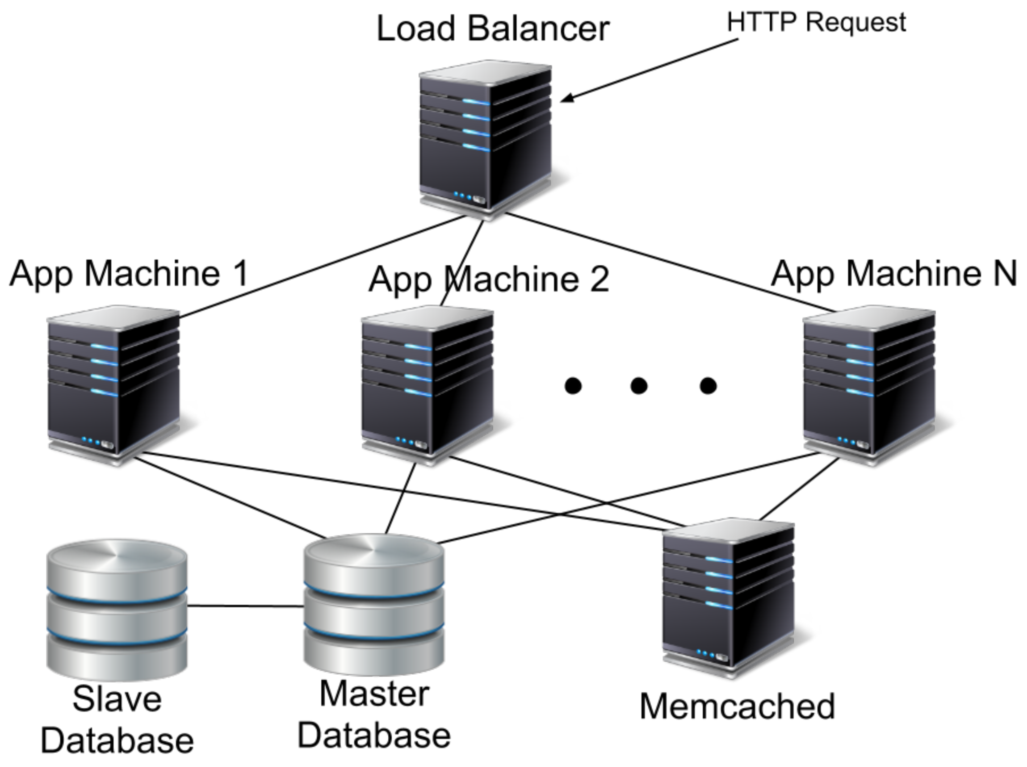


Figure 2.4: iFixit's Production Architecture

2.2.3 iFixit's Current Caching Strategy

iFixit currently has a very *ad hoc* approach to caching. Any specific pages, database queries, or expensive operations are guarded by a cache `get` with a handcrafted cache key. If the object is not in the cache, it is retrieved from the database and stored in the cache for later use. The expiration time is typically on the order of minutes to days depending on the method of invalidation for the data in question. To reduce the risk of cache stampedes, expiration times are varied by *10%* so groups of keys don't expire at the same time. iFixit uses two strategies for cache invalidation that are described below.

The first strategy relies on the cache entry's expiration time to update it with fresh data. After the specified amount of time, the cache is no longer valid so the application is forced to regenerate it and set it in the cache for later use. The cache is never invalidated by the application when the underlying content changes. This strategy is particularly useful for data that isn't required to be absolutely current such as related content, approximate totals, etc.

The second strategy used by iFixit involves deleting cache entries whenever relevant data is changed. It is up to the developer to find the appropriate places to delete cache entries when certain events occur. The advantages to this approach are that it is simple to understand, straightforward to implement, and ensures that fresh data is served if done correctly.

Guides

Currently, a database query is always performed to get the main guide data when looking up objects in the ORM. However, all other data, such as steps, prerequisites, parts, and tools, is cached as one value for 24 hours. This results in a single database query and a single Memcached `get` to retrieve a cached guide. More database queries and Memcached calls are performed to render HTML and retrieve prerequisite guides and non-guide content such as support questions. All cache entries for a guide are invalidated when the guide or any of its children are modified. The next request is forced to retrieve all guide data from the database because none of it is cached.

Rendering HTML for a guide view page can be very costly because guides have 10's to 100's of wiki text fields that must be rendered. Wiki parsing and rendering is expensive because it involves lots of regular expressions, text manipulation, and sometimes database queries to retrieve other content to include in the HTML. To mitigate this, rendered wiki text is cached using an md5 hash of the raw wiki text as the cache key. Even if all rendered wiki text for a guide is cached, there is a large overhead associated with sequential Memcached calls. To reduce this overhead, all rendered guide step bullet wiki text is retrieved with one Memcached getMulti call prior to rendering HTML. Any misses are set in the cache once the wiki text is rendered during HTML generation.

Drawbacks

There are two major problems with these techniques. The first is that the burden of cache invalidation is on the developer. This is because the developer must decide what data to cache and spend time finding all the places where the cache entries should be invalidated. Additionally, this is error-prone because software projects are constantly changing and the fragile caching setup could break.

The next problem is that caching HTML is difficult because each cache entry must be invalidated when data changes. The more cache entries that depend on data, the harder it is to ensure that they will all be invalidated at the appropriate time. This is especially true for fragments of HTML because the cache entries are typically defined in front-end templates but must be invalidated from back-end code when the data is modified. In general, the difficulty of cache invalidation for any given datum is proportional to the number of cache entries dependent on that datum and how fine-grained those cache entries are. As a result, iFixit doesn't currently cache any HTML fragments.

Our caching strategy aims to solve these problems by extending cache expiration times, providing a means for caching HTML, and making cache invalidation happen automatically when the underlying data has changed.

2.3 Related Work

2.3.1 Research Challenges

Most research is focused on addressing two core challenges of caching: cache coherency and ease-of-use. In some cases, maintaining cache coherency is trivial because there is only one cache entry associated with the underlying data. However, with more complex data the situation is much more challenging. There may be dependencies between objects which means that multiple cache entries need to be invalidated at the same time. To further complicate things, the dependencies between objects may be dynamic and must be resolved at run time on a per object basis. This is very common for Web 2.0 sites that are driven by user-generated content.

Ease-of-use addresses the problem that manually controlling the cache is tedious and error-prone. Leaving it up to the application developer to correctly `get`, `set`, and `delete` cache entries results in an *ad hoc* caching strategy. Most research regarding ease-of-use focuses on developing a system that automatically handles all caching operations. This leaves the developer to write application-specific code instead of caching boilerplate.

2.3.2 Incorporating Cache

Caching Technologies

Most web application caching research revolves around caching strategies. It turns out that the underlying cache daemon is largely irrelevant. Memcached is an example of a dynamic cache daemon which is well-suited for dynamic websites. It is by far the most popular web caching system; it is currently in use by Facebook, Wikipedia, YouTube, and many others [4].

Search engines use static caches as well because a large number of search queries return the same data for long periods of time. These queries can be determined by analyzing historical data which would be difficult to do using a dynamic cache [7]. In particular, the knapsack problem can be used to fill the static cache if the request frequency and size of cacheable items can be determined [7]. Additionally, many papers propose a 2 or 3-level cache that incorporates both static and dynamic caches [8, 7].

The exception to this trend is Voras and Zagar's Memcached alternative, SQL-Cached [19]. This tool offers a few features that Memcached doesn't: complex data types and rich querying for retrieval and invalidation. This is possible because SQLCached uses SQLite as its caching mechanism.

Implementing in an Application

Many approaches have been proposed for how to implement caching strategies in web application code. The most straightforward method is to explicitly get, set, and invalidate caches wherever they are needed in the application. However, this is tedious and error-prone [14, 12]. A preferable alternative is to have caching built directly into the underlying framework code. For example, Ruby on Rails, a popular web framework, provides a very rich data model that handles most of the caching details behind the scenes [14].

Gupta, Zeldovich, and Madden developed Cache Genie, an ORM for the Django web framework that automatically handles all necessary caching operations [12]. The project registers Python callbacks for database triggers that are called when queries are run on the database. This allows the tool to update and/or invalidate cache entries depending on the queries. The major advantage of this approach is that no matter how the data is changed, the cache remains consistent because cache invalidation occurs at the data layer. Other projects have used database triggers with similar results [10].

2.3.3 Cache Consistency

Write-Through Caching

Write-through caching is a technique that involves storing the data in the cache at the same time as it is being saved in the persistent store [18]. This technique improves cache hit rate because cache misses will only happen as a result of cache entry eviction or server failure. However, this technique only works for simple data types. Any other objects that depend on the current object being saved can't be updated in the cache because that data isn't directly accessible. Those values must be prefetched and set in the cache or the relevant cache entries must be invalidated to regenerate the values upon next request in a non-write-through manner.

Generational Caching

Another caching technique is referred to as key-based cache expiration [14] and generational caching [17]. The idea is that a timestamp, which corresponds to the last time the object was updated, is included in the key for any cache entries that depend on that object. The timestamp is updated every time the object is modified thus invalidating all necessary cache entries. This approach also updates the timestamps of any objects that depend on the modified object. Generational caching makes fine-grained caching trivial because it isn't necessary to enumerate all potential cache keys that depend on an object when it is modified. Cache entries can be defined arbitrarily and they will be invalidated when the underlying object is updated as long as the cache key contains the timestamp. This technique is employed by 37Signals, a SaaS that provides project management tools and is the origin of the popular web application framework Ruby on Rails[1], and various other websites [14].

One downside to this method is the amount of cache garbage, cache entries that will never be referenced again, that is generated because cache entries are never deleted. However, this isn't much of an issue because most caching technologies use an LRU eviction policy so unused cache entries are eventually deleted when the cache fills up. Additionally, increasing the cache's capacity to make this even less of an issue is trivial with Memcached.

Prefetching

Prefetching content is another common solution for caching that takes advantage of spatial locality. This technique involves caching content before it is requested so it can be retrieved quickly when it is needed. Generally, an event triggers the precaching system to intelligently decide what content to prefetch in the hopes that the data will be requested soon.

Challenger, Iyengar, and Dantzig developed a prefetching caching system that addresses the problem of cache stampedes [10]. Rather than invalidating any necessary cache entries when data changes, the fresh content can be prefetched and the cache's value will be updated in place using write-through caching. This approach avoids cache misses at the cost of slight data staleness because the old copy will still be served while the fresh copy is being prefetched. The website for the 1998 Winter Olympic games deployed this caching technique and reached nearly 100% cache hits [10]. Their system uses a dependency graph to determine which cache entries need to be updated when any given data is modified [10]. When an object is updated, all of the cache entries that have an edge with that object need to be prefetched along with the original object.

Expiration Times

Cambazoglu, Junqueira, and Plachouras use the cache expiration time to invalidate cache entries [9]. The idea is that serving slightly stale data isn't terrible for search engines. Additionally, the rate of change can be determined for a given query and can be factored into the expiration time. This allows frequently changing data to have a shorter time to live and infrequently changing data to have a longer time to live.

2.3.4 Determining What to Cache

Some research has been done to determine what data, when cached, improves cache hit rate the most. Many sources propose that fragments of HTML are most useful because a cache hit for HTML allows the server to avoid generating the HTML as well as retrieving the underlying data from the persistent store [11, 10, 13].

Baeza-Yates et al. suggested that enforcing cache admission policies is a valuable way to improve hit ratio [8]. The driving idea behind it is that there are many singleton queries that pollute the cache because they are not used again. Detecting such queries and leaving them out of the cache leaves more room for caching results of frequently accessed queries thus increasing cache hit ratio. However, Cambazoglu, Junqueira, and Plachouras claim that the eviction policy doesn't have a significant impact on cache hit ratio because evictions are rare since caches can easily be expanded to hold more entries [9].

2.4 Memcached

In the words of its creators,

> "Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering[4]."

There are two main operations in Memcached: `get` and `set`. Clients perform `get` calls by passing in a string cache key that identifies the data that is being requested. For example, `users-1234` could be used to identify the `User` object with `userid` of `1234`. The data is returned to the caller if it is in the cache, otherwise the application must retrieve the data from its primary location. Data is added to Memcached by calling `set` with a cache key, expiration time, and the data to associate with the key. Another specialty command is `getMulti` which is used to retrieve multiple keys in one operation. This is used to improve client performance by reducing the network overhead associated with retrieving many keys in succession. All Memcached operations are constant time (`O(1)`).

There are three ways for cached data to be removed from Memcached. The expire time the data was set with determines the maximum lifespan of the cache entry for which it is valid. Once the data has expired it is no longer returned for `get` requests. Before data has expired, however, items can be evicted due to memory pressure. This occurs when new items are being added to the cache and there isn't any available space. Expired items are evicted first if any are found, otherwise the Least Recently Used (LRU) policy is used to select a cache entry to evict. Finally, cache entries can also be removed by performing `delete` requests with the cache entry's cache key.

To reduce the effects of memory fragmentation, Memcached employs a technique called slab allocation. The total memory available to Memcached is partitioned into 1MB pages. A chunk is a block of memory used to store a single cache entry. Slab-classes determine the maximum size of data to be stored in the slab and consequently the size of chunks in pages assigned to the slab-class. Pages are assigned to slab-classes on an as-needed basis. When storing data in Memcached, the slab-class with the smallest chunk size that the data can fit in is used. Any remaining space in the chunk cannot be used to store any other data and therefore ends up being wasted space. However, the amount of space wasted in slab allocation is less than a naive implementation which results in memory fragmentation. All replacement algorithms operate on a per-slab-class basis.

Memcached is distributed in nature. Adding more servers to a web application involves adding the new Memcached instance's IP address and port to the list of servers that the Memcached client library uses. Keys map to exactly one server so the total memory available in the Memcached cluster is the sum of memory available on each instance. The client library determines which server in the cluster to send the request to based on the cache key. Adding or removing servers negatively affects the cache's performance because cache keys may map to different servers than they did previously. Memcached uses consistent hashing[16] for all cache key lookups; this minimizes the adverse effects of adding or removing servers because the majority of cache keys map to the same servers before and after the change.

Because of these features, Memcached is well-suited for web application caching. All of its operations are constant-time and are distributed amongst all nodes in the cluster. The result is that Memcached calls can scale much better than database queries. Ultimately, websites can cache expensive operations using Memcached to improve performance by decreasing page response time and server load.

Chapter 3

Design and Implementation

In this chapter, we explain in depth the caching solution we implemented for iFixit and later analyze in Chapter 4. But first, we must reiterate iFixit's requirements for such a caching solution.

R1 The caching strategy must be largely transparent to the developer. The developer should only be required to determine what data to cache, not how to cache the data and how to invalidate the cache entries.

R2 The caching strategy must allow for fine-grained caching without adding complexity to cache invalidation.

R3 The caching strategy must provide a means of caching and invalidating fragments of HTML.

R4 The caching strategy must result in increased performance in regards to response time and database load.

The rest of this chapter is organized as follows. In Section 3.1 we explain the high-level design of the caching strategy implemented for iFixit and in Section 3.2

we cover the details of implementing the design in iFixit's code base.

3.1 Design

Out of existing solutions, generational caching most closely meets iFixit's requirements because cache keys change whenever the underlying data changes, so cache entries no longer need to be explicitly invalidated. This allows there to be many cache entries that depend on a single piece of data and the values will not need to be invalidated. For these reasons, we used generational caching concepts as a starting point for our caching solution.

3.1.1 What is being cached

This section describes the three primary types of data that are cached in our caching strategy. The first data type is objects constructed from query results that represent a tuple in the database (see Figure 3.1a). These are used as an ORM which can make modifications to the database as well as retrieve data for consumption. The second data type is fragments of HTML generated in templates like the one depicted in Figure 3.1b. These typically produce output based on data retrieved from the database and need to be invalidated anytime that data has changed. Finally, results of *ad hoc* queries, like the one in Figure 3.1c, are also cached. These typically are used in addition to the ORM to gather more complex data.

```php
<?php
class Guide extends ORMObject {
  private $guideid;
  private $langid;
  private $title;
  private $difficulty;
  private $steps;
}
?>
```

(a) ORM Object

```html
<div class="guide">
  <h3><?= $guide->title ?></h3>
  <p>Difficulty: <?= $guide->difficulty ?></p>
  <div class="steps">
  <? foreach ($guide->steps as $step): ?>
    <div class="step">
      <?= $step->title ?>
      <? /* Step bullet HTML generation. */ ?>
    </div>
  <? endforeach ?>
  </div>
</div>
```

(b) HTML template

```sql
SELECT guideid, langid
FROM guide_prereqs
WHERE prereq_guideid = ? AND prereq_langid = ?
```

(c) *Ad hoc* query

Figure 3.1: Data types being cached

3.1.2 Caching Approach

We first discuss how we incorporate caching into our ORM and how it is used for iFixit's guides. The guide and step ORM objects can be found in Figure 3.2.

```php
<?php
class Guide extends ORMObject {
  protected static $cacheCols =
   ['modified_date'];

  // Primary key.
  protected $guideid;
  protected $langid;

  // Columns from guides table.
  protected $title;
  protected $difficulty;
  protected $device;
  protected $modified_date;

  // From other tables.
  protected $steps;
  protected $parts;
}
?>
```

(a) Guide ORM object

```php
<?php
class Step extends ORMObject {
  protected static $cacheCols =
   ['modified_date'];

  // Primary key.
  private $stepid;

  // Columns from steps table.
  private $guideid;
  private $langid;
  private $title;
  private $orderby;
  private $modified_date;

  // From other tables.
  private $media;
  private $lines;
}
?>
```

(b) Step ORM object

Figure 3.2: Example guide and step ORM objects

The primary table for an ORM object is the database table that it represents. For example, the `Guide` ORM object's primary table is `guides`, which contains `guideid`, `langid`, `title`, `difficulty`, etc. The primary key of a database table can be used with the ORM to select a single record. Once the record is loaded, other data not contained in the primary table is loaded and then the entire object is cached. For guides, the `Guide` object is retrieved from the DB and then the `Step` objects are loaded afterwards. By default, the cache key for objects only includes the values of the primary key columns.

Generational caching relies on a value in the primary table being updated whenever any data in the table is updated. This value can be incorporated into the object's cache key which ensures that the value in the cache will always be correct for that cache key. The cache key changes when the data changes so cache entries using that cache key don't need to be invalidated. To accomplish this, individual classes extending the ORM specify what columns in addition to the primary key to include in the cache key. For guides, this value is the `modified_date` which is set to the current time in microseconds anytime the guide or any of its children are modified. For example, a `Guide` could have cache keys `Guide/1234-en-1368160000.000000` and `Guide/1234-en-1368161111.111111` before and after it is modified, respectively.

In most cases, the additional cache key columns are not known before retrieving the object and must be retrieved before checking the cache. This is typically because the request URL only specifies the primary key and not any other additional information. To avoid hitting the database as much as possible, the entire database tuple that is used to generate the cache key is cached using its primary key. This cache entry is deleted anytime the object is saved.

For example, only the `guideid` and `langid` are initially available when a guide request hits iFixit's servers. For a `guideid` of `1234` and `langid` of `en`, the cache is checked for the `Guide` object using `Guide/1234-en` as the cache key. If it isn't found in the cache, it is retrieved from the database and stored in the cache. This object does not contain any data from other tables such as `Steps`. The full cache key can then be constructed, for example `Guide/1234-en-1368164354.918344`, and used to fetch the rest of the content. If it exists in the cache then it can be returned to the caller immediately. Otherwise, it must be retrieved from the database. Fortunately, the primary table tuple has already been fetched from the database so we only need to load the additional data for the object and store it in the cache.

This strategy works especially well when retrieving an object's additional data is very expensive. If no additional data is loaded then the "fully loaded" object is the exact same as the one fetched when gathering more data to construct the cache key. In this case there is no benefit to using generational caching for retrieving the ORM object because the data cached by the primary key is the same as the data cached with the full set of columns. However, the additional columns can be used in cache keys for data that depends on it to achieve automatic cache invalidation. The amount of extra data loaded in addition to the primary table to make the overhead of generational caching worthwhile varies on a case-by-case basis. This approach is effective for `Guide`s because there is a large amount of additional data to retrieve. Although only `Step`s are described in this example, there are several other tables that must be queried to display a guide to the user.

For fragments of HTML and *ad hoc* query results that can only change when the ORM object changes, the object's full cache key can be used in addition to a qualifying string such as `view-all-html`. Any cache entry that does this will be automatically invalidated when the object changes because the cache key will change. This allows for fine-grained caching without the need to enumerate and delete all cache keys that depend on an object.

Occasionally HTML fragments contain data that changes at different times than the ORM object. That value could be incorporated into the cache key but it would increase the number of possible cache entries for the same data thus reducing its effectiveness. For example, `Steps`' `modified_date`s are updated every time they are modified. However, the step number displayed to the user can be different for the same step on different guides because of the order of prerequisite guides. Rather than including the step number in the cache key, a placeholder value is used in place of the step number. After the HTML is retrieved from the cache, the placeholder is replaced with the actual step number.

3.2 Implementation

3.2.1 Existing Infrastructure

Before we can go into the details of the implementation of generational caching, some background on iFixit's existing code must be presented. Most data is retrieved using a custom ORM that was built in-house. The primary operation for retrieving data through the ORM is `findAllByKey` which returns objects based on their primary keys in the database. A list of primary keys is provided to `findAllByKey` and the ORM selects those tuples from the database and returns the objects constructed from them.

3.2.2 Caching ORM Objects

Most of the work performed while implementing the caching strategy involved augmenting iFixit's ORM to cache objects when they are retrieved with `findAllByKey`. In order to add caching to `findAllByKey`, we add a wrapper method called `findAllCached`. At a very high level, this method constructs cache keys for the given primary keys and performs a cache `getMulti` on them. Any objects that were not cached are retrieved from the database using `findAllByKey` and are then stored in the cache.

We allow classes derived from the ORM to specify which columns, in addition

Operations	Call stack	Description
	`findAllCached`	Call `findAllCached(light)` to get modified dates to construct cache keys.

`GET guide-1234-en`
`GET guide-5678-en`
	`findAllCached`
	`findAllCached(light)`
		Cache miss for light Guide objects. Call `findAllByKey(light)` to retrieve data from primary guide table.

`GET Guides`	`findAllCached`
	`findAllCached(light)`
	`findAllByKey(light)`
		Retrieve data from primary guide table.

`SET guide-1234-en`
`SET guide-5678-en`
	`findAllCached`
	`findAllCached(light)`
		Set light Guide objects in the cache.

`GET guide-1234-en-684`
`GET guide-5678-en-798`
	`findAllCached`	Cache miss for full Guide objects with modified dates in cache keys.

`GET Steps`
`GET Step lines`
`GET Step bullets`
`GET Flags`
`GET Parts and tools`
	`findAllCached`
	`findAllByKey`
		Retrieve rest of data from other guide tables.

`SET guide-1234-en-684`
`SET guide-5678-en-798`
	`findAllCached`	Set full Guide objects in the cache.

Table 3.1: Example call of findAllCached for uncached content

to the primary key, to include in the cache key. More data must be retrieved from the database to construct the cache key if only the primary keys are provided to `findAllCached` and the derived class specifies additional cache key columns. Fortunately, we already have a method, `findAllByKey`, that retrieves objects based on their primary key that can be used to retrieve more data. However, we can do one better and cache that operation by using `findAllCached` instead. For this call, the `light` flag is passed to `findAllCached` to indicate that only the primary table should be loaded. Otherwise, all other relevant data is loaded which

Operations	Call stack	Description
	`findAllCached`	Call `findAllCached(light)` to get modified dates to construct cache keys.

`GET guide-1234-en`
`GET guide-5678-en`
	`findAllCached`
	`findAllCached(light)`
		Cache hit for light Guide objects.

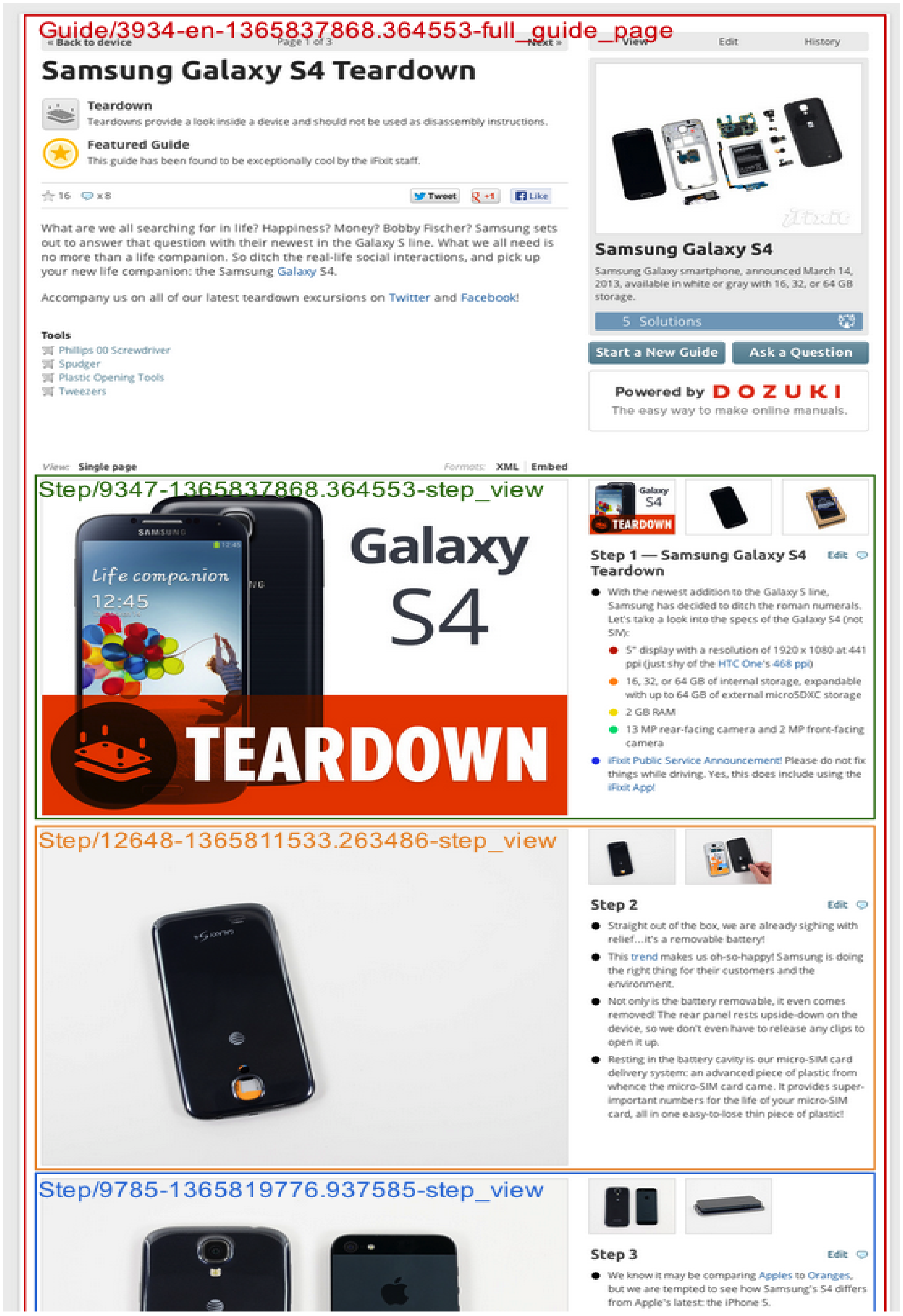
`GET guide-1234-en-684`
`GET guide-5678-en-798`
	`findAllCached`	Cache hit for full Guide objects with modified dates in cache keys.

Table 3.2: Example call of findAllCached for cached content

is exactly what the first `findAllCached` call does. The recursive call constructs a cache key using only the primary key to avoid infinite recursion. Example calls of `findAllCached` for cached and uncached content are found in Tables 3.1 and 3.2.

The Memcached operations are performed using `getMulti` to minimize the network overhead involved in doing multiple sequential Memcached `get`s. All cached objects are retrieved with one `getMulti` call and the uncached objects are retrieved with one database query. The object is cached for later use once all relevant data from other tables has been retrieved.

Currently when a step is modified, all cache entries depending on the guide the step belongs to are invalidated. The next request is forced to retrieve all guide data from the database. In our generational caching scheme, a step edit will update the modified dates of the edited step and the guide it belongs to. The next request will retrieve the main guide data and edited step from the database but all other data will be cached.



Guide/3934-en-1365837868.364553-full_guide_page

« Back to device Page 1 of 3 Next »

View Edit History

Samsung Galaxy S4 Teardown

Teardown
Teardowns provide a look inside a device and should not be used as disassembly instructions.

Featured Guide
This guide has been found to be exceptionally cool by the iFixit staff.

16 x8 Tweet +1 Like

What are we all searching for in life? Happiness? Money? Bobby Fischer? Samsung sets out to answer that question with their newest in the Galaxy S line. What we all need is no more than a life companion. So ditch the real-life social interactions, and pick up your new life companion: the Samsung Galaxy S4.

Accompany us on all of our latest teardown excursions on Twitter and Facebook!

Tools
- Phillips 00 Screwdriver
- Spudger
- Plastic Opening Tools
- Tweezers

Samsung Galaxy S4
Samsung Galaxy smartphone, announced March 14, 2013, available in white or gray with 16, 32, or 64 GB storage.

5 Solutions

Start a New Guide | Ask a Question

Powered by DOZUKI
The easy way to make online manuals.

View: Single page Formats: XML | Embed

Step/9347-1365837868.364553-step_view

Step 1 — Samsung Galaxy S4 Teardown Edit

- With the newest addition to the Galaxy S line, Samsung has decided to ditch the roman numerals. Let's take a look into the specs of the Galaxy S4 (not SIV):
  - 5" display with a resolution of 1920 x 1080 at 441 ppi (just shy of the HTC One's 468 ppi)
  - 16, 32, or 64 GB of internal storage, expandable with up to 64 GB of external microSDXC storage
  - 2 GB RAM
  - 13 MP rear-facing camera and 2 MP front-facing camera
- iFixit Public Service Announcement! Please do not fix things while driving. Yes, this does include using the iFixit App!

Step/12648-1365811533.263486-step_view

Step 2 Edit

- Straight out of the box, we are already sighing with relief…it's a removable battery!
- This trend makes us oh-so-happy! Samsung is doing the right thing for their customers and the environment.
- Not only is the battery removable, it even comes removed! The rear panel rests upside-down on the device, so we don't even have to release any clips to open it up.
- Resting in the battery cavity is our micro-SIM card delivery system: an advanced piece of plastic from whence the micro-SIM card came. It provides super-important numbers for the life of your micro-SIM card, all in one easy-to-lose thin piece of plastic!

Step/9785-1365819776.937585-step_view

Step 3 Edit

- We know it may be comparing Apples to Oranges, but we are tempted to see how Samsung's S4 differs from Apple's latest: the iPhone 5.

Figure 3.3: Annotated guide page with HTML caching

3.2.3 Caching HTML and *Ad Hoc* Queries

We also cache fragments of HTML because HTML generation can be fairly slow and underperforming. An annotated example guide demonstrating this can be seen in Figure 3.3. To help facilitate HTML fragment caching, we implemented a few different caching functions for use in HTML templates.

The first set of functions is `cacheStart` and `cacheEnd` which are used for caching a single block of HTML. `cacheStart` accepts either an ORM object or a string cache key. Another string is supplied that is added to the cache key to differentiate this fragment of HTML from other fragments constructed from the same object. If the HTML was found in the cache it is output to the user and `cacheStart` returns `true`. Otherwise output buffering is started to capture the HTML that is generated when running the template code and `cacheStart` returns `false`. A call to `cacheEnd` signals the end of the cached HTML section. This function closes output buffering, gets the contents of the buffer, caches it, and finally outputs the HTML to the user. A basic example is shown in Figure 3.4.

```
<? if (!cacheStart($guide, 'full_guide_page', CACHE_FOREVER)): ?>
  <div class="guide">
    <h3><?= $guide->title ?></h3>
    <p>Difficulty: <?= $guide->difficulty ?></p>
    <? /* Step HTML generation. */ ?>
  </div>
<? cacheEnd(); endif ?>
```

Figure 3.4: Example usage of cacheStart and cacheEnd

The second caching function we wrote is `batchCache` which is designed for caching multiple fragments of HTML. The most common use case for it is rendering a list of objects one after the other e.g. search results. `batchCache` is better than `cacheStart` and `cacheEnd` for this circumstance because of the network

```php
<?php
function batchCache($cacheKeys, $where, $expire,
 $renderFunction) {
  $templateCacheKeys = [];
  foreach ($cacheKeys as $key => $ormObject) {
    $templateCacheKeys["$key-$where"] = $ormObject;
  }

  $allHtml = cacheGetMulti($templateCacheKeys, $expire,
    function($missing) use ($renderFunction) {
      $found = [];

      foreach ($missing as $cacheKey => $ormObject) {
        ob_start(); {
          $renderFunction($ormObject);
          $html = ob_get_contents();
        } ob_end_clean();

        $found[$cacheKey] = $html;
      }

      return $found;
    }
  );

  foreach ($allHtml as $html) {
    echo $html;
  }
}
?>
```

Figure 3.5: Implementation of batchCache

overhead for sequential Memcached calls. This is especially true for fragments of HTML whose generation time is close to the average time of a Memcached get. batchCache minimizes network overhead by using Memcached's getMulti operation.

The implementation of batchCache is shown in Figure 3.5. batchCache accepts an array of ORM objects and, much like cacheStart, an additional string to differentiate these cache entries from others constructed from the same objects. batchCache calls cacheGetMulti with the cache keys returned by the provided objects' cacheKey methods. For each cache miss, output buffering is started and the supplied rendering function is called with the ORM object. Output buffering is then closed and the resulting HTML is returned to cacheGetMulti which stores it in the cache. Once all the HTML has been generated, cacheGetMulti returns the HTML fragments in the same order that the cache keys were provided. Finally, every fragment of HTML is output to the user in the same order that their corresponding objects were provided. This is demonstrated for Steps in Figure 3.6, which is the cached steps version of Figure 3.1b.

```
<div class="steps">
<? batchCache($guide->steps, 'step_view', CACHE_FOREVER,
    function($step) { ?>
  <div class="step">
    <?= $step->title ?>
    <? /* Step bullet HTML generation. */ ?>
  </div>
<? }); ?>
</div>
```

Figure 3.6: Example usage of batchCache for Steps

The same approach can be applied to caching *ad hoc* query results. Figure 3.7 demonstrates this by using cacheGetAndSet. The cache key is constructed using the guide's full cache key and the string `-prereqs`, so it will be invalidated when the guide is modified. If the value is not cached, `cacheGetAndSet` calls the provided function and caches the result.

```php
<?php
$cacheKey = $guide->cacheKey() . "-prereqs";
$prereqs = cacheGetAndSet($cacheKey, function() use ($guide) {
  return dbQuery("
    SELECT guideid, langid
    FROM guide_prereqs
    WHERE prereq_guideid = ? AND prereq_langid = ?",
   [$guide->guideid, $guide->langid]);
});
?>
```

Figure 3.7: *Ad hoc* query caching

Chapter 4

Validation

Validation is a fairly straightforward process. The basic approach involves gathering various statistics with and without our caching strategy during an experiment based on a repeatable and representative set of input data.

4.1 Experimental Setup

We recorded requests to www.iFixit.com for a full week and selected two sets of representative data to use as input to the experiments. The first data set is typical traffic, which involves accessing a high breadth of content at a fairly steady pace. This experiment lasts 48 hours to observe the long term effects of the caching strategies. The second data set is traffic during iFixit's Samsung Galaxy S4 Teardown[6], which involves a large number of reads and writes to the same resource. This experiment only lasts 24 hours because the interesting results for this data set are regarding the large spike in traffic which only lasts a few hours.

Since our caching strategy was only applied to guides, the request logs are filtered to only include requests viewing or modifying guides. Otherwise, the results would be affected by other non-guide requests which are not using our generational caching method.

The test environment, depicted in Figure 4.1, is an isolated clone of iFixit's production architecture consisting of a load balancer, two application machines, and a database machine. Since the traffic replays are a subset of iFixit's traffic, two application machines are more than enough to handle the load. All machines are AWS EC2 nodes running 64-bit CentOS 5.4. The load balancer is a `m1.large` instance with 7.5 GiB memory and 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each). This machine runs Memcached `1.4.15` which is accessed from the application machines. The database machine is a `m1.xlarge` instance with 15 GiB of memory and 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each). This machine runs MySQL `5.5.29 Percona Server` with EBS-striped volumes. The application machines are `c1.medium` instances with 1.7 GiB of memory and 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each). More information regarding EC2 specs can be found on AWS's website [2].

We test 2 caching scenarios. The *control* is the caching strategy that is currently in use by iFixit, outlined in Section 2.2.3. The *experimental* is the generational caching strategy described in Chapter 3.

We replay the requests in the 2 data sets for both *control* and *experimental* for a total of 4 runs. Requests are performed using a NodeJS script running on the load balancer making requests to `localhost` to reduce latency between the simulated client and the web servers. Every request records Memcached, MySQL, and various other statistics to the database.

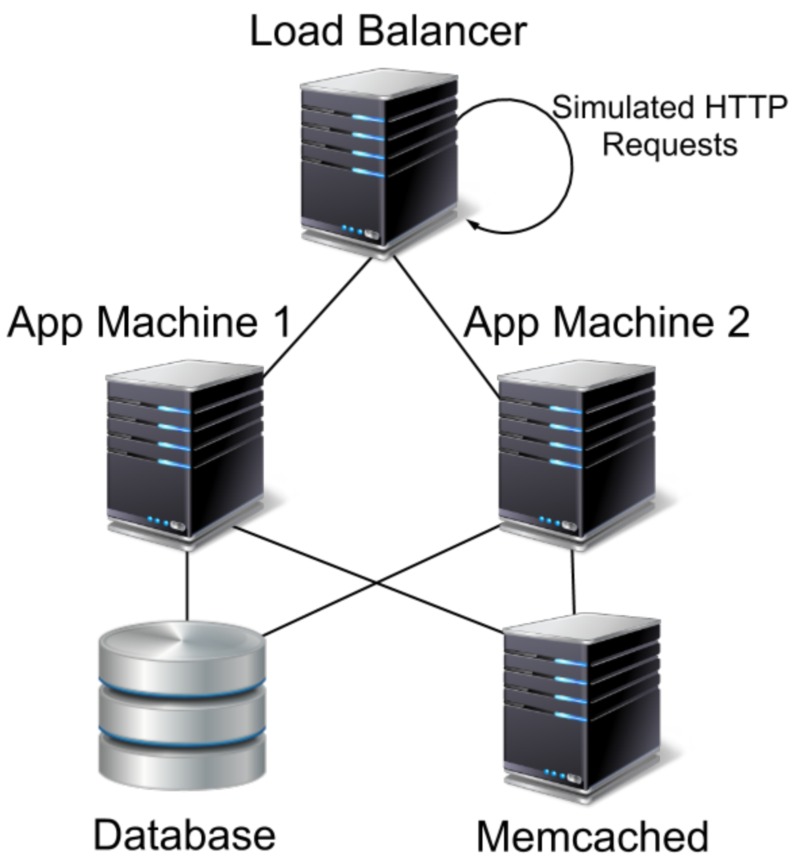


Figure 4.1: Experimental Setup

Before each experiment is run, the database is restored to the state it was in when the requests were originally made to www.iFixit.com. This ensures that the data is the same between runs and all views and edits are working with the same data. The stats logging table is truncated, so only requests during the experiment are in the table when the test is complete. Finally, Memcached is restarted, which empties its contents. After the experiment is run for the full duration, the stats logging table is extracted using `mysqldump` so the data can later be analyzed.

4.2 Measured Values

The most general value we compare is *page response time*, the total amount of time requests spend on our servers. Response time should decrease because more data is cached and can be returned faster than before. Similarly, *HTML generation time* should decrease because fragments of HTML are now cached.

The interesting values for Memcached stats are *get count*, *miss count*, and *hit rate*. All of these values vary drastically from page to page depending on whether or not the content is cached. Pages with uncached content will have a large number of misses while pages with cached content will have either a few cache hits for a large amount of content or many cache hits for finer-grained content that is nested in other cache entries. In general, Memcached misses should decrease and cache hit rate should increase because cache entries aren't set to expire and the only misses will be for new or changed content.

Finally, the interesting values for MySQL are *query count* and *total query time*. Query count should go down because the cache should serve more requests that previously were handled by the database. Additionally, network overhead is reduced because more content is retrieved in batch using fewer queries. For both of these reasons, total database query time should decrease.

4.3 Results

The following are results from the experiments outlined above. *Teardown traffic* refers to the Galaxy S4 Teardown traffic starting right before the teardown was published while *Normal traffic* refers to two standard days of traffic. Both sets of traffic start at 11AM Mountain Standard Time (MST). Figure 4.2 shows the requests per second over time for both data sets. The *Teardown Traffic Excluding Teardown* data points are from the teardown traffic excluding all views and edits for the teardown content, which is very similar to the normal traffic data set.

All "over time" graphs are constructed from 15 minute groupings of requests for a total of 96 and 192 data points for the teardown traffic and normal traffic, respectively. The *Control* data points are from the code that is currently running www.iFixit.com while the *GC* data points are from our improvements using generational caching.

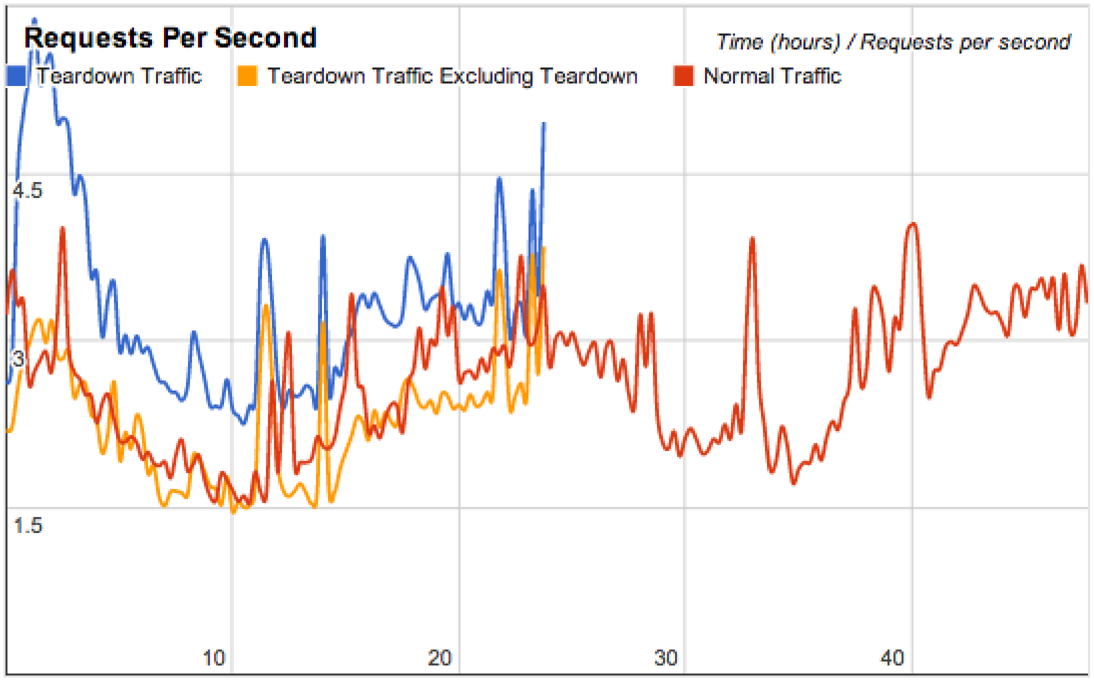


Figure 4.2: Requests per second over time for both data sets

Average times for responses, Memcached, database queries, and HTML generation during both experiments are shown in Table 4.1. As we expected, response, database query, and HTML render times decreased for our generational caching implementation. However, Memcached time has remained approximately the same.

Run	Response	Memcached	DB	HTML
Teardown Control	236.33	78.26	67.56	59.95
Teardown GC	196.67	79.75	48.46	23.52
Normal Control	235.17	76.11	66.17	59.46
Normal GC	183.96	71.87	47.85	22.30

Table 4.1: Average times for all experiments

Memcached and database counts for all experiments are shown in Table 4.2. As we expected, Memcached misses and database queries have decreased for our generational caching implementation. However, Memcached gets and cache hit rate go down. This phenomenon is explained in Section 4.4.

Run	Memcached gets	Memcached misses	Cache hit rate	Queries
Teardown Control	97.93	5.33	0.9456	44.51
Teardown GC	37.52	4.99	0.8668	32.35
Normal Control	92.05	6.05	0.9342	45.95
Normal GC	38.33	5.43	0.8583	32.64

Table 4.2: Average counts for all experiments

Figures 4.3 and 4.4 show that average response time has decreased uniformly for our generational caching implementation. Likewise, Figures 4.5 and 4.6 show that HTML render time decreases quickly at the beginning and continues to decrease as time goes on. Equilibrium is reached much more quickly in the teardown traffic than the normal traffic. A similar phenomenon can be seen in Memcached miss count, shown in Figures 4.7 and 4.8. For the majority of the experiments, miss counts are remarkably similar between the control and our generational caching implementation. Response time, HTML render time, and cache miss count increase around hour 21 which is marked by "Start of cache expirations".

Figures 4.9 and 4.10 show the amount of space used in Memcached as reported by the `bytes` value in `Memcached::getExtendedStats()`[5]. Usage is significantly higher for generational caching than the control and continues to increase over time. The control appears to have reached an asymptote after 48 hours.

4.4 Analysis

In general, measured values reach equilibrium much more quickly during teardown traffic because over 30% of the requests are for the teardown, which gets cached very early on. The normal traffic experiments request a larger breadth of content so a smaller percentage of requested content is cached. We can see in Table 4.1 that the decreased response time is caused by the decreased database and HTML render times. In particular, the sum of the differences in database times and HTML render times is roughly equal to the difference in response time. Memcached time varies because different content is cached between the two strategies and there are different numbers of `get`s. Table 4.2 shows that the decrease in number of queries is the cause of the observed decrease in query time. The number of queries decreased because the requested data is being retrieved from the cache rather than the database.

Unfortunately, Memcached `get`s are very skewed between the two strategies because of *wiki text* rendering. The control experiment has a very large number of `get`s because of the Memcached `multiGet` call for rendered wiki text. Nearly all of these are cache hits which increases the average hit rate. Caching HTML in our generational caching implementation eliminates the need for caching wiki text rendering which is why the Memcached `get` count is so much lower. Similarly,

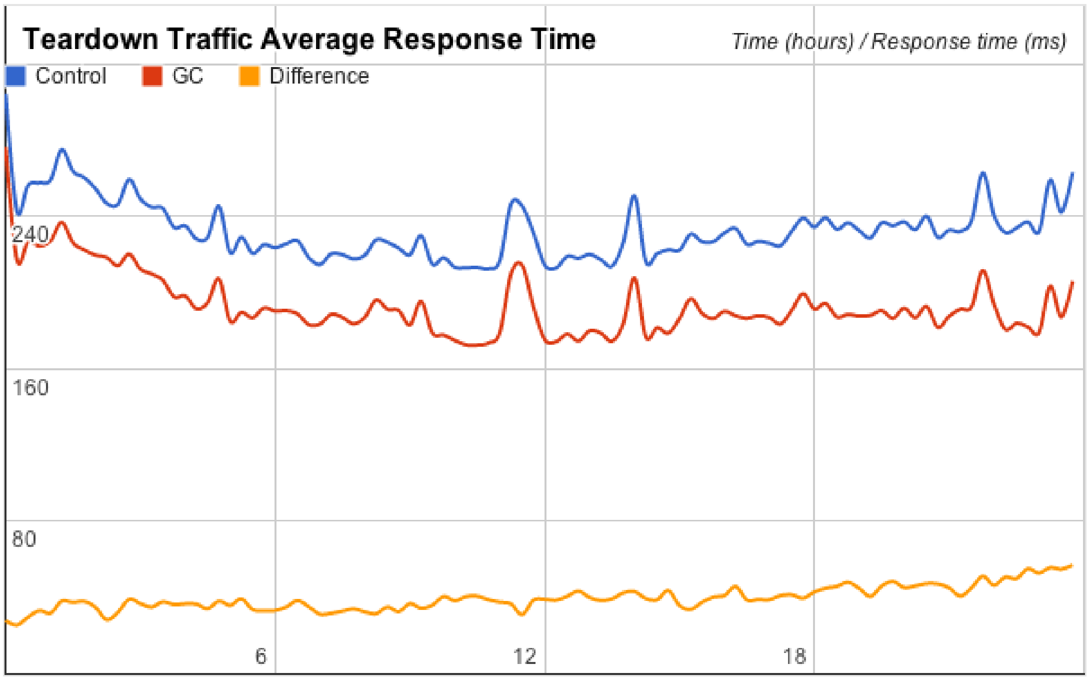


Figure 4.3: Teardown traffic response time

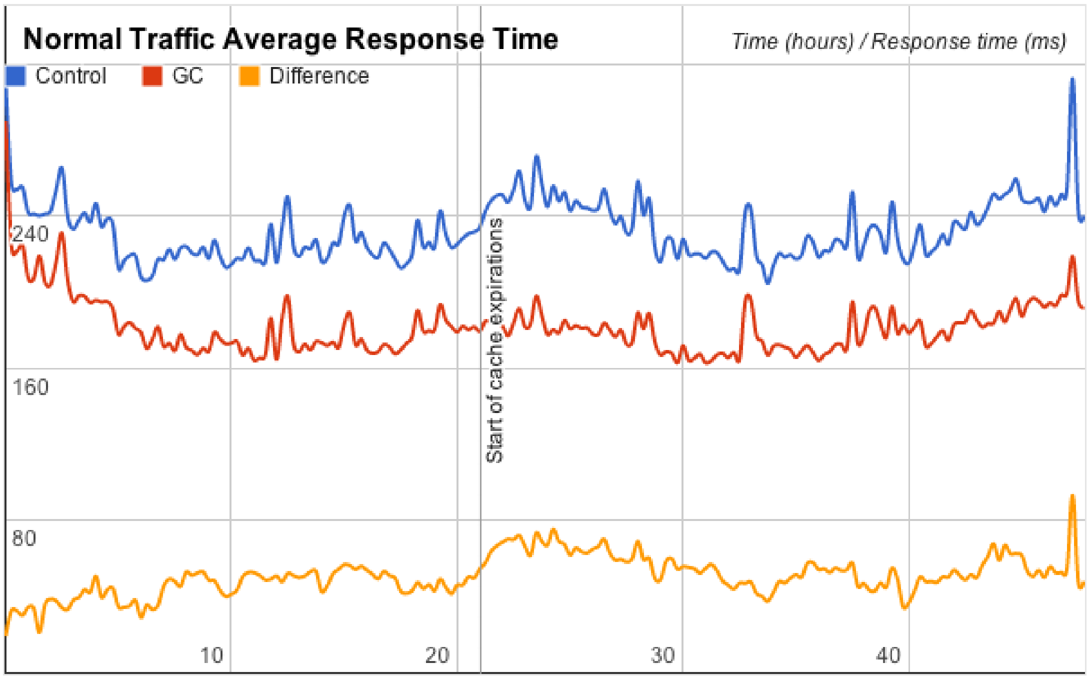


Figure 4.4: Normal traffic response time

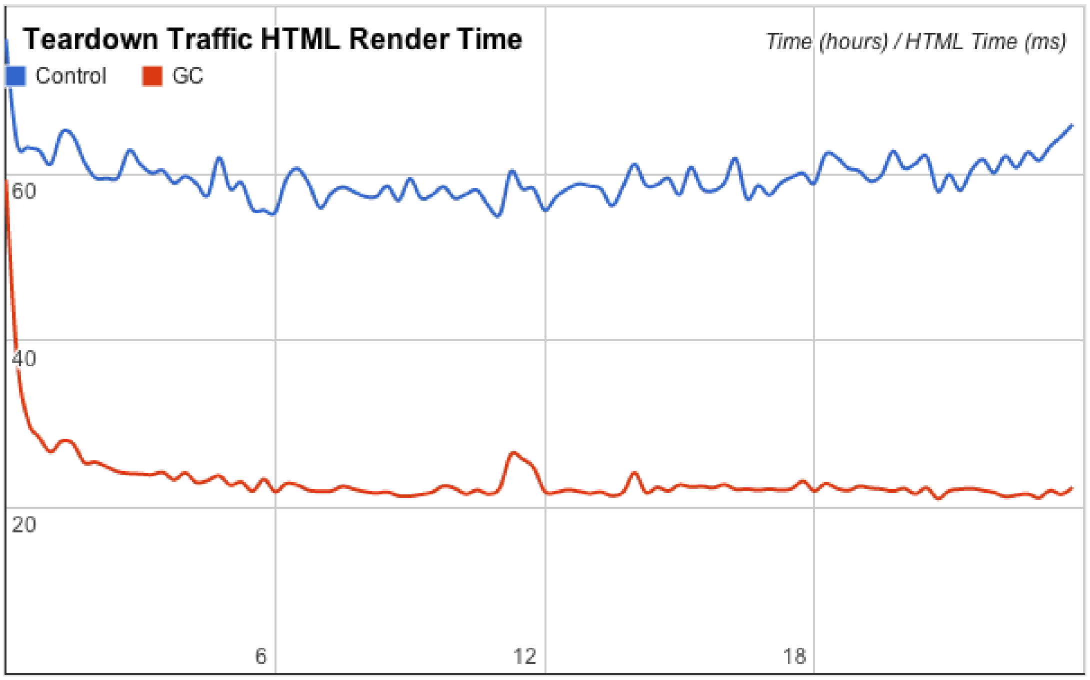


Figure 4.5: Teardown traffic HTML render time over time

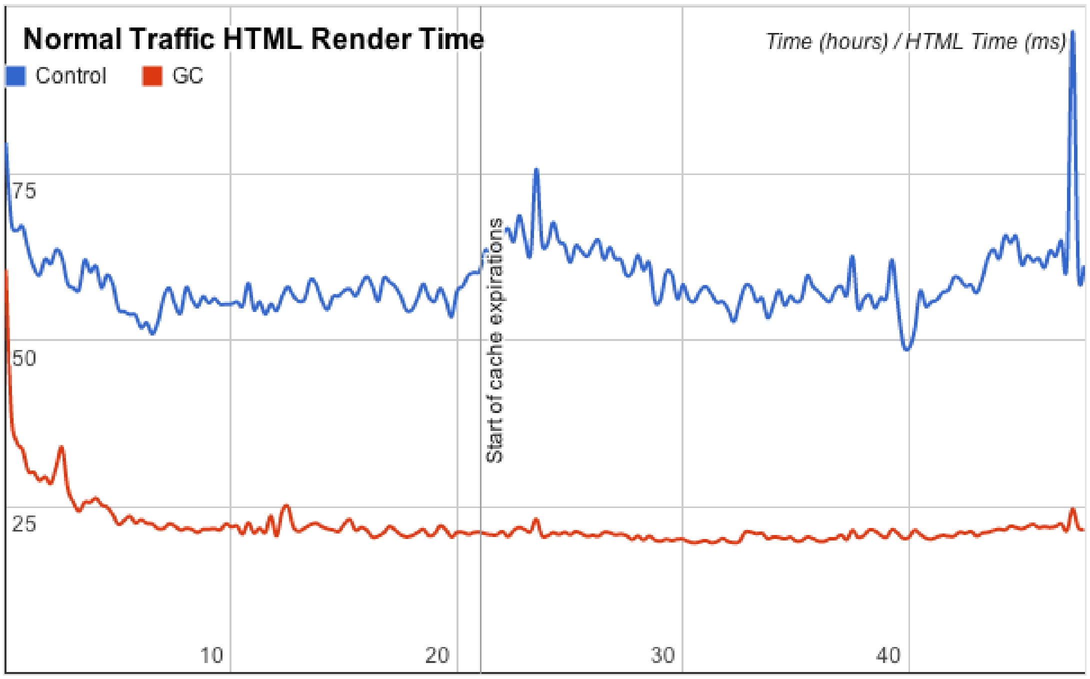


Figure 4.6: Normal traffic HTML render time over time

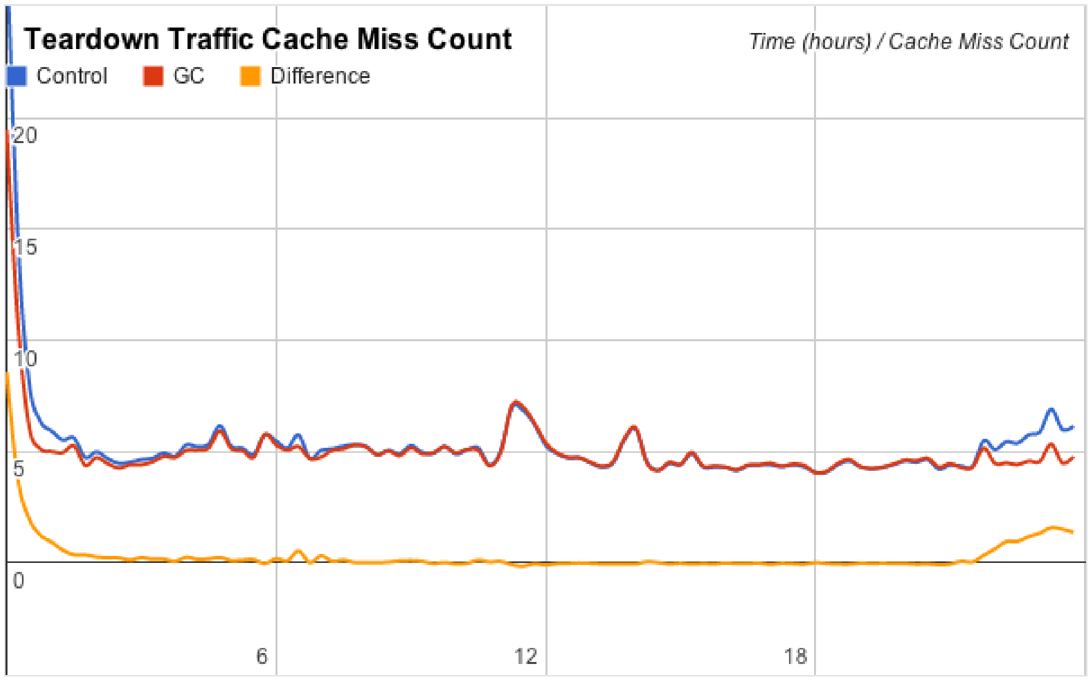


Figure 4.7: Teardown traffic cache miss count over time

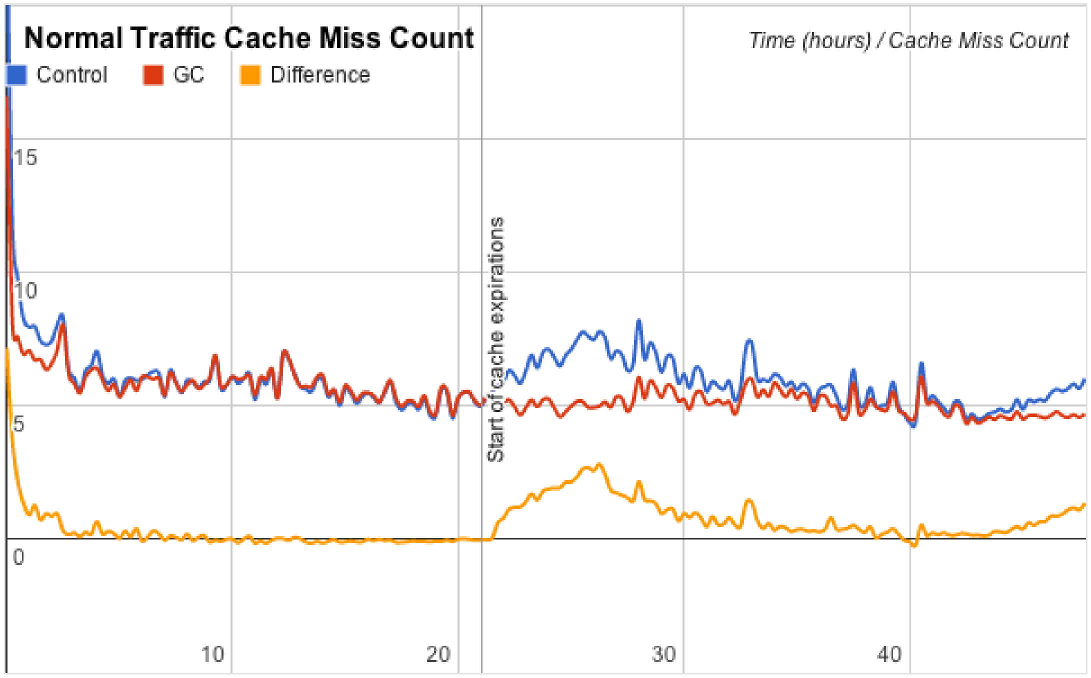


Figure 4.8: Normal traffic cache miss count over time

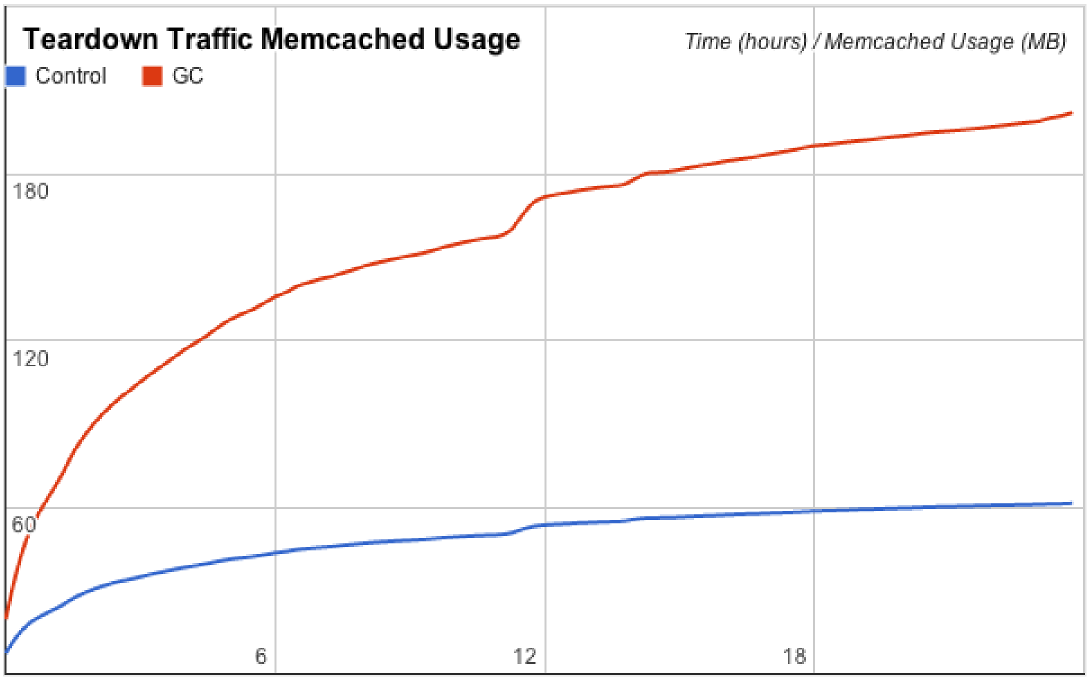


Figure 4.9: Teardown traffic Memcached usage over time

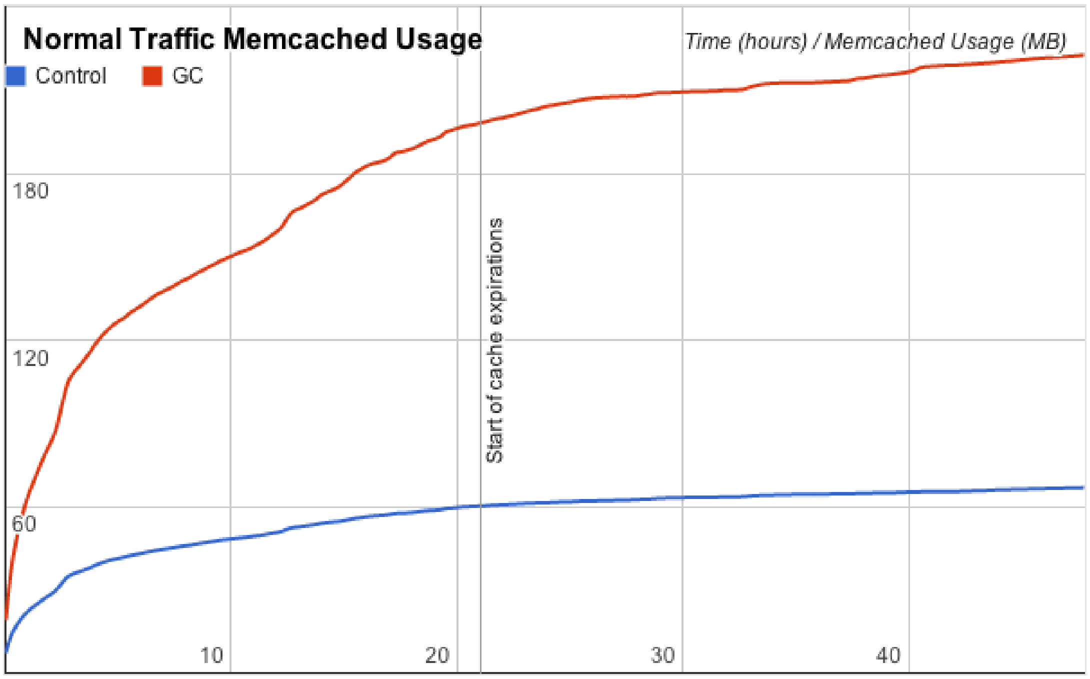


Figure 4.10: Normal traffic Memcached usage over time

cache hit rate decreases for the generational caching implementation because there are no longer a large number of cache hits for rendered wiki text.

Memcached miss counts remain at nearly the same values between the control and generational caching runs for most of the experiment because the requested content is either cached or uncached in both experiments. The first few hours demonstrate how requesting uncached content affects misses because Memcached is flushed immediately before the experiments. There are a large number of misses for the control because of the Memcached `multiGet` call for rendered wiki text. Our generational caching implementation also has a large number of misses because cache misses result in more cache `get`s for nested content which are also misses. They diverge around hour 22 because most guide cache times are set to expire after 24 hours. Varying the 24 hour cache times by `10%` results in cache entries will start to expire after about 22 hours. This doesn't occur in the generational caching implementation because cache entries aren't set to expire at all which results in a decreasing miss count over time.

Memcached usage is much higher for generational caching because much more content is being cached. Additionally, content isn't set to expire so modifying content results in more cache entries rather than new content replacing existing cache entries. There isn't a decrease in Memcached usage at the "Start of cache expirations" marker because expired cache entries aren't immediately removed. Expired cache entries are removed the next time they are referenced or if cache entries need to be removed because Memcached is filled to capacity and there isn't any space for new entries. For the control, any cache misses are immediately followed by cache `set`s so the expired items aren't observed in Figures 4.9 or 4.10.

4.4.1 Requirements

With these results, we have demonstrated that our generational caching solution meets the requirements outlined in Chapter 3. In particular:

R1 The caching solution is largely transparent to the developer. The developer uses the ORM object to construct cache keys and any cache entries using the object's cache key are invalidated when the underlying data changes.

R2 The caching strategy allows for finer-grained caching without adding complexity to cache invalidation. Since cache entries automatically expire anytime the underlying data changes, the number of cache entries dependent on a piece of data doesn't affect the complexity of cache invalidation.

R3 The caching solution allows for caching and invalidating fragments of HTML. This was done at a very granular level with guide steps which were invalidated as necessary.

R4 As we saw in Section 4.3, response time, number of database queries, and database query time is reduced using our caching solution.

Chapter 5

Conclusions and Future Work

Website scaling has been and will continue to be a major issue for small to large web companies. A common solution to this problem, like many other areas of Computer Science, is some form of caching. Values that are expensive to generate, such as rendered HTML and database queries, can be cached so they can be retrieved much faster in subsequent requests. However, ease-of-development, fine-grained cache entries, and cache invalidation are in direct conflict with each other. In particular, cache invalidation becomes more difficult as the number of cache entries that must be invalidated increases. Unfortunately, this burden is typically put on the application developer.

Our approach used generational caching, a technique that attempts to relieve the burden of cache invalidation even in situations with many dependent cache entries. The novel idea is that the cache key changes when the underlying data is updated, which is typically achieved by including the data's modified date in the cache key. All of the cache entries that depend on that data are effectively invalidated because they won't be referenced again. This results in the ability to perform fine-grained caching without forcing the developer to invalidate cache entries when data changes.

We implemented generational caching for iFixit's guides to find that this approach works remarkably well. In particular, HTML generation time, number of database queries, and average response time decreased by 60%, 27%, and 20%, respectively. Just as important, cache invalidation is transparent to the developer because the ORM handled the retrieval of data and generation of cache keys. We are eager to incorporate our generational caching solution into more of iFixit's code base to simplify the implementation of caching and improve performance.

5.1 Future Work

5.1.1 Cache Garbage and Memcached Evictions

The cache garbage that generational caching creates is largely a non-issue because the unreferenced cache entries will eventually be evicted based on the LRU policy. However, the least recently used cache entry could be valid and might be referenced again in the future even if there more recently used cache entries that are not valid. This could happen by viewing resource A and then making several edits to resource B. The several invalid cached versions for each edit of resource B are more recently used than the valid resource A cache. Our experiments did not run long enough to fill up Memcached and cause any cache entry evictions. Measuring changes in miss count when evictions occur would demonstrate whether or not the cache garbage generated by generational caching is an issue. Also, analyzing Memcached miss count across experiments with a varied amount of total memory allocated to Memcached for both caching strategies would show how much more Memcached space is necessary for a generational caching implementation.

5.1.2 Varied Workloads

Our experiments focused on replaying real world traffic to determine how the generational caching solution affects iFixit. Since iFixit's workload involves many more reads than writes, we didn't see how generational caching affects Memcached miss count in a write heavy workload. Generational caching provides many benefits for edits because of the finer-grained caching of ORM objects. Graphing Memcached miss count and page response time for varied percentages of reads and writes would be interesting. As the number of writes increases, generational caching will likely have better response times compared to iFixit's current implementation.

Bibliography

[1] 37signals: Web-based collaboration apps for small business. http://37signals.com.

[2] Amazon ec2 instance types. http://aws.amazon.com/ec2/instance-types/.

[3] ifixit: The free repair manual. http://www.ifixit.com.

[4] memcached - a distributed memory object caching system. http://memcached.org.

[5] Php: Memcache::getextendedstats - manual. http://www.php.net/manual/en/memcache.getextendedstats.php.

[6] Samsung galaxy s4 teardown - ifixit. http://www.ifixit.com/Teardown/Samsung+Galaxy+S4+Teardown/13947/1.

[7] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2(4):20:1–20:28, Oct. 2008.

[8] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th interna-*

tional conference on String processing and information retrieval, SPIRE'07, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 181–190, New York, NY, USA, 2010. ACM.

[10] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 294 –303 vol.1, mar 1999.

[11] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, K. Ramamritham, and D. Fishman. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 667–670, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[12] P. Gupta, N. Zeldovich, and S. Madden. A trigger-based middleware cache for orms. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, Middleware'11, pages 329–349, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] D. H. Hansson. How basecamp next got to be so damn fast without using much client-side ui. `http://37signals.com/svn/posts/3112-how-basecamp-next-got-to-be-so-damn-fast-without-using-much-client-side-ui`, 2012.

[14] D. H. Hansson. How key-based cache expiration works. http://37signals.com/svn/posts/3113-how-key-based-cache-expiration-works, 2012.

[15] A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and characterization of large-scale web server access patterns and performance. *World Wide Web*, 2(1-2):85–100, Jan. 1999.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[17] J. Kupferman. Web application caching strategies: Generational caching. http://www.regexprn.com/2011/06/web-application-caching-strategies_05.html, 2011.

[18] J. Kupferman. Web application caching strategies: Write-through caching. http://www.regexprn.com/2011/06/web-application-caching-strategies.html, 2011.

[19] I. Voras and M. Zagar. Web-enabling cache daemon for complex data. *CIT*, 16(4):317–323, 2008.