UNIVERSITY OF WATERLOO
Faculty of Engineering
Nanotechnology Engineering

Web Caching with Consistent Hashing

Tagged Inc.
San Francisco, CA

Prepared By:

Rajesh Kumar Swaminathan
ID #20194189

Userid rswamina
4A Nanotechnology
Confidential-1

rajesh@meetrajesh.com

December 26, 2009

Rajesh Kumar Swaminathan
#1205-9633 Manchester Drive
Burnaby, BC V3N 4Y9

December 26, 2009

Dr. Marios Ioannidis, Director
Nanotechnology Engineering
University of Waterloo
Waterloo, Ontario N2L 3B9

Dear Sir:

This report, entitled, "Web Caching with Consistent Hashing" is my fourth and final work term report (WKRPT 400). I completed this new report at the end of the second term of the double work term immediately following 3B, spanning the months of May 2009 to August 2009. This report was completed during my work term at Tagged Inc. in San Francisco, California. It is a confidential-1 report.

Tagged is a social networking company that runs and maintains tagged.com, America's 3rd most popular online social network. At Tagged, I was a member of the revenue projects team which was part of the engineering group. The revenue projects team is responsible for pushing out revenue-generating features in a timely and speedy fashion. I was also part of the photo review team that was responsible for designing and engineering a high-throughput photo review system that would enable a team of reviewers to review tens of thousands of user uploaded photos everyday.

The purpose of this report is to investigate the pros and cons of a particular type of caching strategy known as *consistent hashing* as it applies to the scalability of a massively trafficked website. This caching strategy is compared with the de facto caching solution known as *naive caching*. The idea for this topic was hinted to me by my supervisor Mr. Brent N. Francia, whom I would like to thank for proofreading my report. Brent was very interested in knowing the flexibility and scalability obtained by using *consistent* caching over the usual technique of *naive* caching that we had already been using.

The topic of this report proved itself to not only be insightful and exciting, but a topic that, I hope, will be of immense usefulness to the engineering group at Tagged, for who this report is written for. There are a lot of issues, benefits and outcomes to be discussed while implementing an easy-to-maintain cache that scales with the number of caching servers in use, and hopefully this report will be able to shed some light on the complexities involved.

I hereby confirm that I have received no further help other than what is mentioned above and in the references section in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Rajesh Swaminathan
ID 20194189

Contributions

Tagged Inc. is a social networking company that runs and maintains Tagged.com, America's 3rd most popular online social network. Tagged specializes in the social "discovery" aspect of social networking. Tagged itself is a relatively small team of about 45 employees. I was a member of the revenue projects team which was one of the many divisions of the engineering group. This team comprised of one full time engineer and three co-op students, all either students or alumni from the University of Waterloo. Two of these co-ops were in their second year while I was in my fourth year of our respective computer science and engineering programs.

The main goal of the revenue projects team was to push out revenue-generating features in a timely and speedy fashion. The quicker we pushed out features, the more we could monetize on them. My primary tasks as part of this group was to read and understand the specifications for new features, and then design the back end database tables and structure of the code such that all the requirements of the feature could be met. These tasks proved tricky as the structure of the code would have to be designed to allow for any possible future additions and improvements of the feature.

Every time our team came up with a design, we would call a design review meeting where key engineers at Tagged would critique the design and point out potential pitfalls. I had the opportunity to run one of these design review meetings myself which helped me gain immense experience in presentation skills, critical analysis, and traffic forecasting . In addition, I had to document the design on the company's internal wiki which helped me gain skills in organization, documentation, and clarity.

In addition to the design, I also wrote PHP code to implement the business logic of the feature. I also did some simple front-end work in HTML, CSS and JavaScript to specify the look and feel of the feature. I wrote email templates, and conducted a few A-B tests to find out whether certain changes to existing features would help increase the popularity of the feature. (See the glossary for definitions of terms and acronyms.)

Working at Tagged has been enormously enriching. I learned a great deal about building and managing large-scale websites that experience millions of hits a day. There were numerous real-world challenges that we had to tackle before we were ready to release a feature. I acquired a great deal of knowledge on social networking and how people react online. I definitely realized a very strong positive correlation between our users' engagement and the speed of our site. The faster our site, the more our users loved our site.

Two brand-new features I worked on this work term were "Top 8 Featured Users" and "VIP Subscriptions". The first feature was a leader board feature where members could pay to display themselves on the homepage for the purposes of self-promotion. The second feature was a VIP "package" that users of our website could subscribe to by paying a fixed monthly price. This package came bundled with a set of 5 features that other regular users could not take advantage of. Working on these features helped me learn about the various technologies and tools we use here at Tagged to help us build highly scalable websites.

A lot of these features that are written are often hit by millions of users each day. We have a very elaborate stack of servers and databases, but even still the amount of load we receive can be tremendous. Hence any kind of performance improvement is greatly helpful in reducing the load on our servers. The single-most important technique to reducing load is caching, which is the act of storing frequently-accessed items in a separate pool for quick fetching.

At Tagged, we make use of a very fast in-memory cache known as memcache which was already helping us reduce our load on the database servers vastly. It was clear that the more caching servers we had available, the more data we would be able to cache, and the faster the site would be. Thus we needed it to be very easy to increase the number of caching servers at any time. This is the primary connection between the report and my job. We already knew that the faster the website was, the more engaged our users were. So any performance improvement to our core caching strategy would be highly beneficial. The purpose of this report is to quantify the benefits in using more modern caching strategies and to analyze what trade-offs would have to be made when considering a switch from our existing vanilla caching strategy.

Writing this report provides a permanent record of my work term and helps me document my experience here at Tagged. It also helps me practice my skills of presentation and evaluation of a high-level caching mechanism. There are a lot of issues, benefits and outcomes to be discussed while implementing an efficient and effective caching strategy, and hopefully this report will be able to shed some light on the complexities involved.

In the broader scheme of things, it was straight-forward knowledge that users to our social networking website were attracted to a constant stream of new features. No one wanted to come back to a static, stale website. Hence the more features we designed for members to interact with other members, and the more engaging and responsive these features were, the more time the users would spend on our site and the more page views they would generate for us. Users would then spend more time and money on our site and click on ads. So the applications we built and the performance optimizations we recommended contributed directly to the bottom line of the company.

Executive Summary

The purpose of this report is to investigate the pros and cons of two caching strategies as it applies to the scalability of a massively trafficked website. The goal is to analyze the ease of implementation, flexibility, and performance gains of two types of caching mechanisms, namely *consistent* caching and *naive* caching.

The scope of the report is an audience that has a basic understanding of websites, web pages and standard web tools such as browsers, servers, and content and would like to learn how to enhance the performance of their web pages through caching.

What is caching? Caching is the act of saving the output of a complex and computationally expensive operation so it can be used readily the next time without being computed again. This saved output is known as the cache. Of course, since the cache is only a saved copy, it does not reflect reality, so if any of the data that the cache relied on were to change, the cache would have to be regenerated.

The pro to caching is that it lessens the burden on the web and database server, but the con is the extra overhead in maintaining and expiring the cache correctly. If the cache is not expired when it should be, we would end up showing stale content to the user, which is unacceptable. Furthermore, it should be easy to add/remove caching servers to the caching pool fairly easily without having to expire a significant portion of the cache. So the end goal of the report is to quantify the ease-of-use and performance improvements of implementing an efficient caching mechanism using a consistent hashing technique versus a simple, naive, and easy-to-implement vanilla caching technique. We analyze the two implementations' pros and cons in the light of relevant constraints.

The report is split into five sections. These five sections are Introduction, Requirements, Design, Analysis, and Conclusions.

The major points covered in this report are:

- What caching is and concepts behind caching
- Why caching is important for scalability

- Partitioning algorithms and their importance
- Virtual nodes and their usefulness
- Performance criteria and their weights for deciding between two or more caching strategies
- Performance analysis of two partitioning algorithms, namely consistent hashing and naive hashing
- Evaluation of these two caching strategies in light of the above performance criteria

The major conclusion of this report is that a caching strategy that uses consistent hashing is a significant improvement over naive hashing. However, the naive caching strategy is a lot simpler to implement and only requires one line of code. It therefore outperforms the consistent hashing strategy in terms of performance and ease of implementation. A simple simulation revealed that consistent caching was over 45% slower than naive caching. However, the naive caching strategy, despite its benefits, is extremely undesirable when it comes to flexibility, horizontal scalability, maintenance, and adding or removing servers from the cache pool.

The major recommendation of this report is to replace Tagged's naive caching strategy with a more scalable consistent hashing strategy. Although less performant, it is overall a better solution than the naive caching strategy given all the real-world constraints such as ease of implementation, flexibility, etc.

Conclusions

From the analysis in the report body, it was concluded that:

- Consistent caching is overall a significantly better solution than naive caching.
- The naive caching strategy is a lot simpler to implement and only requires one line of code. It therefore outperforms the consistent hashing strategy in terms of performance of code execution, and ease of implementation. A simple simulation revealed that consistent caching was over 45% slower than naive caching.
- The consistent hashing strategy, on the other hand, is significantly more flexible, easily maintainable, more horizontally scalable, and allows for the easy and relatively inexpensive addition of new servers to the cache pool as the demands of the cache increase.

Recommendations

Based on the analysis and conclusions in this report, the following recommendations are proposed.

1. Replace Tagged's naive caching strategy with a more scalable consistent hashing strategy since consistent hashing is a better strategy overall.
2. Allocate engineering human resources to investigate the effectiveness of adding replication, failure detection, fault-tolerance, and versioning to the consistent hashing solution strategy recommended above. This will increase the size of the cache, but will add to overall site stability and increased uptime which contributes directly to Tagged's bottom line.
3. Allocate engineering human resources to research into further optimizations within the consistent hash algorithm to make it run faster so that its poor execution performance does not become a bottleneck in the long run.

Table of Contents

List of Figures

List of Tables

1 Introduction

The purpose of this work term report is to investigate the pros and cons of two caching strategies as it applies to the scalability of a massively trafficked website. The goal is to analyze the ease of implementation, flexibility, and performance gains of 2 types of caching mechanisms, namely *consistent* caching and *naive* caching.

The scope of the report is an audience that has a basic understanding of websites, web pages and standard web tools such as browsers, servers, and content and would like to learn how to enhance the performance of their web pages through caching.

1.1 Caching

What is caching? In computer science, a cache is a collection of data duplicating original values stored elsewhere or computed earlier. This is usually done when the original data is expensive to fetch owing to longer access time (such as for databases stored on disk) or to compute due to computational complexity, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, it can be used in the future by accessing the cached copy rather than re-fetching or recomputing the original data. [1]

There are many different types and levels of caches, but because Tagged's product is a website, we are primarily concerned with *server-side caches.* Server-side caches are employed by web servers to store previous responses from the web servers or database servers, such as generated HTML or user data. Server-side caches reduce the amount of information that needs to be generated, as information previously stored in the cache can often be re-used. This reduces server load and processing requirements of the web and database servers, and helps improve responsiveness for users of the website in question.

The data in the backing store may be changed by entities other than the cache, in which case the copy in the cache may become *out-of-date* or *stale.* Alternatively, when the client updates the data in the cache, copies of that data in other caches will become stale.

1.2 Caching Concepts

There are three different but important aspects or mechanisms to be concerned about while implementing any kind of cache:

1. **Freshness:** Freshness specifies how long a cache is valid for. This mechanism allows a cache to be used by someone without re-checking it to see if it is still valid. It is assumed that if the freshness is specified as 5 minutes, the cache can be used for that amount of time without worrying about staleness. If the freshness is too low, then the effectiveness of the cache is reduced since we would need to re-retrieve the data again from the original source frequently. If the freshness is too high, we may end up displaying stale content to the user which in most cases is unacceptable. Usually, the tolerance level for stale caches depends on how stale the cache is. A cache of a user's profile information that is 1 minute stale may not be that bad, but if it is a couple of hours old, then the user will be left wondering why his profile information never got updated. In some cases, no amount of staleness is tolerable like a user's bank balance or the status of an online currency exchange transaction.
2. **Validation:** Validation is the process of checking whether a cached response is still good after it becomes stale. This step is usually not favored because the process of checking if a cache is valid is often as expensive as retrieving the data from the original store in the first place.
3. **Invalidation:** Invalidation is the process of marking a cached object as invalid or expired. It is usually the side effect of changing some piece of data that the cache relied on. So for example, if the HTML for the profile page for a user is cached, the user changing his/her date of birth would invalidate the cache since the user's age would now be different.

1.3 Caching at Tagged

At Tagged, we already use a simple in-memory caching mechanism known as memcache. In-memory access is much more faster than disk-based access and so this helps us reduce our load on the database servers vastly. Currently, we have a cluster of 128 memcache servers across which all cache data is near-uniformly distributed. It was clear that the more caching servers we had available at our disposal, the more data we would be able to cache, and the faster the site would be overall. Thus we needed it to be very easy to increase the number of caching servers at any time. Also, caching servers occasionally fail due to hardware problems, so our caching strategy had to be able to deal with such ad-hoc, unannounced failures gracefully without having a significant impact on the site's performance.

This is the primary motivation behind re-thinking the strategy behind distributed caching. In the current caching scheme, it is a very painful task to add or remove servers from the cache pool. Adding even one server to the pool requires a tremendous amount of data to be moved between servers. Thus, the purpose of this report is to quantify the benefits and trade offs of moving to a better caching model that allows us to add and remove servers easily, and minimize the amount of data that needs to be moved upon such infrastructural changes.

1.4 Why Cache?

These days, just making a website that runs is not sufficient. It is obvious that the longer the website is up, the more revenue-generating potential it has. Thus the goal is to keep the website running all 24 hours a day regardless of differences in user traffic patterns throughout the day.

Hence, the code that is written for the website not only has to work and be bug-free, but it also has to be scalable enough to handle thousands, sometimes millions of users. On top of that, it also needs to be fast for all these users because there is clear positive correlation between speed of the site and user engagement. [2] The faster a site is, the more actions the user can perform in a fixed amount of time.

What this means is that we as engineers have to make sure the code that we and other developers write runs as quickly as possible. Sometimes, throwing more hardware at the problem will help, but hardware is not always cheap and the goal is to solve the problem with the least monetary cost. This is where caching can come into play. Caching is a software solution that saves the company money by reducing the amount of hardware required to deliver the same number of pages. In the conventional webpage generation process illustrated in Figure 1, caching helps speed up the process by storing part of the content from the database (DB) in the web server's memory (RAM). Data in RAM can be accessed much more quickly than data in the database.

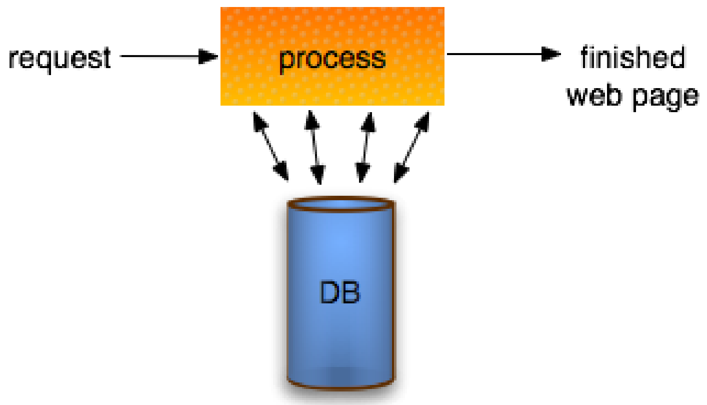

Figure 1: How a conventional dynamic web page is generated. Caching can help reduce the load on the database (DB) which is slow since it is disk-based. [3]

Unfortunately, caching and dynamic content generally do not work well together. However, when caching is used correctly, it can help solve many of the performance and scalability problems of a sluggish website. Caching is often times the first solution that software engineers implement to improve the performance of a website.

2 Requirements

Problem Definition: Design an efficient and scalable caching strategy for Tagged that meets the following requirements:

1. The cache must save all infrequently updated data such as a user's profile information.
2. The cache must be in the form of a distributed *dictionary*, otherwise known as a key-value store that stores pairs of keys and values like $(k_1, v_1), (k_2, v_2)$.
3. The cache must be stored across a cluster of computers, preferably in a way that makes it easy to manipulate the dictionary without having to think about the details of the cluster.
4. The cache must allow for new servers to be added to the cache pool without significantly affecting the performance and uptime of the cache.
5. The caching strategy must support the ability to tweak how frequently a caching server gets hits based on its capacity.
6. The cache must allow for replication and must degrade automatically and gracefully if one or more servers go down.
7. The cache must be efficient. This means that it must be easy to read and write to the cache. It must also be efficient to check if a particular key exists in the cache since this is a common operation.

2.1 Performance Criteria

The following criteria in Table 1 are to be used when deciding which caching mechanism is more suitable for implementation at Tagged:

Table 1: A list of performance criteria to be used when evaluating one or more caching mechanisms.

Criterion	**Weight**	**Description**
Ease of Implementation	15%	Work needed to introduce and implement the cache
Horizontal Scalability	50%	Ability to add/remove servers from the cache pool
New Server Load	20%	Ability to gradually add a new server to the cache pool
Performance Degradation	15%	Speed loss accumulated when working with the cache

3 Design

In order to address the detailed requirements specified in Section 2, we investigate the concepts and complexities involved in choosing the correct caching solution strategy.

3.1 Partitioning Algorithms

One of the key design requirements for caching at Tagged is that it must scale incrementally. This requires a mechanism to dynamically partition the data fairly uniformly over the set of nodes (i.e., cache servers) in the system. What this means is that we need to know where each key in the cache will live before the key is stored or retrieved from the cache.

There are basically two mechanisms to assign each possible key to a server. These two mechanisms are detailed in the following two subsections.

3.1.1 Naive Hashing

The naive partitioning algorithm is very easy to implement and quick to evaluate. Suppose we have a cluster of n computers. We number the computers $0, 1, 2, \ldots, n-1$, and then store the key-value pair (k, v) on computer number $\text{hash}(k) \bmod n$, where $\text{hash}(\cdot)$ is any hash function that converts key k, an arbitrary string, to a non-negative integer. If we are using a good hash function, then $\text{hash}(k) \bmod n$ is uniform across $0, \ldots, n-1$ for any reasonable distribution of keys. This ensures our distributed dictionary is spread evenly across all the computers in our cluster, and does not build up too much on any individual computer.

The code to implement naive hashing is quite simple:

```
public function get_server(string $key) {
    // 1. store a list of the IPs of the 24 cache servers available to us
    $servers = array('232.145.26.0', '232.145.26.1', '232.145.26.2',
                     '232.145.26.3', '232.145.26.4', '232.145.26.5',
                     '232.145.26.6', '232.145.26.7', '232.145.26.8',
                    '232.145.26.9', '232.145.26.10', '232.145.26.11',
                    '232.145.26.12', '232.145.26.13', '232.145.26.14',
                    '232.145.26.15', '232.145.26.16', '232.145.26.17',
                    '232.145.26.18', '232.145.26.19', '232.145.26.20',
                    '232.145.26.21', '232.145.26.22', '232.145.26.23');
    // 2. hash the key using the cyclic redundancy checksum polynomial
    $hash = (crc32($key) >> 16) & 0x7fff;
    // 3. count the number of servers available to us
    $num_servers = count($servers);
    // 4. now pick the server based on the hash value
    return $servers[$hash % $num_servers];
}
```

A variant on this naive partitioning scheme was described in [4]. For example, given a set of 23 caches numbered $0, \ldots, 22$, we might hash key u to cache server number $h(u) = 7u + 4 \bmod 23$. 7 and 4 are arbitrary constants that remain unchanged once chosen. This partitioning scheme is quite similar to the one described above and provides no obvious advantage.

Naive hash-based distributed dictionaries are simple, but they have serious limitations. For example, say we are using a cluster of computers to crawl the web. We store the results of the crawl in a distributed dictionary. But as the size of the crawl grows, we will want to add machines to our cache cluster. Suppose we add even just a single machine. Instead of computing hash(k) mod n we are now computing hash(k) mod $(n+1)$. The result is that each key-value pair will get reallocated completely randomly across the cluster. The result is that we will need to move a fraction $n/(n+1)$ of our data to the new machines. For large n, this fraction approaches 1, so we will essentially need to move nearly all data to different servers.

This data-moving process will be slow, and might potentially be expensive. During the time when data is being moved to their new servers, the cache will be unavailable. It will be as if the cache suddenly disappeared. The databases will experience an immediate spike in traffic as practically everything has to be pulled out of the DB and stuck back into the cache. This is also potentially inconvenient if the site is active and requests are being served every second. The high amount of load on the databases could make them crash bringing the whole site down.

Similar problems arise if we add a larger block of machines to the cluster, or if we lose some machines due to failure.

The traditional solution to this problem is that the system administrator waits until a time at night when site traffic is not as high, usually around 3 AM, and adds new servers at this time. This is the only time when the site can handle an invalidated cache. However, this would severely affect users in other timezones around the world such as Europe and Asia. For a mission critical site like a banking application, this trade off may be unacceptable.

The alternate solution to the above problem is to dynamically change n over a period of time. For read operations, we check if the requested key exists on server $\text{hash}(k) \bmod n$ first. If it does, then we serve it from there. If it does not, then we check if the same key exists on $\text{hash}(k) \bmod (n+1)$. If it does, then we serve it from there. If not, we fetch the key by querying the database, and store it on server $\text{hash}(k) \bmod (n+1)$. This will slowly migrate the data to the new servers without causing too many cache misses.

3.1.2 Consistent Hashing

An alternate partitioning scheme to decide which keys live on which server is known as consistent hashing. This scheme allows us to distribute the caching load across multiple storage hosts in quite a clever manner. In consistent hashing, the output range of the hash function is treated as a fixed circular space or "ring". This means that the largest hash value wraps around to the smallest hash value. Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position. If that server is down, we go to the next one, and so on. Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that the departure or arrival of a node only affects its immediate neighbors while other nodes remain unaffected.

Unlike naive hashing, consistent hashing requires only a relatively small amount of data to be moved: if we add a machine to the cluster, only the data that needs to live on that machine is moved there; all the other data stays where it is. This is the *fundamental* advantage of consistent hashing over naive hashing.

To understand how consistent hashing works, we imagine wrapping the unit interval $[0, 1)$ onto a unit circle. Suppose we number the n cache servers as $0, \ldots, n-1$. If the hash function has range $[0, R)$, then we rescale the hash function via $x \rightarrow \text{hash}(x)/R$, so that the hash function maps onto the range $[0, 1)$, i.e., effectively onto the unit circle. This idea is depicted diagrammatically in Figure 2.

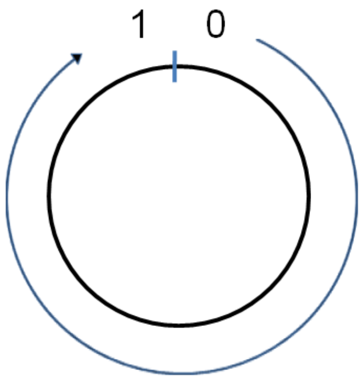

Figure 2: The output values of the hash function normalized to a unit interval $[0, 1)$ onto a unit circle. [5]

Then we can hash machine number j to a point hash(j) on the circle, for each machine in the range j $= 0, 1, \ldots, n-1$. Figure 3 shows what the ring might look like for an $n = 3$ machine cluster.

The points will therefore be randomly distributed around the circle. Or we can force the nodes to be uniformly spread out around the ring by pinning each node's position on the unit ring. Now suppose we have a key-value pair we want to store in the distributed dictionary. We simply hash the key onto the circle, and then store the key-value pair on the first machine that appears clockwise of the key's hash point. For example, for the key shown in Figure 4, the key-value pair is stored on machine number 1.

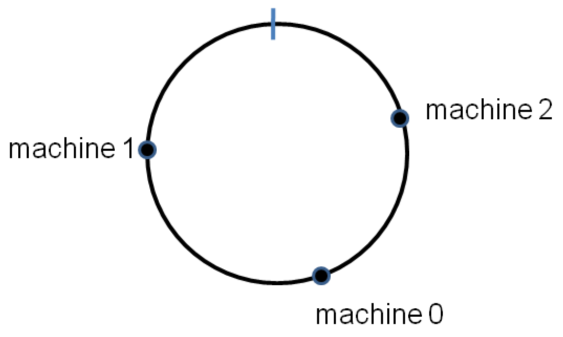

Figure 3: How a consistent hashing ring might look like for an $n = 3$ machine cluster. [5]

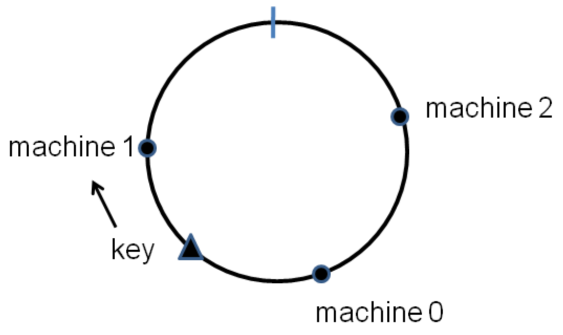

Figure 4: The key-value pair is stored on the first machine that appears clockwise of the key's hash point. In this example, the key is stored on machine 1. [5]

Because of the uniformity of the hash function, a fraction roughly $1/n$ of the key-value pairs will get stored on any single machine.

Now if we add an extra machine, machine 3, to the cluster, it goes to the point hash(n) as shown in Figure 5. In doing so, most of the key-value pairs on the ring are completely unaffected by this change. But we can see that some of the key-value pairs that were formerly stored on machine 1 (including our example key-value pair) will need to be moved to the new machine, i.e. machine 3. But the fraction that needs to be moved will typically be $1/(n + 1)$ of the total, a much smaller fraction than was the case for naive hashing which was $n/(n + 1)$.

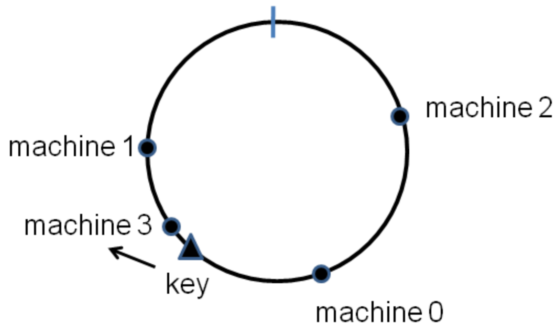

Figure 5: When a new server is added to the ring, the key now gets allocated to machine 3 instead of machine 1. [5]

This basic consistent hashing algorithm introduced above presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. This can be easily solved by positioning the nodes uniformly around the ring such that each node handles exactly $1/n$ of the load. Second, the basic algorithm described in the above paragraph is oblivious to the heterogeneity in the performance of nodes. All nodes will experience similar loads regardless of their capabilities and capacities. Third, when a machine is added to the cluster, all the keys redistributed to that machine come from just one other machine. Ideally, the keys would come in a more balanced way from several other machines. These issues are addressed in the following section on virtual nodes.

3.2 Virtual Nodes

To address the issues on consistent hashing outlined in the previous section, we may use a variant of consistent hashing (similar to the one used in [6] and [7]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. These *virtual nodes* look like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions called *tokens* in the ring.

These virtual nodes are like replicas of the server. By adding these replicas, we increase both the uniformity with which key-value pairs are mapped to machines, and also ensure that when a machine is added to the cluster, a smaller number of keys are redistributed to that machine from each of the other machines in the cluster.

Using virtual nodes has the following advantages [8]:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes. Of course, this will not help if all the other servers are already running at maximum capacity.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible for can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure. More points for a server means it covers more of the ring and is more likely to get more resources assigned to it.

Virtual nodes in the form of node replicas also help reduce the spread in the distribution of keys across cache servers. In one particular simulation described in [9], it was shown that the spread decreases log-linearly with the number of replicas used. In Figure 6, the x-axis is the number of replicas of cache points on a log scale. When it is small, we see that the distribution of objects across caches is unbalanced, since the standard deviation as a percentage of the mean number of objects per cache (on the y-axis, also logarithmic) is high. As the number of replicas increases, the distribution of objects becomes more balanced. This experiment shows that a figure of one or two hundred replicas achieves an acceptable balance — a standard deviation that is roughly between 5% and 10% of the mean.

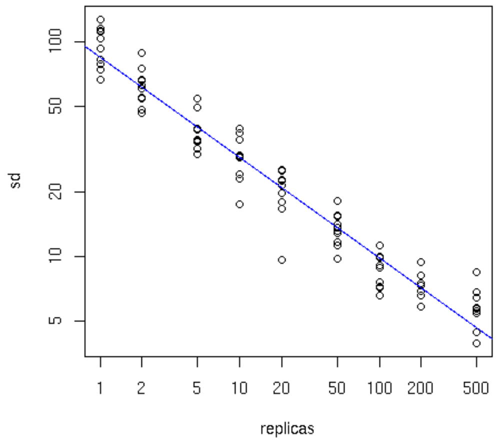

Figure 6: As the number of replicas is increased, the spread of keys across cache servers decreases. A figure of one or two hundred replicas achieves an acceptable balance of a standard deviation between 5%-10%. [9]

3.3 Variable Capacities

If different servers have different storage capacities or different performance characteristics, we would want to spread the load on the cluster across the servers in a corresponding manner. This is easily achieved with consistent hashing and virtual nodes: the number of virtual nodes that a node is responsible for can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

This technique also allows us to avoid dumping $1/n$ of the total load on a new server as soon as we turn it on and add it to the cluster. We would like for new servers to be slowly added to the cluster, so that the load on them increases gradually. This can be easily achieved by starting with one virtual node on the ring for this server, and then increasing the number of virtual nodes slowly to the desired number over the period of an hour or so. This is particularly important for caching services that are disk bound and need time for the kernel to fill up its caches. Such “gradual ramp-ups” avert a common problem in software and hardware engineering known as the Thundering Herd Problem [10].

3.4 Consistent Hash Implementation

A basic implementation of consistent hashing with a fixed number of replicas per node is shown below in the PHP programming language.

```
class cons_hash_ring {
    private $_num_servers;
    private $_num_replicas;
    private $_ring;
    private $_max_hash = ˜PHP_INT_MAX;
    private $_min_hash = PHP_INT_MAX;
    public function __construct($num_servers, $num_replicas) {
        $this->_num_servers = $num_servers;
        $this->_num_replicas = $num_replicas;
        $this->_ring = array();
        $hash = 0;
        // initialize the ring
        for ($i=0; $i < $num_servers; $i++) {
            for ($j=0; $j < $num_replicas; $j++) {
                $this->_ring[] = array($i, $j, $hash);
                if ($hash > $this->_max_hash) {
                    $this->_max_hash = $hash;
                }
                if ($hash < $this->_min_hash) {
                    $this->_min_hash = $hash;
                }
                $hash += (1 / ($num_servers * $num_replicas));
            }
        }
        // sort the ring
        usort($this->_ring, function($a, $b) {
                return strcmp($a[2], $b[2]);
            });
    }
    public function get_server($key) {
        $hash = ((crc32($key) & 0x7fffffff) % 1000000) / 1000000;
        // edge case where we cycle past the hash value of 1 and back to 0.
        if ($hash >= $this->_max_hash) {
            return 0;
        }
        // find the next node in the clockwise direction
        foreach ($this->_ring as $node) {
            if ($hash < $node[2]) {
                return $node[0];
            }
        }
    }
}
```

The above class represents a consistent hash ring which is stored as a sorted map of integers. The class accepts two parameters: the number of physical servers available, and the number of replicas to create for each node.

A user of this ring might use the above class to choose a server for a given key as shown below.

```
// store a list of the IPs of the 24 cache servers available to us
$servers = array('232.145.26.0', '232.145.26.1', '232.145.26.2',
                 '232.145.26.3', '232.145.26.4', '232.145.26.5',
                 '232.145.26.6', '232.145.26.7', '232.145.26.8',
                 '232.145.26.9', '232.145.26.10', '232.145.26.11',
                 '232.145.26.12', '232.145.26.13', '232.145.26.14',
                 '232.145.26.15', '232.145.26.16', '232.145.26.17',
                 '232.145.26.18', '232.145.26.19', '232.145.26.20',
                 '232.145.26.21', '232.145.26.22', '232.145.26.23');
$cache = new cons_hash_ring($num_servers=count($servers), $num_replicas=3);
$server = $servers[$cache->get_server($key)];
```

There are of course more applications to this consistent hashing idea than just simple caching. Consistent hashing is a powerful idea for anyone building services that have to scale across a group of computers.

3.5 Replication

To address the problem of dealing with failed nodes, we replicate the cached data on $n-1$ additional hosts as shown in Figure 7. In this example $n = 3$ which means each key is stored on 3 servers and therefore replicated on $3 - 1 = 2$ additional servers. Replication ensures high availability and durability against individual node failures.

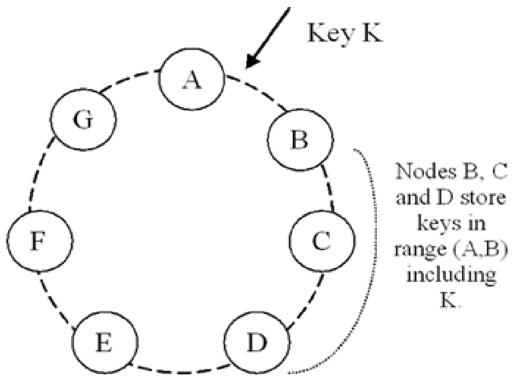

Figure 7: If we replicate the data stored in node B in nodes C and D, then we can spread the load on node B across nodes C and D. Also, if node B were to fail, we can also access the data from nodes C and D. [8]

In this replication scheme, nodes B, C, and D store keys in the range (A,B] including the example key shown in Figure 7, i.e. Key K. Node B is responsible for replicating its data over to the $n-1$ clockwise successor nodes in the ring, i.e. nodes C and D in this example. This results in a system where each node is responsible for the region of the ring between it and its n^{th} predecessor. In Figure 7, node B replicates the key k at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

What this allows us to do is to spread the load on node B across nodes C and D. Also, if node B were to fail for some reason or needs to be taken down for maintenance, we can always access the data from nodes C and D. Consequently, this replication scheme ensures high availability of the cache.

4 Analysis

4.1 Possible Solutions

As discussed above in Section 3, two possible solutions exist when it comes to choosing a partitioning strategy.

1. Implement a naive caching strategy. What this entails is that the IP addresses of the list of caching servers is put in an array, and we pick one server by hashing the key to an integer and taking the modulo of that integer with respect to the number of servers in the cache pool. This type of caching was explained in detail in Section 3.1.1.
2. Implement a more intelligent consistent hashing strategy. This strategy treats the cache pool as a "ring" and keys are assigned to servers based on the relative positions of the servers and keys on this ring. This type of caching was explained in detail in Section 3.1.2.

The above two solutions will be evaluated against the performance criteria specified in Table 1 in Section 2.1.

4.2 Solution Comparison

4.2.1 Data Migration Requirements

In using a naive caching partitioning strategy, the result we saw is that we will need to move a fraction $n/(n+1)$ of our data to the new machines upon addition of one new server. For large n, this fraction approaches 1, so we will essentially need to move nearly all data to different servers. This process will be slow, and might potentially be very expensive. During the time that all data is moved to their new servers, the cache will be unavailable, and will slow down the site significantly.

On the other hand, in using a consistent hashing partitioning scheme, the fraction of data that needs to be moved will typically be $1/(n+1)$ of the total, a much smaller fraction than was the case for naive hashing.

Thus the fraction of data that needs to be moved in naive caching is larger than the fraction of data that needs to be moved in consistent caching by a factor of n.

The percentage of data that needs to be moved as a function of the number of servers in the cache pool upon the addition of one new server to the pool is shown in Figure 8 for both naive and consistent hashing schemes. Even for a modest number of 10 servers, the difference is significant: 90.9% of the data needs to be moved for naive caching, whereas only 9.1% of the data needs to be moved for consistent hashing, an improvement of around 82%!

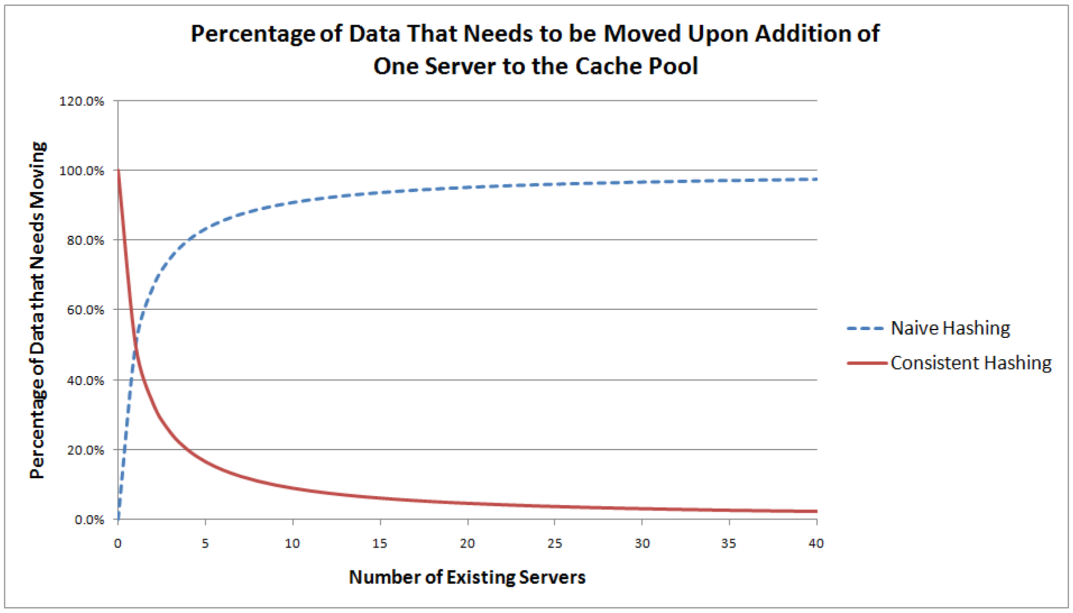

Figure 8: Consistent hashing outperforms naive hashing in terms of percentage of data that needs to be moved upon addition of one server to the cache pool. Chart generated with Microsoft Excel.

Furthermore, the more number of servers we have, the *less* data we need to move when using consistent hashing. This is an important advantage over naive hashing where the amount of data that needs to be moved *increases* with increasing number of servers.

Thus for anything more than 2 servers, consistent hashing is better than naive hashing if we seek to move as little data as possible around upon addition of new servers.

4.2.2 Node Failure Tolerance

Naive hashing cannot handle the death of a server until an admin explicitly removes it from the pool. Until then, the system will keep returning a cache miss and will go to the database to fetch the content. When this content is fetched from the database, there will be no place to put this key back into cache since the server assigned to that key is down, requiring the key to fetched from the database each time, over and over again.

However, with consistent hashing, the system will automatically start using the next server going clockwise in the ring without interference from the system administrator.

4.2.3 Code Execution Time

It can be seen from the implementation of the two partitioning algorithms, namely naive and consistent hashing that naive hashing will be faster to execute and more performant than consistent hashing. In fact, naive hashing is a constant time, i.e. $\Theta(1)$ algorithm with respect to the number of servers in the cache pool. What this means is that the algorithm will take the same amount of time to determine which server to use for a given key regardless of the number of servers in the pool.

Consistent hashing, on the other hand, is a linear-time, i.e. $\Theta(n)$ algorithm, where n is the number of servers in the cache pool. What this means is that the run time of the consistent hash algorithm to determine which key should be paired up with which server scales linearly proportionally to the number of servers in the cache pool.

This difference in execution time is an important consideration in the decision-making process since the algorithm is run very frequently: it is run each time a key is set or fetched from the cache.

To quantify the difference in execution time, the two algorithms were run 10,000 times on a consumer-grade Fujitsu laptop with a 1.6 GHz Intel Core 2 Duo 32-bit CPU with 2 gigabytes of RAM. The PHP version used was 5.2.4. Three trials were run for each hashing scenario and averaged. Three scenarios were studied: a) simple vanilla naive caching, b) consistent hashing with 24 servers and 0 replicas, and c) consistent hashing with 24 servers and 3 replicas for each server resulting in $24 \times 3 = 72$ nodes in total. The results are summarized in Table 2.

Table 2: Benchmark of run times of naive and consistent hashing strategies. All times are in microseconds and averaged over 10,000 iterations. Tests were run on a Fujitsu laptop with a 1.6 GHz Intel Core 2 Duo 32-bit CPU with 2 gigabytes of RAM.

Trial	**Naive Hashing (μs)**	**Consistent Hashing (μs) [No Replicas]**	**% Diff with Naive**	**Consistent Hashing (μs) [3 Replicas]**	**% Diff with Naive**
1	40.7	45.7	12.3 %	61.1	50.1%
2	43.3	44.3	2.3%	62.0	43.2%
3	41.6	47.3	13.7 %	59.3	42.5%
Avg.	41.9	45.8	9.3%	60.8	45.2%

From the above table, it can be seen that the consistent hashing algorithm with no replicas is 9.3% slower to execute than simple, naive hashing, whereas consistent hashing with 3 replicas is 45.2% slower than simple, naive hashing. Thus the performance degradation involved in choosing consistent hashing over naive hashing can be quite significant and must be paid close attention to.

4.3 Computational Decision Chart

In the following table (Table 3), each caching mechanism is scored, on a scale from 1-10, against the criteria in Table 1. The higher the number, the stronger the caching mechanism scores for that criterion. The score is then weighted against the importance of the criterion presented in Table 1, and then summed to provide an overall score for that caching mechanism.

For example, the naive caching mechanism has an ease of implementation score of 8.0, but that criterion is only 15% important, so the adjusted score is $8.0 \times 15\% = 1.2$.

As we see in Table 3, the consistent hashing mechanism is superior than the naive caching mechanism since it has a higher overall score. We thus conclude that Tagged should go ahead with the implementation of a consistent hash mechanism rather than its current naive caching mechanism.

Table 3: A computational decision chart that assigns a quantitative score to each of the decision-making criteria specified in Table 1. The higher the number, the stronger the caching mechanism scores for that criterion.

Criterion	**Weight**	**Naive Caching**		**Consistent Hashing**	
		Score	Value	Score	Value
Ease of Impl.	15%	8.0	1.2	4.0	0.60
Horz. Scalability	50%	2.0	1.0	9.0	4.50
New Server Load	20%	6.5	1.3	9.5	1.90
Perf. Degradation	15%	9.5	1.4	3.0	0.45
Sum			4.9		7.45

However, we also saw that the naive caching strategy outdoes the consistent caching strategy in execution performance by over 45%. For a large number of servers, i.e. over 500, this could be an important consideration. At Tagged, we only have 128 cache servers, so this consideration is not as significant. We should nonetheless investigate further optimizations within the consistent hash algorithm to make it run faster so that its $\Theta(n)$ performance does not become a bottleneck in the long run.

We should also investigate other improvements to the consistent hashing strategy that makes the system more robust to failure. We would need to investigate the effectiveness of adding replication, failure detection, fault-tolerance, and versioning to the consistent hashing solution strategy. This will increase the size of the cache, but will add to overall site stability and increased uptime which contributes directly to Tagged's bottom line.

5 Concluding Summary

In this report, we introduced the major concepts behind caching, server-side caching and distributed dictionaries. We explained what caching is and outlined the three most important aspects of caching, namely freshness, validation and invalidation. We looked at the how caching was already being implemented at Tagged and why caching was necessary for site scalability.

Since caching of database results was absolutely critical to scale our web pages at Tagged, we specified some minimum requirements for an efficient caching implementation. We specified the criteria that were most crucial to us in deciding which caching mechanism to implement and then weighted these criteria to distinguish between their importance. We then proceeded to outline a detailed design for a caching strategy at Tagged. Two key cache partitioning strategies were discussed in detail, namely naive caching and consistent hashing. We saw that the most effective caching strategy was the one that allowed us to add or remove servers from the pool without invalidating too much of the cache. This requirement was referred to as *horizontal scalability*.

We then specified a simple algorithm to implement consistent hashing at the partitioning layer. This algorithm helps us decide which key should live on which server. It also decides which additional secondary and tertiary servers the data might be available on. We provided a basic implementation in the PHP programming language. We also saw that consistent hashing with just 24 servers and 3 replicas was already over 45% slower than simple, naive hashing.

The performance improvements of two caching strategies were obtained and studied in light of the criteria specified in Table 1. We saw that the naive caching strategy was significantly easier to code-up and implement since it was only a few lines. However, ease of implementation was only one aspect of our decision-making criteria that accounted for only 15% of the effectiveness of the entire solution. From the computational decision chart in Table 3, we saw that consistent hashing, although less efficient and more complex than naive hashing, was an overall better solution in terms of horizontal scalability and ease with which new servers could be added to the pool of cache servers. This consistent hashing solution also outperformed the naive caching solution in the other important criteria of *new server load*. This means consistent hashing gives us the ability to introduce a server to the pool gradually, so it does not get hit by a stampede of requests as soon as it is turned on.

We eventually concluded that the consistent hashing mechanism was superior since it was a better solution overall.

Finally, we recommended future investigation of the effectiveness of adding replication, failure detection, fault-tolerance, and versioning to the consistent hashing strategy outlined in the sections above. This will increase the size of the cache, but will add to overall site stability and increased uptime, which contributes directly to Tagged's bottom line.

6 Glossary

1. **A-B Testing** is a method of testing where two different versions of the same feature are shown to users in order to determine which version performs better.
2. **CSS** stands for cascading style sheets. CSS is a way of specifying the look and feel of a website by defining margins, padding, borders, colors, etc.
3. **DB** stands for database. It is a store for data from where data can be easily queried and retrieved.
4. **HTML** stands for Hypertext Markup Language. It is a way of specifying the structure of a web page and the various elements contained in it such as paragraphs, headings, images, hyperlinks, tables, etc.
5. **IP** stands for internet protocol. It is the unique address given to each server on the network.
6. **JavaScript** is a scripting language that can be executed within the browser that helps us, among other things, to dynamically modify the structure of a web page by injecting content into it.
7. **PHP** is a scripting language that enables dynamic generation of a page based on the user who is logged in and his/her preferences. It also helps us interact with the database to store/retrieve user data.
8. **RAM** stands for random access memory. It is a temporary storage mechanism for computers and is very fast compared to storing things in a more permanent location such as the hard-drive. Its main drawback is that RAM is volatile so once the server is restarted, its contents are obliterated.

References

[1] "Cache", *Wikipedia, The Free Encyclopedia.*, http://en.wikipedia.org/w/index.php?title=Cache&oldid=333941188. Retrieved Dec 25 2009.

[2] Jakob Neilsen, "Feature Richness and User Engagement", *Jakob Nielsen's Alertbox*, http://www.useit.com/alertbox/features.html. Retrieved Dec 25 2009.

[3] Frank Talk, "BFT: Our case for 100% de-normalization", *Franks Talk's Personal Blog*, http://www.frankf.us/wp/?p=25. Retrieved Dec 26 2009.

[4] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y. 1999. "Web caching with consistent hashing". *Computer Networks* **31** (11): pp. 1203-1213.

[5] Michael Nielsen, "Consistent Hashing", *A Series on the Google Technology Stack*, http://michaelnielsen.org/blog/consistent-hashing/. Retrieved Dec 25 2009.

[6] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. "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* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[7] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[8] Werner Vogels, "Amazon's Dynamo", *Werner Vogels' weblog on building scalable and robust distributed systems*, http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html. Retrieved Dec 25 2009.

[9] "Consistent Hashing Simulation Results", *Tom White's Blog*, http://www.lexemetech.com/2007/11/consistent-hashing.html. Retrieved Dec 25 2009.

[10] "Thundering Herd Problem", *Wikipedia, The Free Encyclopedia.*, http://en.wikipedia.org/wiki/Thundering_herd_problem. Retrieved Dec 25 2009.