UNIVERSITY OF WATERLOO
Faculty of Engineering
Nanotechnology Engineering

Impact of Output Caching on Website Scalability

Tagged Inc.
San Francisco, CA

Prepared By:

Rajesh Kumar Swaminathan
ID #20194189

Userid rswamina
3B Nanotechnology
Confidential-1

rajesh@meetrajesh.com

May 4, 2009

Rajesh Kumar Swaminathan
#27-8289 121A St.
Surrey, BC V3W 1G6

May 4, 2009

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

Dear Dr. Ioannidis,

This report, entitled, "Impact of Output Caching on Website Scalability" is a resubmission of my third work term report (WKRPT 300) to clear a failure of my previous report submitted in my 3A term in Spring 2008. I completed this new report at the end of the first term of the double work term immediately following 3B, spanning the months of January 2009 to April 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 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.

The purpose of this report is to investigate the pros and cons of output caching as it applies to the scalability of a massively trafficked website. 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 performance improvements of output caching. I would also like to thank Mr. Terrence Chay for explaining many concepts behind output caching to me.

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 and the software product we ship everyday. There are a lot of issues, benefits and outcomes to be discussed while implementing an efficient output cache, and hopefully this report will be able to shed some light on the complexities involved.

I vouch to the fact 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 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 four co-op students, all from the University of Waterloo. Two of these co-ops were in their fourth year while the other two were in their second year of their respective engineering programs.

As part of the revenue projects team, my main goal 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 task was to read and understand the specification for the feature, and then design the back end database tables and structure of the code such that all the requirements of the feature could be met. This task proved tricky as the structure of the code would have to be designed to allow for future expansion 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 and critical analysis. 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 tests to find out whether certain changes to existing features would help increase the popularity of the feature. (See 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 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 were making use of a very fast in-memory cache known as memcache which was already helping us reduce our database load vastly. But even still, all HTML was being re-generated each time for every page request. We use an open source templating engine known as Savant to help us generate the HTML templates. To speed up this process, we started caching HTML output in memcache so certain portions of a page, if not all, could be output to the browser almost instantaneously. 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 features would be highly beneficial. Hence, we implemented a site-wide output caching mechanism. The purpose of this report is to quantify the benefits and to analyze how these benefits reduce as we start caching more and more volatile (quickly expiring) output.

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 output cache, 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 these features were, the more time the users would spend on our site and the more page views they would generate. Users would then spend more time and money on our site and click on ads. So the applications we built contributed directly to the bottom line of the company.

Executive Summary

The purpose of this report is to investigate the pros and cons of output caching as it applies to the scalability of a massively trafficked website. The goal is to analyze the performance gains of 2 types of output caching mechanisms, namely page-level caching and box-level 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 output caching? Output 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 output 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, we would end up showing stale content to the user which is unacceptable. So the end goal of the report is to quantify the performance improvements of implementing an efficient output caching mechanism and analyzing the implementation's pros and cons in the light of relevant constraints.

The report is split into four sections. These four sections are Introduction (Section 1), Requirements (Section 2), Design (Section 3), and Analysis (Section 4).

The major points covered in this report are:

- What caching is and concepts behind caching.
- Why caching is important for scalability.
- Three levels of caching and their pros and cons.
- Criteria for choosing which pages to cache.
- Performance criteria and their weights for deciding between two or more caching strategies.

- Algorithm in PHP for checking if cached data exists.
- Technique to expire cache when data changes.
- Performance analysis of two caching mechanisms, namely page-level and box-level caching.
- Evaluation of these two caching mechanisms in light of above performance criteria.

The major conclusion of this report is that page-caching is 60% more performant than box-caching. However, the box-caching strategy outperforms the page-level mechanism in all the other three criteria, namely flexibility, ease of implementation and volatility.

The major recommendation of this report is to implement a box-level caching strategy. Although less performant, it is overall a better solution than page-level caching 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:

- Page caching was 60% more performant than box caching.
- The box-caching mechanism outperformed the page-level mechanism in all the other three criteria, namely flexibility, ease of implementation and volatility.
- The box-level caching mechanism, although less performant, is superior since it is a better solution overall if we take into account all the real-world constraints.
- A hybrid solution that uses page-level caching for certain users' profiles while box-level caching for others to take advantage of page-level caching's superior performance gains may be a much more effective solution than choosing one over the other.

Recommendations

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

1. Implement an element- or box-level caching strategy instead of a page-level caching strategy for a user's profile page since box-level caching is a better solution overall.
2. Investigate the effectiveness of a hybrid caching strategy where certain users' profiles are cached using the page-caching strategy for superior performance while others are cached using the box-caching strategy so that the cache isn't constantly regenerated due to high volatility.

Table of Contents

Contributions iii

Summary vi

Conclusions viii

Recommendations ix

List of Figures xii

List of Tables xiii

1 Introduction 1

1.1 Caching . . . 1

1.2 Caching Concepts . . . 1

1.3 Caching at Tagged . . . 2

1.4 Why Cache? . . . 3

1.5 Types of Server-Side Caching . . . 4

1.5.1 No Caching . . . 5

1.5.2 Data Caching . . . 6

1.5.3 Element Caching . . . 6

1.5.4 Page Caching . . . 7

1.6 Caching Criteria . . . 7

2 Requirements 9

2.1 Performance Criteria . . . 9

3 Design 9

3.1 Caching Profile Data . . . 11

3.2 Caching Profile Boxes . . . 11

3.3 Checking the Cache . . . 12

3.4 Expiring the Cache . . . 13

4 Analysis 13

4.1 Possible Solutions . . . 13

4.2 Performance Gains 14
4.3 Computational Chart 15

5 Concluding Summary **16**

6 Glossary **18**

References **19**

List of Figures

1 Web page process flow 3
2 Drop down to select US states 5

List of Tables

1 Pros and cons of different caching mechanisms 8
2 Performance criteria for evaluating one or more caching mechanisms 9
3 Performance improvement of two caching strategies 14
4 Computational chart to decide between two caching mechanisms 15

1 Introduction

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 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, such as generated HTML. 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 server, 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 money 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 may be as expensive as generating it 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 the changing of 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 cache and so this helps us reduce our database load vastly. But even still, all HTML is being re-generated each time for every page request. We use an open source templating engine known as Savant to help us generate the HTML templates. To speed up this process, we started caching HTML output in memcache so certain portions of a page, if not all, could be output to the browser almost instantaneously. This is the primary motivation behind implementing output caching. The purpose of this report is to quantify the benefits of caching and to analyze how these benefits reduce as we start caching more and more volatile (quickly expiring) output.

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, 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.

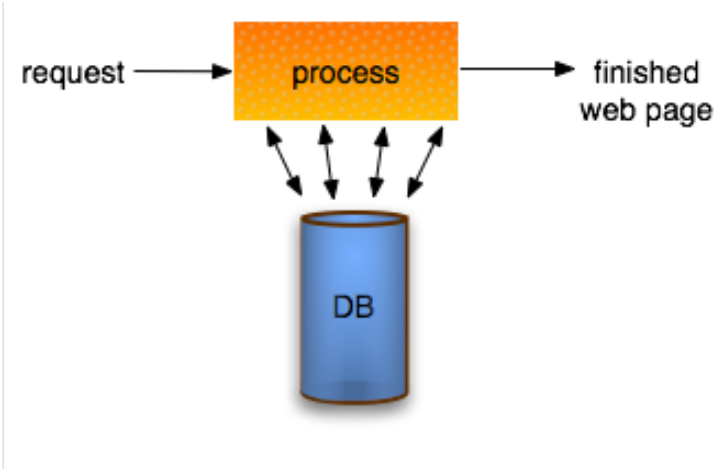

Figure 1: How a conventional dynamic web page is generated. Caching can help reduce the load on the database (DB) which is slow since it is disk-based. Figure from [3].

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.

1.5 Types of Server-Side Caching

Server-side caching is different from other kinds of caches like client-side or proxy server caching in that the former happens on the server itself *before* any content is output to the browser while the latter happens once the data has left the web server, either at the browser level or in a middle tier somewhere. Client-side caching at the browser level does not give us much flexibility while trying to cache dynamic content. Furthermore, since the caching happens outside the web server, they offer very little control in terms of when to expire the cache, etc.

The best type of caching is where we move the data from the database as close as possible to the state in which it is ready to be delivered to the client, the browser in our case. Let us illustrate this concept by looking at a common example: a drop down menu which contains a list of states from which the user is expected to choose his/her home state as in Figure 2.

So for example, let us suppose we have a table in the database containing a list of all the names and abbreviations of each state in the US. The goal is to build the dropdown box using the database so that as we add states or need to make changes to the name of the states (to correct a typo for instance), those changes are reflected in the web pages that contain that drop box.

So since the list of states do not change all that frequently, at least not in the next couple of hours in which time hundreds of people could possibly be hitting the web page, it does not make sense to hit the database to dynamically build the drop down each time.

There are four different way of building the state selection drop down box. The method we choose defines the *level* of caching. They are:

1. **No Caching:** Hits the data source for each request of the page.

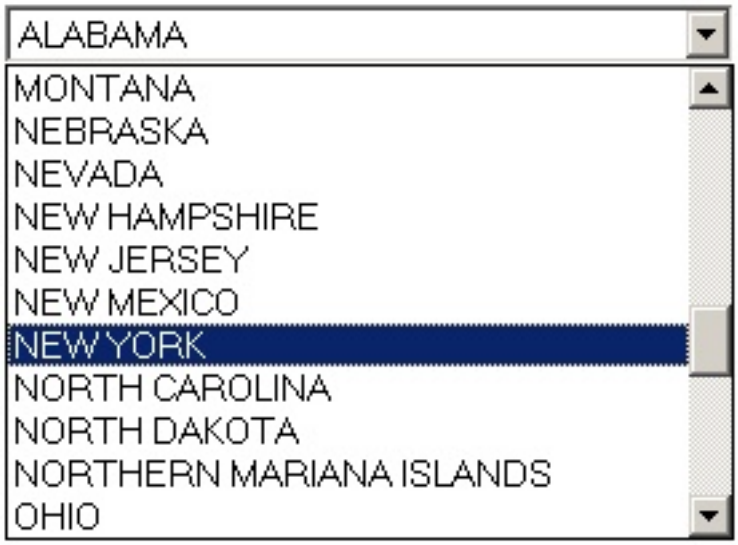

Figure 2: A drop down box to allow a user to select a state from a list. There are different ways of caching the HTML for this drop box. Figure from [4].

2. **Data Caching:** Stores the data in a temporary high-speed location such as in the RAM and then builds the output using it.
3. **Element Caching:** Takes the data and transforms it into the appropriate output and then caches the output.
4. **Page Caching:** Caches the whole page and saves it as a file.

Now we will dissect each of these levels of caching and analyze their pros and cons.

1.5.1 No Caching

This is the easiest method and does not really need much discussion. This method is only mentioned to provide a baseline to compare against for the others. Most truly dynamic data that change frequently like stock quotes should not be cached. Also if the cost of accidentally displaying stale content is very high, like in a user's bank statement, the output should not be cached. In these cases, it makes sense to dynamically generate the output each time. Generally, real-time data should not be cached to not only because we might receive complaints from user about seeing stale data, but also the cost of generating, storing, and checking the validity of the cache for real-time

data would be higher than generating the output itself. This defeats the purpose of the cache which is to speed up page generation.

1.5.2 Data Caching

Data caching is the act of making a copy of the data from the database or other slow data sources like external websites and storing it in a location that is faster to access. Where this data is stored depends on how it will be used and how quickly and frequently it will be accessed. We could either store the data as an application level variable in memory, or on disk for larger amounts of data.

However, text files on disk are accessed very slowly, so this method is useful only when caching data obtained from a very slow link externally. For example, we could cache HTTP requests from other servers. The cache could be useful in those scenarios where the network link is not reliable or if the response from the other server takes too long. This is a much better option than displaying an error to the user or displaying a blank white page.

While this method can substantially increase performance, it still leaves us with the data in a state that we can manipulate it. In our example, we can not only build a select box of states, we could also display them in another unrelated table, or sort them in reverse order for whatever reason. In short, the data is still flexible and we can manipulate it easily in order to use it for a variety of purposes.

1.5.3 Element Caching

This level of caching takes data caching and goes one step further. The question to ask is that if we are always going to be building a drop down out the list of US states, why should we be caching the list of states when we might as well just cache the HTML needed to generate the drop down? The goal here is to speed things up even more by only doing the processing to build the box once and then caching the output. Then when we want to display the box again, it will be quick and easy.

While this method is faster and just as easy as data caching, the single biggest drawback is that we lose some of the flexibility. For example, if we wanted to extract a simple list of states from a HTML string containing a drop down box of the states, we would need to parse the HTML string which is a lot of work. At that point it is probably easier to go back to the database to get the data again.

1.5.4 Page Caching

In this level of caching, we do all the processing to generate the page once and then cache the *entire* HTML page and all the elements contained in it. The benefit here is that there is no processing involved at all when requesting the cached files. There is no code that needs to be executed once the page is built except for perhaps checking if the cache is valid.

The main drawback here is that determining when to expire the cache can be a little difficult since no processing is happening. If any data in the database that the cache contains is changed, the cache needs to be updated. This can sometimes be a lot of data for large pages. If the cache is not expired when the data changes, we risk displaying stale content to the user and this is usually a bad thing. Hence there is a certain amount of manual work that needs to be done to refresh the cache or setting up a scheduler to automatically delete the cache at a fixed interval.

The following table (Table 1) summarizes the four levels of caching and their respective pros and cons.

1.6 Caching Criteria

The most difficult part of any caching system is knowing when the content should not be cached anymore. If we do not cache it long enough, it can defeat the purpose of the cache in the first place. On the other hand, if the cache is not refreshed often enough, we run the risk of displaying stale data to the end user.

Hence the granularity of the cache depends on the *volatility* of the data. Volatility defines how

Table 1: A list of various caching mechanisms and their pros and cons.

Caching Mechanism	**Pros**	**Cons**
No Caching	Always up-to-date	Needs to hit database on every request
Data Caching	Good when database access is slow or data is obtained from external server	Need to access external data server each time which consumes bandwidth
Element Caching	Good for elements or parts of the web page that change infrequently	Still a lot of the page being generated dynamically
Page Caching	Fastest performance	Needs to be re-generated frequently

quickly the data changes. In our example above, the drop down box for a list of states makes for an ideal element to cache since it is hardly going to change, if at all. We therefore say that the data that defines the US states has low volatility. Content that lives in the database mostly for maintenance purposes and is used a lot but rarely changes would make a perfect candidate for some form of caching.

A user's profile page is a great example of when to cache. Once the user has submitted his profile information, he/she may change it a couple of times in the first week or so, but for the most part it rarely changes, so it would be a good idea to cache the entire profile page for that user.

The most generic solution is to cache based on two important criteria:

1. Amount of traffic to the page
2. Amount of data that seldom changes that the page uses

If a page gets a lot of traffic then we would want it to be as fast as possible and use as few resources as possible. Even a small performance increase on a busy page can make a big difference in the website's apparent speed and processor utilzation on the web server. If the generation of the web page has to hit the database every time, it is going to be typically slow unless the database implements some kind of lower-level cache on its own.

On the same note, if a page uses a lot of slow data sources, then the page is a natural candidate for caching even if it does not get a lot of traffic just so it loads faster when it is requested.

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 efficient.
3. Under no circumstance should stale data be shown to the user.

2.1 Performance Criteria

The following criteria in Table 2 are to be used when deciding which caching mechanism is more suitable for Tagged:

Table 2: A list of performance criteria to be used when evaluating one or more caching mechanisms.

Criterion	**Weight**	**Description**
Flexibility	15%	Ability to use cache in multiple scenarios
Ease of Implementation	15%	Work needed to maintain the cache
Volatility	35%	Frequency of cache expiration
Performance Improvement	35%	Speed gains compared to no caching

3 Design

We could implement caching at two levels, both of which have the ability to address all the requirements above:

1. Cache the entire page as a single object.

   Pros: The pro of this level of caching is that the cache would heavily speed up the processing of the page because the entire page can be output as static content verbatim, much like how a static CSS or Javascript file may be output. The cached content can also be compressed for more efficient storage on the server. The added benefit of caching the entire page is that

we can host this static cached version of the dynamic page on one of our content distribution networks (CDNs) such as Akamai or Limelight that are spread across the world for very speedy content delivery depending on the origin of the request.

Cons: But on the other hand, if a user's profile page is cached in its entirety, whenever the user changes his profile information, the cache will be marked as stale and will be regenerated the next time it is accessed. Therefore, the cache object is likely to expire more frequently thereby reducing the usefulness of the cache. Even one small change in the data that the page relies on invalidates the entire cache requiring it to be re-generated again.

2. Cache portions of the page as a collection of cache objects.

Pros: The advantage of caching portions of the page as multiple cache objects is that if a small piece of information were to change, we would only need to expire a few of the cache objects in the collection. The remaining portions of the page would still be output quickly. Caching portions of the page would imply that we mostly cache AJAX responses from the server. The pages that are content heavy are usually dynamically loaded as "boxes" using AJAX calls to the server to increase the perceived response of the server. Caching the AJAX responses means that we do not cache the output of the entire web page, but of certain infrequently changing parts of the website like the user's "friends box" or "gifts box" for example. This is the primary advantage of caching portions of a page as opposed to the entire page.

Cons: This has a higher overhead in checking and maintaining the cache. It also does not eliminate the need for the browser to make an AJAX request to the server to obtain the cached data. This limitation can be lifted by rendering the page as a whole without making any AJAX requests.

Thus the caching of the profile page is designed in such a way to minimize the number of times we would need to mark it as stale.

3.1 Caching Profile Data

Since at Tagged the user's profile data is so large, in order to implement an efficient caching mechanism, we split the user's profile data into 4 chunks:

1. **Simple User Data:** The user's basic profile information such as name, gender, profile picture, privacy settings, language settings, etc.
2. **Profile User Data:** User information that is commonly needed when a user is logged in, when someone views the user's page, or tries to interact with the user. Examples include the user's timezone, relationship status, ethnicity, sexual orientation, etc.
3. **Extended User Data:** User information that is accessed rarely such as the user's IP address, the ID of the user who referred this user, search preferences, etc.
4. **Messy User Data:** Everything else about the user. For example, the user's response to random polls/surveys he maybe presented with upon login.

Splitting the user's profile data into chunks means that we do not have to invalidate the *entire* cache even if one small piece of user data were to change. In our case, we would have 4 caches for each user, and we would only need to expire one of the caches for every change in the user's profile data.

3.2 Caching Profile Boxes

The profile page of a Tagged user contains 13 boxes that uses various chunks of that user's profile data. The most frequently changing box is the "What's New" box which is a mini-feed of the user's most recent actions. The least commonly changing box is the user's "about me" box that displays the user's favorite music, books, shows and movies. Most of these 13 boxes usually change only when the user logs on and performs an action. But there are a few boxes, namely "Gifts", "Tags",

and “Comments” that change when someone else on the site, other than the user, interacts with the user.

3.3 Checking the Cache

The HTML content for each box on the profile page is obtained by means of an AJAX call. Each AJAX request translates to an API call in the PHP system. This following is the general strategy to use in PHP to check for existing cache data. If the cache data exists, return that and exit right away. Otherwise, generate the content in the regular fashion using the Savant template and then save it to the cache before returning it.

```
public function some_api_method($params) {
    // 1. check if this a valid api call
    // 2. check if user is logged on
    // 3. run input filters on $params
    // 4. get the cache object for this api method
    $cc = $_TAG->apicache($params);
    // 5. is there existing cache data for this api method?
    if ($data = $cc->getCache()) {
        // 5.1 return the cached output
        return $data;
    } else {
         // 5.2.1 otherwise generate output regularly
        $page = new Savant($templateName);
        $html = $page->fetch();
        $result = $this->generateResult(array('result' => $html));
        // 5.2.2 and save it to the cache for future use
        $cc->setCache($result);
        // 5.2.3 return the output
        return $result;
    }
}
```

3.4 Expiring the Cache

One of the requirements of our caching strategy is to not show any stale data to the end user under any circumstances. This calls for a way for lower-level objects such as the profile object to report to the higher level cache object instructing it to expire all cache data associated with that profile chunk.

The design proposes an elaborate *messaging system* where the caching system can "subscribe" to events triggered by the lower-level profile objects. Thus the caching objects are constantly listening for changes in profile data, and when the profile object publishes to the world that it has been changed, the caching layer can pick up on this change and decide what to do. It can either expire the cache associate with the profile chunk that got changed. This means that the next request for that user's profile information will be slow since it will need to be retrieved from the source. The alternative is for the caching layer to regenerate the cache right away for the profile chunk that was just expired. This has the benefit that the next time a request comes in for that user's profile data, it can be delivered immediately without going back to the source. But on the other hand, it means that we may be generating cache objects for data that could potentially never be accessed again thus wasting space.

4 Analysis

4.1 Possible Solutions

1. Implement a page-wide cache. This level of caching was explained in detail in Section 1.5.4.
2. Implement an box-level (or element-level) cache. This level of caching was explained in detail in Section 1.5.3.

The above two solutions will be evaluated against the performance criteria specified in Table 2 in Section 2.1.

4.2 Performance Gains

How do we measure the performance gains obtained through output caching? The best way to do this is to measure the amount of time it takes for the page to return its output. For this we use a simple diagnostic technique and a timer to time the duration it takes to make the API call. If the API call was cached, it would return almost instantly. Otherwise, the page will generate the output dynamically and return its output after a delay. The difference in these two times will yield the performance gains through output caching.

All tests were run on a 2.4 GHz Intel Core 2 Quad CPU with 3 gigabytes of RAM. The PHP version used was 5.2.5 with a standard APC cache. The results are summarized in Table 3.

Table 3: Benchmark of output times of two caching strategies and their respective improvements. All times are in milliseconds.

	# Ajax Requests	**# Using Cache**	**No Cache (ms)**	**Box Cache (ms)**	**% Diff**	**Page Cache (ms)**	**% Diff**
1	17	9	4887	1952	60%	488.77	90%
2	17	9	4947	1944	61%	408.21	89%
3*	17	0	3885	–	–	–	–
4*	17	0	3863	–	–	–	–
5	18	9	2714	2318	15%	372.27	86%
Avg.	17.2	5.4	4059	2071	**27%**	436.252	**89%**

* = These trials were run with output caching disabled. Hence there is no data for timer or a % difference.

From the above table, it is clear that the box caching mechanism offers only a 27% performance improvement over the baseline case whereas the page caching mechanism offers a performance improvement as high as 89%. Thus we find that the page caching mechanism provides roughly 60% more savings in performance over the box caching mechanism. This improvement is expected since page caching avoids the overhead of making AJAX calls to the server, processing the response, and injecting the output into the existing HTML page. The improvement is also more significant since only 50% of all AJAX requests are being served by the cache. If we were to have more AJAX requests for each box cached, the difference between the the box caching mechanism and the page

caching mechanism would not be as dramatic.

4.3 Computational Chart

In the following table (Table 4), each caching mechanism is scored, on a scale from 1-10, against the criteria in Table 2. The higher the number, the stronger the caching mechanism scores for that criterion. The score is then weighted against the importance of the criterion and then summed to provide an overall score for that caching mechanism.

For example, the box caching mechanism has an ease of implementation score of 7.0, but that criterion is only 15% important, so the adjusted score is $7.0 \times 15\% = 1.1$.

Table 4: A computational chart that assigns a quantitative score to each of the decision-making criteria specified in Table 2. The higher the number, the stronger the caching mechanism scores for that criterion.

Criterion	**Weight**	**Page Cache**		**Box Cache**	
		Score	Value	Score	Value
Flexibility	15%	2.0	0.30	6.0	0.90
Ease of Impl.	15%	3.0	0.45	7.0	1.10
Volatility	35%	1.0	0.35	7.0	2.50
Perf. Improv.	35%	8.9	3.12	2.7	0.94
Sum			4.20		5.30

As we see in Table 4, the box caching mechanism is superior than the page caching mechanism since it has a higher overall score. We thus conclude that Tagged should go ahead with the implementation of a box caching mechanism rather than a page caching mechanism.

However, we also saw that the page caching strategy outdoes the box caching strategy in performance by almost 60%. This is a great performance improvement that we don't want to lose. We should therefore invest some time and resources to investigate a hybrid caching strategy where certain users' profiles are cached using the page caching strategy for superior performance while others are cached using the box caching strategy so that the cache isn't constantly regenerated due to high volatility.

5 Concluding Summary

In this report, we introduced the major concepts behind caching, server-side caching and output caching. 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 scalability. We analyzed the three *levels* or mechanisms of caching and studied their pros and cons. These three caching levels were data caching, element caching and page caching. We then proceeded to look at the criteria that would make a page ideal for caching. We saw why "amount of page traffic" and "amount of infrequently changing data on page" were the two most important criteria in deciding which pages to cache.

Since output caching was necessary 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 caching strategies were discussed in detail, namely page caching and element-level caching, also known as box caching. We saw that the most effective caching strategy was one that minimized the number of times we would need to mark it as stale.

We focused our attention on caching the user's profile data and his/her profile page since this was the most frequently accessed page on the entire site. Due to the size and complexity of the user's profile data, we split it into four chunks, thereby establishing a caching hierarchy. This way, if a user's profile information were to change, we would only need to expire the cache objects that relied on one of these four chunks.

Next, we studied the structure of the profile page and observed how most of the "boxes" on this page changed only upon the user taking action. But 3 out of the 13 boxes were generated using content that could change based on other people's actions on the website. This observation is key to understanding how frequently a cache box expires, i.e. its *volatility*.

We then specified a simple algorithm to check the cache for the existence of cached output. If

cached data is available, we use that, otherwise we generate it and store it in the caching layer before returning it. We provided the skeleton code in PHP for such a check in an API function call.

Since the caching mechanism deployed would tolerate absolutely no stale content, we needed to design an elaborate *messaging* system that allowed lower-level data objects to publish a notification when their data changed which would then be picked up by the higher-level caching objects, by virtue of having subscribed to these events, and proceed to delete or re-generate their cache.

The performance improvements of two caching strategies were obtained and studied in light of the criteria specified in Table 2. We saw that page caching was 60% more efficient than box caching. However, performance improvement was only one aspect of our decision-making criteria that accounted for only 35% of the effectiveness of the entire solution. From the computational chart in Table 4, we saw that the box caching mechanism outperformed the page level mechanism in all the other three criteria namely flexibility, ease of implementation and volatility. We eventually concluded that the box level caching mechanism was superior since it was a better solution overall.

Finally, we recommended investigation of a hybrid solution that uses page-level caching for certain users' profiles while box-level caching for others to take advantage of page level caching's superior performance gains.

6 Glossary

1. **AJAX** stands for asynchronous Javascript and XML. It a mechanism through which content can be obtained from the server without having to force the browser to refresh the page.
2. **APC** stands for advanced PHP cache. This extension to PHP allows us to cache partially compiled PHP code for speedy execution.
3. **API** stands for application programming interface. An API is a set of functions which in our case the browser calls via an AJAX call to obtain HTML content to display on the page.
4. **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.
5. **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.
6. **HTTP** stands for hyper-text transfer protocol. It is the protocol by which the browser communicates with the web server.
7. **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.
8. **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.
9. **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.
10. **Savant** is a templating system for use in PHP applications. It allows us to specify the variables used in the template and their values and helps us generate HTML quickly using these template variables.

References

[1] "Cache", *Wikipedia, The Free Encyclopedia.*, http://en.wikipedia.org/w/index.php?title=Cache&oldid=286990042. Retrieved May 1 2009.

[2] Jakob Neilsen, "Feature Richness and User Engagement", *Jakob Nielsen's Alertbox*, http://www.useit.com/alertbox/features.html. Retrieved May 4 2009.

[3] Frank Talk, "BFT: Our case for 100% de-normalization", *Franks Talk's Personal Blog*, http://www.frankf.us/wp/?p=25. Retrieved April 21 2009.

[4] John Peterson, "Server-Side Caching Options", *ASP101.com*, http://www.asp101.com/articles/john/server_side_caching/index.asp. Retrieved April 18 2009.

[5] G. C. Stierhoff and A. G. Davis. "A History of the IBM Systems Journal." *IEEE Annals of the History of Computing, Vol. 20, No. 1 (Jan. 1998), pp. 29-35.*

[6] Mark Nottingham, "Caching Tutorial for Web Authors and Webmasters", *mnot.net*, http://www.mnot.net/cache_docs/index.html. Retrieved April 19 2009.

[7] Mike Pub, "Why should we use caching?", *mikespub.net*, http://mikespub.net/tools/postnuke/caching.html. Retrieved April 25 2009.

[8] Scott McFarland, "Caching with ASP.NET", *4 Guys From Rolla.com*, http://aspnet.4guysfromrolla.com/articles/022802-1.aspx. Retrieved April 25 2009.

[9] ViSolve White Paper, "Web Caching - A cost effective approach for organizations to address all types of bandwidth management challenges", *ViSolve Publishing*, http://www.visolve.com/squid/whitepapers/ViSolve_Web_Caching.pdf, March 2009.