Building a Scalable Collaborative Web Filter with Free and Open Source Software

Michael G. Noll
Hasso Plattner Institute, Germany
& University of Luxembourg, Luxembourg
michael.noll@hpi.uni-potsdam.de

Christoph Meinel
Hasso Plattner Institute, Germany
meinel@hpi.uni-potsdam.de

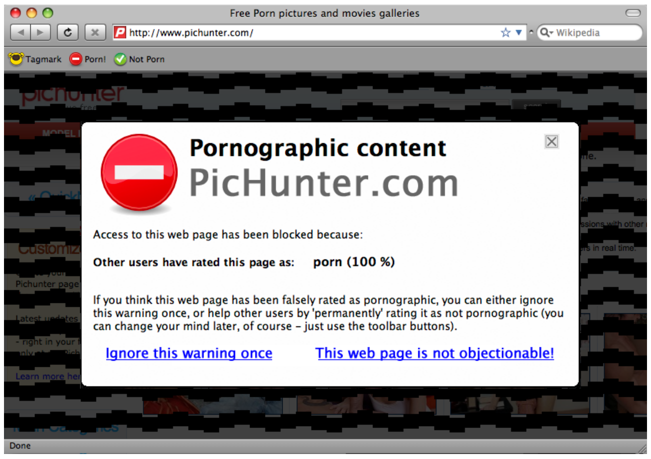
Abstract

In this case study, we describe the design and architecture of a scalable collaborative web filtering service, TaggyBear, which is powered by free and open source software. We will introduce the reader to the ideas and concepts behind TaggyBear, and discuss why we picked the software components that form the basis of the service. We will talk about how we combined or extended their functionality to build the TaggyBear service, and provide some initial benchmarking results and performance figures.

1. Introduction

In this paper, we will describe the design and architecture of the social web filtering service *TaggyBear*, which we have developed as part of an Internet Security research project. Similar to traditional social bookmarking services such as Delicious.com, TaggyBear allows users to bookmark web pages and annotate these bookmarks with unstructured keywords, so-called *tags*. Seen in this way, TaggyBear can be used for tasks such as storing your bookmark collection online, recommending interesting websites to your friends, or knowledge acquisition. However, the same data is also leveraged to create new services which go beyond the standard bookmarking scheme. For example, we have shown in a previous work [10] how to perform web search personalization via social bookmarks by adding a "personalization layer" on top of search engines.

In this paper, we focus our description of the TaggyBear system on the social web filtering component which can be used to filter unwanted or harmful Internet content. Users can collaboratively categorize web pages, and mark them as objectionable (e.g. pornographic content, violence, racism), dangerous (e.g. phishing, malware) and the like. Figure 1 shows a screenshot of the TaggyBear browser extension blocking access[1] to an objectionable website because the majority of users have rated it as pornographic.


Free Porn pictures and movies galleries
http://www.pichunter.com/
Wikipedia
Tagmark
Porn!
Not Porn
Done

Pornographic content
PicHunter.com

Access to this web page has been blocked because:

Other users have rated this page as: **porn (100 %)**

If you think this web page has been falsely rated as pornographic, you can either ignore this warning once, or help other users by 'permanently' rating it as not pornographic (you can change your mind later, of course – just use the toolbar buttons).

Ignore this warning once   This web page is not objectionable!

Figure 1. TaggyBear browser add-on

The rest of the paper is organized as follows: In the next section, we outline the concepts behind the TaggyBear service in more detail, and the requirements that arise from them. In section 3, we describe the system design and architecture that we have implemented with free and open source software to meet these requirements, and we provide some benchmarking results. Lastly, we summarize and discuss our findings in section 5.

2. Scenario

2.1. Use Case: Social Web Filtering

The social web filtering approach of TaggyBear leverages the "wisdom of the crowd" in the spirit of Wikipedia. It

[1]The user's preferences determine which type of web pages should be blocked, thus giving the user full freedom to shape her Internet experience by herself.

allows end users to collaboratively rate and categorize web pages, and analyzes and aggregates the collected data for the purpose of web filtering. In turn, users can query TaggyBear for the user community's rating information about a given web page so they can decide themselves whether they do or do not want to retrieve/see the page. It is important to note that any filtering happens *on the client side* since TaggyBear only provides the necessary rating information on request – it's up to the client end point how it reacts to this information[2]. This approach is very different from current server-side filtering where access to web pages is blocked on ISP level, for instance [3]. One could say that TaggyBear acts as a kind of social "overlay" of the Internet, and provides a true democracy of the web with regard to content filtering.

2.2. Ratings

The main landing point for users is the TaggyBear website. Here, they can perform tasks such as creating and sharing bookmarks, browse their network of friends, et cetera. In the context of social web filtering, they can also ask TaggyBear for *rating information* about a web page. These ratings are derived from social bookmarking data as we will describe shortly. For any web page, TaggyBear may return up to three different types of ratings:

- the *user rating*
- the *community rating*
- the *system rating*

The order of preference for the three rating types is decided by the client software, or the end user (see the browser add-on example in section 2.3). Please note that the distinction into three different rating types is also important for the system design as we will see later.

The user rating of a web page is the rating provided by the given user herself. The rating is the set of tags with which the user annotated the bookmark of a web page. Tags bear various kinds of semantic meanings [1, 7], and we have shown in previous studies [9, 11] that they are often used for classification purposes. This means that we can exploit tags from the social bookmarking aspects of TaggyBear for social web filtering without putting another burden on the end user – he even has an additional benefit from her bookmarking activities!

In contrast to other social bookmarking services, there is a voting element in tags on TaggyBear which transform tags to ratings as described in [8]. Prefixing a tag with a minus sign "-" is defined as a negative vote (e.g. "`-porn`"), whereas in any other case the tag counts as a positive vote. Also, users can opt to make a bookmark private for privacy reasons. In this case, neither the bookmark nor the resulting rating is shared with other users.

The community rating of a web page is the aggregation of all public user ratings. In the easiest case, the aggregation is the set of all tags and their associated votes. For each tag, we keep track of the number of positive votes and the number of totals votes. These can be used to calculate percentages for a given tag, and determine whether the majority of users voted for or against a specific tag. For the context of this paper, we notate community ratings as `tag:pos:total`; for example, `porn:13:68` means that 13 out of 68 users, 19%, voted positively on tag `porn`.

The system rating is a special rating provided by the TaggyBear operators when and where needed. The purpose of the system rating is two-fold. First, it is used to mitigate the cold-start problem where there isn't a sufficiently large amount of rating information available at the launch phase of the service. Second, it is used to harden the service against abuse and spam, e.g. to protect against "rating bombs" that would result in preventing access to innocent (and most notably, popular and highly visited) web pages. On one hand, the system rating serves as a moderator function by providing an operator-managed whitelist and blacklist of web pages similar to services such as Google Safe Browsing[3]. On the other hand, a client software is not required to honor the system rating at all, so the system rating is a "voluntary" moderation feature and not a hidden type of censorship.

user rating
research, science, weblab, phd, hpi, -porn
community rating
hpi:56:56, research:42:42, germany:26:31, ..., porn:0:1
system rating
(none)

Table 1. Exemplary ratings for the HPI home page. For example, 26 out of 31 users voted positively for the tag "germany".

To give an impression of how such ratings would look like in practice, consider the exemplary ratings for the home page of the Hasso Plattner Institute, http://www.hpi.uni-potsdam.de/, shown in table 1.

[2]For example, client software for individual end users should give full control to its users. In a school environment however, the network administrator might want to interface a central web proxy with TaggyBear to enforce specific filtering actions.

[3]http://code.google.com/apis/safebrowsing/

2.3. Clients

Since the main purpose of social web filtering is to protect users from unwanted and harmful content as they browse the Internet, we have developed a TaggyBear browser add-on for Mozilla Firefox. Whenever a user visits a new web page, the add-on queries the TaggyBear API for rating information. Access to the web page will be blocked (see figure 1) depending on the user's preferences. In our case, the order of preference for ratings is:

$$user\ rating > system\ rating > community\ rating$$

The end user's own rating is the *ultima ratio*. The system rating has precedence over the community rating to prevent abuse of the service. The content of a blocked page is obfuscated, and a warning popup window informs the user *why* the add-on triggered the protection mechanism. The popup also comes with interface elements to ignore the warning just temporarily or to permanently overwrite the community rating with the user's own, individual rating. The latter allows the user to easily create her own whitelist and blacklist of web pages.

The browser add-on also provides interface elements to create new bookmarks ("Tagmark" button in the exemplary screenshot in figure 1) so that users don't have to visit the TaggyBear website, and interface elements to quickly rate and categorize web pages into specific content categories with a single click ("Porn" and "Not Porn" buttons in this example).

Another way to use TaggyBear is to interface it with web proxies such as Squid[4] in a centrally managed network, where the web proxies are configured to query TaggyBear for rating information about any requested web pages and block access depending on how the user community categorized these pages.

2.4. Requirements

We identified three main system requirements:

1. Real-time access and updates (read/write) for individual user data, including user ratings
2. Lookup performance (read) for ratings
3. Quality of community data

In terms of (1), users expect that any of their actions such as creating or modifying bookmarks take effect immediately. This requirement also means that *user ratings* – which are derived from bookmarks – should be updated in real-time.

[4]Squid is a free and open source caching proxy for the web supporting HTTP, HTTPS, FTP and other protocols. http://www.squid-cache.org/

In terms of (2), client software such as the TaggyBear browser add-on will direct a large and steady number of queries towards the service. Whenever a user visits a new web page, the add-on will trigger a request for rating information, i.e. user, community, and system ratings. In order to meet the demand, lookups of such ratings must be very efficient. In addition, caching strategies should be put in place to mitigate service load and improve scalability.

In terms of (3), the community ratings aggregated from all public user ratings should meet quality requirements such as being up to date and free of spam or junk data. This means that the system should be designed in a way that allows for efficient analysis of large amounts of data and handle service growth.

3. Setup

3.1. Overview

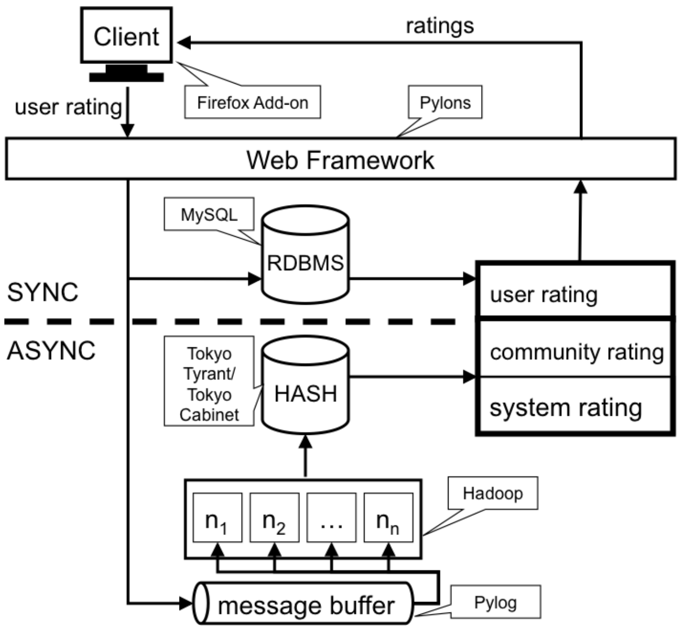


Figure 2. System overview (simplified)

A simplified overview of the system is shown in figure 2. For the sake of readability, we omitted components such as reverse proxies or caches. The Linux-based system consists of five main components as follows, all of which are or are based on free and open source software:

- Web framework:
  Pylons (http://www.pylonshq.com/)
- Data stores:
  MySQL Community Server (http://www.mysql.com/),

Tokyo Cabinet and Tokyo Tyrant
(http://tokyocabinet.sourceforge.net/)

- Message buffer:
  Pylog – Python logger, a custom development based on Twisted (http://twistedmatrix.com/)

- Cluster for data aggregation and analysis:
  Hadoop (http://hadoop.apache.org/core/)

- Client:
  Add-on for Mozilla Firefox (http://www.mozilla.org/)

Based on the requirements outlined in section 2.4, we decided to separate the system into two complimentary parts. First, there is the "synchronous" part which is responsible for handling user data including user ratings. Here, we provide TaggyBear users *real-time* access to their own data, i.e. there is no delay on retrieving, adding, updating or deleting their data. Additionally, flexibility with regard to features and future feature additions to the TaggyBear service is important. We therefore chose an RDBMS as a data store, namely the *MySQL Community Server*.

Second, there is the "asynchronous" part which is responsible for handling community and system ratings. The idea is to buffer any incoming data (user bookmarks for aggregation into community ratings) and periodically process and clear these buffers through batch jobs. Here, lookup performance and efficient data aggregation and analysis are the primary objectives. Since the feature set in terms of lookups is rather fixed, we do not need the flexibility of a full-fledged RDBMS. Instead, we chose a hash table as a data store for community and system ratings as it provides constant-time $O(1)$ lookup on average. We use the library *Tokyo Cabinet* and its companion *Tokyo Tyrant* for this task. Tokyo Cabinet is the successor of QDBM and like its ancestor written by Mikio Hirabayashi. Tokyo Tyrant adds a network interface to Tokyo Cabinet so that the hash table can be accessed remotely.

For data aggregation and analysis, we picked the Hadoop framework which supports the MapReduce [2] distributed computing metaphor and comes with its own fault-tolerant distributed file system, HDFS . However, HDFS is designed and tuned for reading and writing *large* files – writing rather small incremental "updates" to HDFS whenever a user adds a new bookmark (for later aggregation) is not optimal. We worked around this problem by developing a lightweight, persistent message buffer called *Pylog* based on the *Twisted* networking framework.

In the following sections, we describe the TaggyBear system and its components in greater detail.

3.2. Data flow

In this section, we describe the data flows for adding data to and retrieving data from TaggyBear, respectively. It should give the interested reader an insight into how TaggyBear works behind the scenes so that the following sections are easier to understand. We are only describing the data flows for the API here (see section 3.4), but the flows for the web interface outlined in section 3.3 are analogous.

The data flow for adding bookmarks is shown in figure 3. When a user submits a new bookmark through a client software to the API, the request and its payload is first run through several sanity checks and input filtering. If it passes these tests, the bookmark is stored in the user's bookmark collection in the RDBMS and then, if the bookmark is public, submitted to the message buffer (the RDBMS and the message buffer are described in section 3.5) . In this case, the API returns a `200 OK` HTTP status code to the client. If it does not passes the test, the API returns an appropriate error code, for example `403 FORBIDDEN` if user authentication failed.

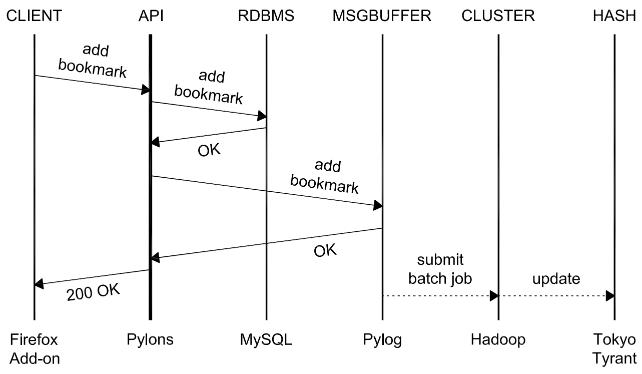


Figure 3. Data flow for adding bookmarks/ratings (WRITE). The dashed lines represent asynchronous tasks that are carried out at a later time.

The previously described steps occur synchronously. The processing of bookmarks in the message buffer is conducted asynchronously at periodic intervals (currently every 30 minutes), when the buffered bookmarks will be copied to the Hadoop cluster's distributed file system, HDFS. As described in section 3.6, MapReduce jobs will be started to aggregate the public bookmarks into community ratings, and the result of these jobs will be inserted into the hash table through Tokyo Tyrant.

The data flow for looking up rating information is shown in figure 4. When a client requests rating information about a URL, the request is routed via the API to the hash table (see section 3.5) which will return any available community and system ratings. If the request comes from an authenticated user, the RDBMS is also queried for any individual user rating. Finally, the rating results are returned back to the client.

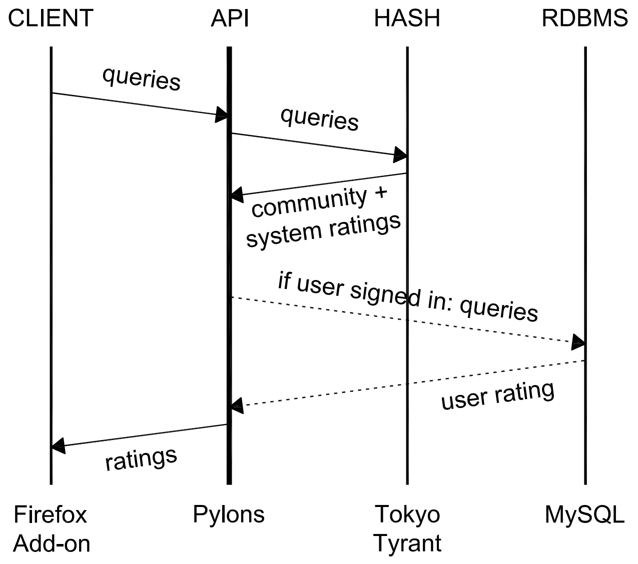


Figure 4. Data flow for lookups (READ). The dashed lines represent tasks that are only carried out if the user is signed into TaggyBear.

As one can see, unauthenticated requests – for example from a web proxy configured to interface with TaggyBear – will only result in one query against the hash table, the result of which can be adequately cached because it is indiscriminately valid for any user. For authenticated users, a request will additionally result in a SQL query against the RDBMS.

3.3. Web Interface – for humans

As we said previously, the main landing point for users is the TaggyBear website. The website provides a number of features such as managing and backing up your bookmark collection, subscribing to RSS feeds, browsing your network of friends, or looking up information about web pages.

The website is implemented with the Python web framework *Pylons*. Pylons is a very lightweight and very modular framework: It is easy to use your favorite Python components and libraries for templating, database access, request dispatching et cetera. Most interestingly, Pylons is not very tied to specific conventions with regard to data models and database access and their interaction with other parts of framework[5]. This is ideal for our intended setup involving separated, heterogeneous data stores.

3.4. API – for computers

TaggyBear allows programmatic access to its data via a public RESTful API [5]. REST, or Representational State Transfer, refers to a collection of architectural principles used for transfer of information over the web, but is now used to describe simple RPC-based protocols using XML over HTTP [6]. Its benefits – which are part of the reasons why a lot of open source web applications are using REST – include being lightweight and cacheable, which helps to reduce server load and improves scalability. Additionally, it uses the inherent HTTP security model, which means that system operators can restrict certain methods (e.g. `DELETE`) to certain URIs by firewall configurations.

Path prefix for REST API URIs: /api/rest/$version$

GET	/items/$hash$	get ratings for URL
POST	/$user$/bookmarks	add a new bookmark
PUT	/$user$/bookmarks/$hash$	update a bookmark
DELETE	/$user$/bookmarks/$hash$	delete a bookmark

Table 2. REST API features (excerpt). URLs are represented by their MD5 hashes in the REST API URIs.

Like the web interface, the TaggyBear API is implemented with Pylons and is thus part of TaggyBear's web framework. The API features include adding, modifying, retrieving and deleting data from TaggyBear (see table 2). For example, the browser add-on uses the API to submit new bookmarks to TaggyBear or query for rating information about web pages.

3.5. Data storage

In this section, we talk about the data storage components of TaggyBear. The current setup consists of three different data stores:

- RDBMS – MySQL
- Hash table – Tokyo Cabinet/Tokyo Tyrant
- Message buffer – Pylog

As we said previously, the main purpose of the RDBMS is to store individual user data. We use the Python library

[5]A big advantage of web frameworks such as Ruby on Rails or Django is that they *do* have very specific conventions how their components play together, because it makes common development tasks so much easier. In our case however, we need the flexibility that Pylons innately provides.

SQLAlchemy[6] to interact with the MySQL database(s) of TaggyBear. For the context of this paper, we do not go into details on how to scale MySQL databases but refer interested readers to excellent works such as [13, 6].

The hash table is powered by Tokyo Cabinet. It provides constant-time $O(1)$ lookup on average and $O(log\ n)$ in the worst case. Tokyo Cabinet is also very fast for writing data: Hirabayashi reported 1.5 seconds (elapsed time) for storing 1 million records in a hash table database[7]. This is an important criterion because we need to import large numbers of community ratings into the hash table after data aggregation. Tokyo Tyrant provides a network interface to Tokyo Cabinet, which we use when querying the hash table for community ratings via Pylons (see section 3.2 on data flows). While there is a performance loss when accessing Tokyo Cabinet databases through Tyrant, particularly when using its HTTP or memcached-compatible interface, it is still more than adequate for our needs. Tyrant also supports features important for scalability and fault tolerance such as replication, update logs, hot backup.

The message buffer is a special data store as it is only used to buffer incoming bookmarks until they can be processed and aggregated in the Hadoop cluster (see section 3.6). An important feature for us was persistent storage of incoming messages to prevent data loss in case of problems such as system crashes or power outages. We intended at first to use a message queue for this task but could not find a software implementation that satisfied our needs. Projects such as Apache ActiveMQ[8] or RabbitMQ[9] did not meet our lightweight or low-complexity requirements (by far). We considered other alternatives such as the Ruby-based Starling[10], developed by the microblogging service Twitter, but even Twitter itself is allegedly moving away from Starling due to performance issues. At the end, we decided to implement a simple yet lightweight and efficient message buffer ourselves, *Pylog*, on top of the Twisted networking framework for Python. Its only purpose is to accept "messages", in our case properly encoded user bookmarks, sent from Pylons via the web interface or the API and reliably log them to file as quickly as possible. Unlike a message queue, Pylog does not need to guarantee FIFO behavior. Instead, we add timestamps to messages and let Hadoop do the sorting during the periodical MapReduce runs[11]. Whenever it is time to start another data aggregation run, we instruct Pylog to rotate its buffer file which is then copied to the Hadoop cluster for processing. We tested the performance of Pylog on two identical machines with a Xeon E5335 2.0 GHz Quad Core CPU and 4 GB of RAM running Ubuntu Linux 8.04 Server Edition with the default Linux kernel 2.6.2419-server. The machines were connected by a switched, full duplex Fast Ethernet network. The average throughput was 14,628 "bookmark messages" per second, more than enough for our setup. If needed, more Pylog instances can be added, and the sum of their buffer files be jointly copied to the Hadoop cluster for the next MapReduce runs.

[6]SQLAlchemy is a SQL toolkit and Object Relational Mapper for Python. Among other things, it allows for a more "Pythonic" interaction with relational databases. http://www.sqlalchemy.org/

[7]http://tokyocabinet.sourceforge.net/spex-en.html

[8]http://activemq.apache.org/

[9]RabbitMQ is an implementation of AMQP, an emerging standard for high performance messaging. http://www.rabbitmq.com/

[10]Starling is a persistent queue server compatible with the memcached protocol. http://rubyforge.org/projects/starling/

[11]Hadoop is *very* good at sorting data. It recently won the Terabyte Sort Benchmark in the record time of 209 seconds on a cluster of 910 nodes in July 2008. http://developer.yahoo.com/blogs/hadoop/

3.6. Data aggregation

In this section, we talk about how individual user bookmarks are aggregated into joint community ratings for given URLs through MapReduce jobs run on the Hadoop framework. We focus our description on how we use Hadoop to compute community ratings, but we also it for other tasks such as web log analysis. We have chosen Hadoop for data aggregation tasks because it allows for linear scaling in terms of data processing, and it is built for use with commodity (rather inexpensive) hardware. If more processing power is needed, it is generally sufficient to just add more machines to the cluster.

Regarding Hadoop alternatives, there exist tools which implement parts of the Hadoop functionality such as the job queue TheSchwartz[12] or the distributed filesystem MogileFS[13]. However, neither of them comes in the neatly integrated and ready-to-use package that is Hadoop. Additionally, the development and adaptation of Hadoop in the wild have been showing a great momentum in the past, particularly after Hadoop became a top level Apache project in 2008. A bonus for the Java-based Hadoop in practice is also its so-called streaming utility, which allows developers to write MapReduce jobs in the programming language of their choice. Finally, having a Hadoop cluster in place enables us to perform tasks that are not directly related to TaggyBear's operation but still important, e.g. log file analysis or fault-tolerant storage of archived data.

We have implemented data aggregation for computing community ratings with two MapReduce jobs that are chained together, i.e. the output of the first job is the input of the second. Please remember that ratings are derived from the tags with which a bookmark has been annotated as described in section 2.2. The role of the first job is data consolidation: It merges multiple submissions or modifications of bookmarks including any derived ratings *of the same URL by the same user* into a single "update". This can

[12]http://code.sixapart.com/trac/TheSchwartz

[13]http://danga.com/mogilefs/

easily happen in our setup because using a message buffer can lead to "pending updates" per definitionem. The role of the second job is the actual aggregation: It combines the ratings of multiple users by URL. Optionally, we could filter out known spammers during this step or promote the ratings of known expert users.

The total time needed for data aggregation can be approximated by the following formula:

$$\begin{aligned} t_{total} = \quad & t_{fs2hdfs}(I, B) \\ + \quad & t_{Hadoop}(I, B) \\ + \quad & t_{hdfs2fs}(O, C) \\ + \quad & t_{TokyoTyrant}(O, C) \end{aligned}$$

where $t_{fs2hdfs}(I, B)$ is the time needed to copy a message buffer file of I bytes containing B bookmarks from the local file system to HDFS[14]; $t_{Hadoop}(I, B)$ is the time needed to aggregate these bookmarks through Hadoop MapReduce jobs; $t_{hdfs2fs}(O, C)$ is the time needed to copy the aggregation output of O bytes containing C community ratings from HDFS to the local file system; and $t_{TokyoTyrant}(O, C)$ is the time needed to insert these community ratings into the hash database.

The times for data import and export, $t_{fs2hdfs}(I, B)$ and $t_{hdfs2fs}(O, C)$ are mainly network-limited, and can be handled by proper network/rack setup.

The time for the actual aggregation via the two described Hadoop MapReduce jobs, $t_{Hadoop}(I, B)$, is influenced by a variety of factors such as the number of Hadoop data nodes (which serve HDFS data) and tasktracker nodes (which process data) in the cluster, job parameters such as the number of reduce jobs to be run, and other factors such as the size of intermediate data. Figure 5 shows benchmarking results for a cluster of four machines connected by a switched, full duplex Fast Ethernet network. Each machine has an Intel Xeon E5335 2.0 GHz Quad Core CPU, 4 GB of RAM, hardware RAID5 data storage, and runs Ubuntu Linux 8.04 Server Edition with the default Linux kernel 2.6.2419-server. We used Hadoop version 0.18.0 released in August 2008 for the benchmark. The results are averaged over several runs with the slowest and fastest results being removed from the samples.

Interestingly, we can see that the number of bookmarks aggregated per second *increases* with number of input bookmarks. This is mainly due to two reasons. First, there is a rather fixed overhead for starting and running MapReduce jobs with small input data. Second, if the input data is not sufficiently large, less cluster nodes are used in general for executing the job. In other words, more cluster nodes

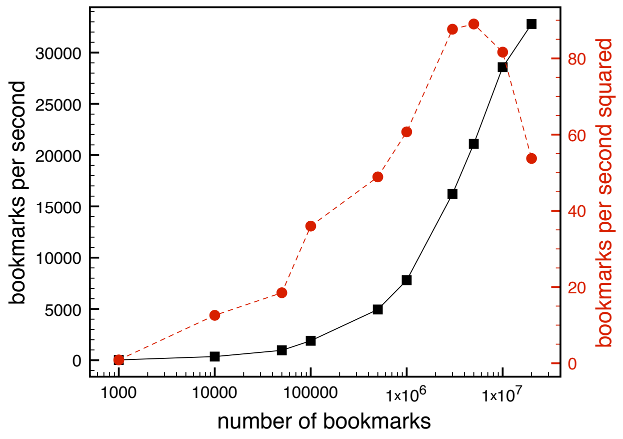


Figure 5. Benchmark for aggregation of public user bookmarks into community ratings through Hadoop MapReduce jobs. The solid black line and the red dashed line show the bookmarks per second and bookmarks per second squared, respectively. Note the logarithmic scale of the X axis.

are only activated when needed but will then lead to higher total throughput. Still, the time to process 100,000 bookmarks is less than one minute. At some point, the "acceleration" that Hadoop gets from increasing amounts of input bookmarks starts to slow down until the performance actually degrades (relatively speaking). In our benchmark, this inflection point is approximately at 5,000,000 bookmarks. Most likely, this is caused by limitations in network I/O – the cluster nodes cannot receive data fast enough and end up idling, waiting for data. We expect that switching from Fast Ethernet to Gigabit Ethernet would improve the benchmark times. Nonetheless, the aggregation results are more than satisfying for our needs: The cluster aggregates 1 million new/updated bookmarks in about two minutes, and 10 million bookmarks in less than ten minutes. Considering that the most popular bookmarking service in the Internet today, Delicious.com, received about 7.5 million bookmarks *in one month* in December 2007 according to the study of Wetzker et al. [12], we seem to be well-positioned in terms of data aggregation.

Finally, $t_{TokyoTyrant}(O, C)$ is the time needed to insert or update community ratings into the hash table through Tokyo Tyrant. A so-called PUT operation is equivalent to adding or overwriting an entry for a URL in the hash table.[15] We measured 8,372 PUTs/s on average on the same

[14]This includes the time needed for replication of data chunks to multiple nodes as configured by Hadoop's `dfs.replication` parameter. The default replication value is 3.

[15]Generally, we have to query the caches or the hash table itself first in order to retrieve the current (i.e. outdated) community rating of a given URL in order to perform correct updates. Here though, we are more interested in the *write* speed, i.e. PUTs, particularly because a write is more

hardware when using the Python memcached module as client. If needed, the throughput could be increased by using Tyrant's C/C++ API.

3.7. Caching

Caching is an important technique to reduce server load and improve scalability. In our setup, we use server-side and client-side caching at various places, some of which are described in the following paragraphs.

On the server side, we employ the integrated caching functionality of Pylons' components. For the web interface and the API, we make use of the built-in caching options of the Mako template engine to cache generated content where needed, e.g. the TaggyBear front page or RSS feeds. Alternatively, a reverse proxy such as Perlbal[16] could be used to cache generated web content[17]. For the time being though, we use a reverse proxy in front of Pylons only for enabling controlled and secured access from the outside web to the Pylons server instance.

RDBMS queries to the MySQL database(s) are also cached. Pylons provides a convenient interface to memcached[18], where we cache the results of expensive (slow) SQL queries. We also use MySQL Proxy[19] for query analysis, and are currently testing it for failover and transparent load balancing of databases.

For the client side, we add ETag headers [4] to generated web pages and API responses where appropriate. ETag only helps if the entire page can be cached, and it can prove difficult to set up correctly in a load balancing scenario where a client may request the same content but get a response from different servers on each request. When used properly though, ETag allows compatible web browsers – and HTTP clients in general – to perform client-side caching of web pages, thus further reducing server load.

The TaggyBear browser add-on benefits implicitly from ETag support in Firefox, but the add-on also comes with its own TaggyBear-specific, in-memory caching functionality for API requests in particular. The latter is needed mainly due to GUI interactions. For example, the add-on must handle tab switches in Firefox for processing and updating the currently shown web page (e.g., showing the warning popup in figure 1), and user interactions such as tab switches should not result in unnecessary TCP connections – like asking whether the client-side ETag is still current – to the TaggyBear service.

expensive than a read operation in our case.

[16]Perlbal is a Perl-based reverse proxy load balancer and web server. http://danga.com/perlbal/

[17]On a related note, it is often helpful to use a "regular" web server such as lighttpd, nginx, or Apache HTTP Server to handle static web content such as images or CSS files.

[18]Memcached is a distributed non-persistent object caching system. http://www.danga.com/memcached/

[19]MySQL Proxy is a simple program that sits between a MySQL client and MySQL server. It can monitor, analyze or transform their communication. http://forge.mysql.com/wiki/MySQL_Proxy

4. Related Work

There exists a couple of community-driven, security-related services in the Internet that work similarly to TaggyBear or have similar goals, some of which will be described below.

Vipul's Razor[20] is a signature-based, distributed network for email spam detection and filtering. Basically, users can collaboratively report spam and non-spam messages to the service. Razor computes the signature for each message, i.e. a checksum of the body's content, and stores information about the message in its database. Other users can then query Razor and its database whether a given message is known to be spam. The role of signatures of emails for Razor and the MD5 hashes of URLs for TaggyBear serve similar purposes. Both email signatures and URL hashes are used to detect identical "objects", where object identity for Razor is "same message body" (i.e. actual content) and "same URL" for TaggyBear (a pointer to content).

OpenDNS operates PhishTank[21], a free service where users can collaboratively share information about phishing websites, i.e. websites that attempt to trick users into providing personal information such as bank account data. PhishTank allows users to submit and verify the alleged phishing status of a web page, both via the PhishTank web interface and through an API.

The design of TaggyBear differentiates from services like Razor or PhishTank particularly in two areas. First, the "opinion" of the user community (i.e. community ratings) is not restricted to a binary feature, i.e. *spam* and *non-spam*, *phishing* and *non-phishing*, or *like* and *dislike*. Instead, TaggyBear leverages the full folksonomy and tagging vocabulary of its users. The data aggregated from user input is multi-dimensional in the sense that a user can freely decide whether he's interested in filtering by, say, the `phishing` dimension, `porn` or `webmail`. Second, one of the main goals of TaggyBear was to build a system that is not only about security and filtering (which "restrict" the user's Internet experience in one way or the other even though its done on behalf of the user), but one that also provides features that actively help the user in his everyday tasks by *enriching* his Internet experience, or features that are simply fun to use. Just the fact that TaggyBear is based on a social bookmarking service – with all its benefits for end users – extends TaggyBear to being more than a simple web filter. Another example is the web search personalization component we have developed [10]. Here, we leverage and apply the same user input to a different problem domain which is

[20]http://razor.sourceforge.net/

[21]http://www.phishtank.com/

information retrieval. This allows a user to benefit from his bookmarking activity not only with regard to bookmarking per se but also by improving his search experience.

5. Conclusion and Discussion

The current architecture of TaggyBear seems to meet its requirements adequately. Without free and open source software and the support of the people creating and using it, this would have been hardly possible. On the other hand, we also made experiences that were not so pleasant, and which we mention here as part of our case study report. For example, the web framework Pylons has been a moving target. Quite often, new (even minor) releases deprecated part of the functionality, or ended up breaking existing code right away. Most notably, the authentication & authorization component, AuthKit, and the database components come to mind in this regard, though problems caused by the latter can partly be blamed on changes to SQLAlchemy. Incomplete, outdated, or confusing documentation also posed problems, particularly for Hadoop, which resulted in several blog articles of the first author about how to setup and use Hadoop, part of which have been integrated into the Hadoop wiki. Luckily, the situation has recently been improving for Hadoop, meaning that more people outside of the Hadoop developer community are able to benefit from the software's great functionality.

For the future, we have already several new developments for TaggyBear in mind. For example, to refine the analysis of user bookmarking behavior in order to further improve the quality of community ratings. Having a powerful data processing tool like Hadoop at hand is a big plus in this regard. Of course, we also want to extend the benchmarking and profiling of TaggyBear, and improve the areas where proper scalability might become an issue, while avoiding premature optimization as it is "the root of all evil".

We hope that the reader could find our descriptions and reasoning interesting and, in particular, helpful for her or his own practical programming and development projects.

References

[1] M. Ames and M. Naaman. Why we tag: motivations for annotation in mobile and online media. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 971–980, New York, NY, USA, 2007. ACM.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[3] R. J. Deibert, J. G. Palfrey, R. Rohozinski, and J. Zittrain. *Access Denied: The Practice and Policy of Global Internet Filtering*. The MIT Press, Cambridge, Massachusetts, 2008.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616: Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999. Network Working Group, W3.

[5] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[6] C. Henderson. *Building Scalable Websites*. O'Reilly Media, Sebastopol, CA, USA, 2006.

[7] C. Marlow, M. Naaman, D. Boyd, and M. Davis. Ht06, tagging paper, taxonomy, flickr, academic article, to read. In *Proceedings of HT '06*, pages 31–40, 2006.

[8] M. G. Noll and C. Meinel. Design and anatomy of a social web filtering service. In *Proceedings of 4th Int'l Conference on Cooperative Internet Computing*, pages 35–44, Hong Kong, 2006.

[9] M. G. Noll and C. Meinel. Authors vs. readers: A comparative study of document metadata and content in the www. In *Proceedings of 7th Int'l ACM Symposium on Document Engineering '07*, pages 177–186, Winnipeg, Canada, 2007.

[10] M. G. Noll and C. Meinel. Web search personalization via social bookmarking and tagging. In *Proceedings of 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference, Springer LNCS 4825*, pages 367–380, Busan, South Korea, 2007.

[11] M. G. Noll and C. Meinel. Exploring social annotations for web document classification. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2315–2320, Fortaleza, Ceara, Brazil, 2008. ACM.

[12] R. Wetzker, C. Zimmermann, and C. Bauckhage. Analyzing social bookmarking systems: A del.icio.us cookbook, 2008. To appear in ECAI 2008.

[13] J. Zawodny and D. J. Balling. *High Performance MySQL: Optimization, Backups, Replication, Load Balancing & More*. O'Reilly Media, Sebastopol, CA, USA, 2004.