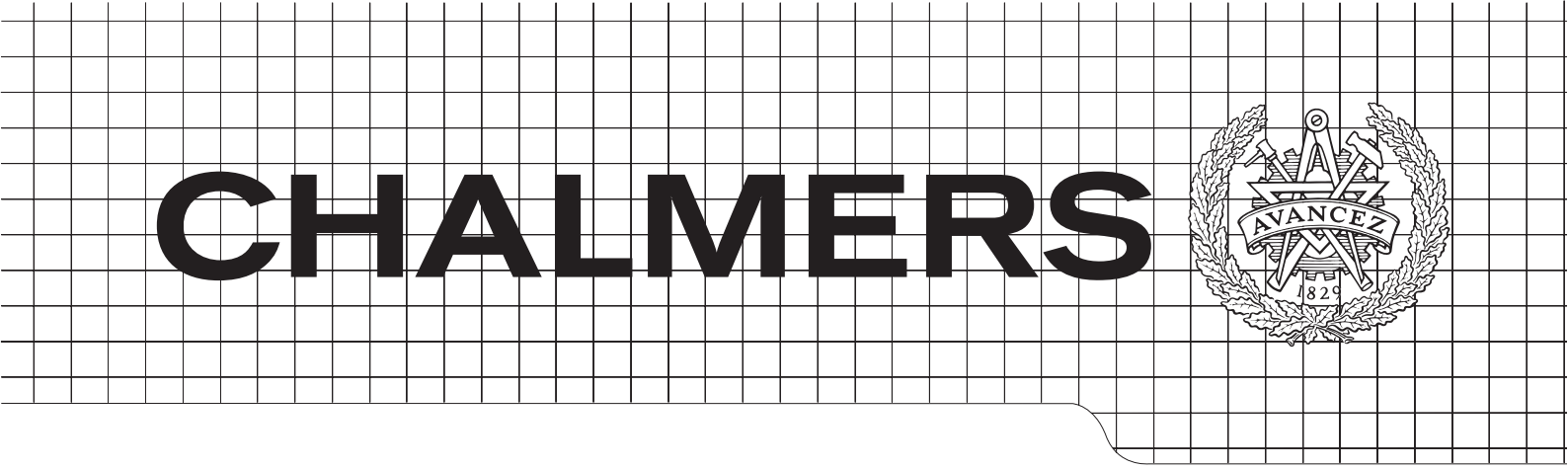


Ranked Activity Streams

Master's Thesis in Computer Science: Algorithms, Languages and Logic

OLA HOLMSTRÖM

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2014
Master's Thesis 2014

Ranked Activity Streams
Design and Implementation of an Activity Engine

OLA HOLMSTRÖM

Examiner: GRAHAM KEMP

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2014

Abstract

A common component of social networks today is an activity stream, which is a list of recent activities performed by users. The purpose of an activity stream is to give users an easy way to keep up to date with other users. Activity streams can often grow very quickly and become so large and update at such a fast pace that they become unusable for the purpose they were intended. To ameliorate this problem an activity stream can be ranked, giving each activity a score based on its relevancy, bubbling relevant activities to the front or hiding activities that are below a certain threshold. Doing this will in theory help users keep up to date with activities that are relevant. The aim of this project is to create a system that manages a ranked activity stream called ”FanFlow” on the photo-sharing website YouPic.com. The ”FanFlow” activity stream consists of photos uploaded by YouPic.com users which a user is a fan of. The first part implements a system termed an activity engine which routes each photo to the correct ”FanFlow”. The second part is to implement and evaluate ranking metrics on this activity stream.

Acknowledgements

The author would like to thank my supervisor at YouPic.com Navid Razazi and my examiner Graham Kemp at Chalmers for providing guidance and input during the project.

Ola Holmström, Gothenburg 16/5/14

Contents

Chapter 1

Introduction

In this chapter, the main topic of the project and the motivation behind it are introduced. This starts off with a presentation of the problem background in Section 1.1. Following this, the problem formulation is given in Section 1.2. Finally, an outline of the rest of the report is given in Section 1.3.

1.1 Background

Modern social media has made keeping up to date with the activities of people seamless[1]. If one of your friends on Facebook changes their relationship status it will immediately show up in your news feed, if someone you follow on Twitter tweets, your cellphone might buzz with a notification that this has occurred. The information one can monitor is virtually limitless and available the instant it happens.

This convenient model of sharing information is based on subscribing to a person's activities and three of the most popular social networks at the time of writing (Facebook, Twitter and Google+) use a variation of it. For example on Facebook you have the "News Feed" which is the activities of your friends. These collections of activities are usually called activity streams[2] but go by other names such as Timelines (Twitter) and Feeds (Google+ and Facebook).

Activities are in their simplest form small phrases consisting of an actor, a verb, an object and a target[2] an example would be "Alice liked a photo on Facebook" where Alice is the actor, liked is the verb, and photo is the object and Facebook is the target. What is and what isn't an activity is wholly defined by the service for example the only activities on Twitter are tweets and retweets unlike Facebook which counts everything from updating your telephone number to commenting on a photo as an activity. When activities are collected they make an

activity stream. A good activity stream should form activities into a narrative that is easy for the reader to follow and understand.

While activities are lightweight and not a lot of data is required to represent them the sheer amount that can be produced by a user and the need to deliver them to other users makes handling them a significant scalability problem. On a large social network the average number of receiving users can be as high as 100 000[3]. Even on a small service all the tiny nuggets of information generated by every user reflected by their number of followers can become a torrent that is difficult to manage. Ensuring that activities reach their correct destination in a timely manner may sound like a simple problem but at scale the system can get overloaded. An overloaded system might see increasing delays and in the worst case might start dropping activities.

A system that manages activity streams is called an activity engine and the first part of this project will be describing and implementing one. Often an activity engine is specifically tailored to a service's needs, for example Twitter uses the in memory database Redis to store activity streams so that they can be retrieved in milliseconds[4]. Other services might not prioritize fast retrieval but instead focus on things such as space efficiency or ease of filtering the activity stream, for them the internals of their activity engine might be radically different.

Just like a system can get overwhelmed by the amount of activities so can a user receiving them[5], logging into their account to see a hundred, a thousand or ten thousand updates since they last used the service, when this happens there is need to either highlight or bring to the top activities that are interesting[6]. Both Facebook and Google+ use some sort of ranking that is not solely chronological. While Google+ ranking has not been clearly defined, Facebook has published theirs which they call "EdgeRank"[7]. The main idea behind EdgeRank is to rank activities based on the affinity between the receiving user and the creating user, the thought being that if a user interacts positively with another's users activities they would like to see more of them. Ranking activities can have unintended consequences and designing them is as much psychology as it is engineering[8].

While the usefulness for the user of ranking activity streams can be debated (Twitter's Timelines are strictly chronological) doing it is a difficult problem because of the amount of activities that need to be ranked. Ranking algorithms need to be designed to be fast so that they don't add unnecessary delays to the system, usually relying on caching for important variables such as affinity score in "EdgeRank". Creating ranking algorithms that are both fast and increase user engagement is the second part of this project [9].

1.1.1 YouPic.com

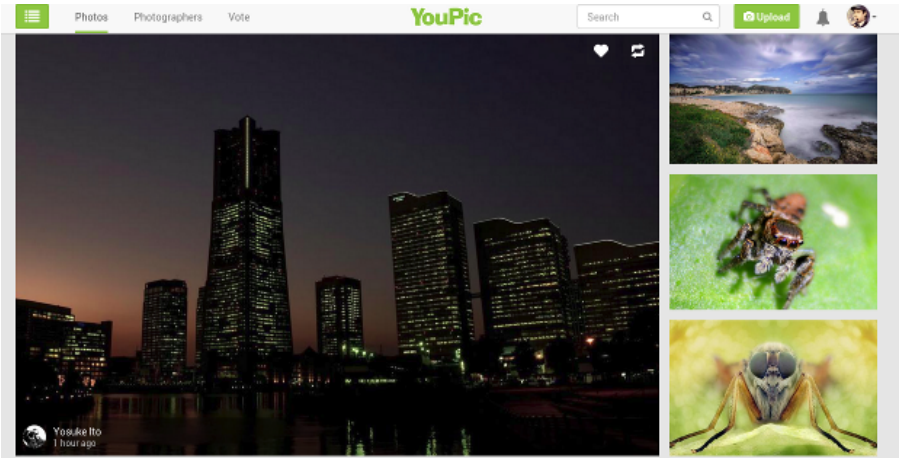
YouPic.com is a photo-sharing website for professional and traveling photographers. It has since 2012 been in Chalmers Innovation's incubator program, at the time of writing YouPic.com has 30000 users and 200000 uploaded images.

On YouPic.com users create relationships between each other by becoming fans. Like 'follow' on Twitter becoming a fan of a user is unidirectional and doesn't require the other user to accept the relationship.

Users show their appreciation for photos by making them a 'favorite'. A favorite is either on or off and a user can only favorite a photo once.

The activity stream FanFlow on YouPic.com consist of the photos from users of whom that user is a fan. In this project it will be ranked by one of four algorithms described in chapter 3.

Figure 1.1: YouPic.com's FanFlow activity stream, showing photos from a selection of users.



1.2 Problem formulation

The purpose of this project is to describe, implement and analyse a system for managing ranked activity streams (an activity engine with a ranking function.)

Main questions this project tries to answer are:

- What are the methods an activity engine needs to implement?

- How are these methods best implemented in context of YouPic.com?
- What is a good metric for ranking activities in the FanFlow activity stream?
- What are the pros and cons of the implemented activity engine?

To answer these questions, first, the abstract methods of an activity engine are described and defined, then concrete implementations of these methods are done. This leads into an analysis of the chosen implementation, finally a conclusion with examples of extensions.

1.3 Related work

- Berkovksy et al's paper *Personalized Network Updates: Increasing Social Interactions and Contributions in Social Networks* [10] provides an introduction to ranked activity streams and an implementation of a ranking algorithm. They develop a personalized model for predicting the importance of activities in a social network. Their ranking algorithm takes into account the strength of the relationship between users and evaluates the ranking algorithm based on how often the first activity in the stream is clicked. Their conclusion is that a stream ranked with their algorithm successfully promoted activities that were relevant to the user and made information easier to find.
- *Personalized activity streams: sifting through the river of news by*[11] written by Guy et al gives a good overview of the problem of information overflow in activity streams. The paper gives an explanation of the higher importance of freshness in the relevancy of activities compared to other recommendations. They introduce the concept of *throughput*, defined as the number of activities their system produces over given period of time as a metric for the effectiveness of activity stream recommendations. One the filtering algorithms they describe is a called a *Stream-based profile*, which is built through analysing the user's own activity stream, this is similar to some of the ranking algorithms used for the FanFlow in chapter 4.

1.4 Contribution of project

The key contribution of this project is an implementation of an activity engine for YouPic.com together with an analysis of it. Another con-

tribution is the evaluation of different algorithms for ranking activity streams.

1.5 Outline of report

The report is structured as follows:

- Chapter 2 presents definitions of the activity engine methods. The definitions aim to clearly and in a concise way describe why the methods are needed and what they do.
- Chapter 3 describes the implementation of the activity engine. In this chapter information about the data structures and the ranking algorithms is found.
- Chapter 4 presents an analysis of the implementation in chapter 3 with both empirical measurements of performance and theoretical analysis of the algorithms and data structures.
- Chapter 5 a discussion of the implementation and the problem formulation.
- Chapter 6 concludes the report by evaluating what has been done and if it has been successful with a section on further work.

Chapter 2

Definitions

In this chapter the behavior of four core methods that make up an activity engine are defined. These four methods will be given a concrete implementation in the following chapter. The methods are simple so that the trade offs in the implementation will be clear. The aim is to be clear and concise about what these methods do and what implications they have for the implementation.

2.1 Put

The first and most fundamental method of an activity engine is the *put* method which adds an activity to an activity stream. It takes two arguments x (an activity) and A (an activity stream) and returns an activity stream A' which must include x and all activities in A. The *put* method should prevent duplication of activities in the stream, if x is in A then put(x, A) should return A.

A good implementation of the *put* method ensures good write performance. If the activity engine should be optimized for writing the *put* should be as fast as possible.

2.2 Get

Reading of activity streams is done by the *get* method. The method takes a pivot activity x where it begins reading, a natural number n which is the maximum number of activities it will read and an activity stream A. The reason for using a pivot activity instead of a numerical offset is to avoid duplication on sequential reads. This problem can be illustrated by imaging reading the activities from one to five and then later reading the activities five to ten, if the range one to five has changed during the intermittent period it is likely that five to ten will

contain duplicates from the previous read of one to five. If the pivot activity x does not exist in A then the returned slice should be empty.

The implementation of *get* affects both write and read performance of an activity engine as activity engines can choose to implement combined activity streams as unions which greatly increases the complexity of *get*. This will be further explored in the next chapter.

2.3 Route

When an activity is created it does not yet belong to any activity stream. The task of putting it in the right activity streams falls to the *route* method. The *route* method takes two arguments, x which is an activity and As which is a set of activity streams. *route* works by applying $put(x, A)$ to each activity stream in the set As. For example defining a null route (no activities are added to any activity streams) is simply passing an empty set as As.

2.4 Rank

The final method is *rank* which given an activity x returns an element which has an ordering. *rank* is used for determining the position of an activity in an activity stream. For example a chronological activity stream would define *rank* as $rank(x) = time(x)$ where the function $time(x)$ returns the timestamp of the activity x.

Chapter 3

Implementation

This chapter will first explain the motivations behind the activity engine and what YouPic.com wants to achieve with it, then go through a concrete implementation of the activity engine. The activity engine will be implemented using the abstract methods from the previous chapter.

3.1 YouPic.com's FanFlow

YouPic.com' FanFlow is an activity stream consisting of content by users a user is a fan of. It is analogous to Twitter's "Home Timeline" substituting tweets for photos and followers for fans.

With the FanFlow, YouPic.com wishes to increase engagement with the content on its site, the idea being that a user will be more interested in content from users they are fans of because they have already shown their approval of these users.

In the FanFlow activity stream the only activities are of the type "X uploaded a photo on YouPic.com". In standard activity format "X" is the subject, "uploaded" is the verb, "photo" is the object and YouPic.com is the target[2].

To try to maximize engagement with the photos the FanFlow activity stream is ranked by one of four algorithms. The algorithms are assigned equal groups of user and are compared by the number of favorites in each set to see which one increases engagement the most. One of the algorithms will be a control which is just a chronological ranking.

3.2 Activities

The only activities in the FanFlow are photos, and photos are stored by YouPic.com in the relational database MySQL. All photos have an

integer that uniquely identifies them in the database which is called a primary key, the photos primary key is its *photo_id*.

Using the *photo_id* in the FanFlow activity stream is a compact way to reference photos without needlessly copying information, although doing this adds an extra step of having to look up the *photo_id* in the database. Using this approach has the extra benefit of reflecting changes to the photo information immediately.

3.3 Fan In and Fan Out

In broad terms there are two main ways to implement activity streams, both of which have distinct strengths and weaknesses.

The first approach is called ”Fan In”[4]. In a ”Fan In” system activities are stored together with metadata that determines which activity streams they belong to, activity streams themselves are not explicitly stored but computed on demand. The ”Fan In” name comes from the fact that when a read is requested the activities are pulled together by some function into an activity stream. This type of system can be naturally implemented in a relational database by issuing a query which acts as the ”Fan In” function. Usually this query is a join on the different activity streams of users.

The two main advantages of ”Fan In” systems are that they are fast to write to and that they are space efficient. Writing a ”Fan In” system is fast because there is no need to worry about routing the activity at that time. The space efficiency comes from the fact that there are no duplicate activities.

The weakness of ”Fan In” systems are that reading activity streams is expensive because they have to be computed by a function every time, although this can be ameliorated to a degree by caching the result. Caching the result though does not help with the fact the activity streams need to be rebuilt every time a new activity is added. Another weakness is that of scalability, in a distributed ”Fan In” system the activities may be scattered across several machines making the ”Fan In” function very expensive as it might have to query them all.

The second way to implemented activity streams is called ”Fan Out”[4]. A ”Fan Out” system stores each activity stream separately and when an activity is created it is copied into each one it belongs to. The reason why it’s called ”Fan Out” is because the activities are said to fan out into each activity stream when they are created.

”Fan Out” system are good because they allow for low overhead when reading activity streams. This is because the activity streams are already assembled and do not need to be built on the fly when they

are read. ”Fan Out” systems is also scalable as the activity streams are independent and all the activities needed are found in the activity stream data structure. This makes them efficient to distribute to several machines as a read only has to determine what machine the requested activity stream is on.

The drawbacks with ”Fan Out” systems are that they are expensive to write to and that they take up more space than a ”Fan In” system. The cause of the expensive write is that each activity needs to be put in each activity stream directly instead of this being done when they are read. The space inefficiency comes from the fact an activity might be duplicated in several activity streams, instead of as in the ”Fan In” approach where an activity resides in only one place.

To decide which approach should be used when implementing the activity engine for YouPic.com’s FanFlow there are some question that need to be answered:

- Will the activity stream be read more than it is written to? The FanFlow is meant to be the main way that users explore photos on YouPic.com and therefore it is expected to be read quite frequently. Another a very important factor to consider is the retrieval time that a system might have. Studies have shown that users respond poorly to slow retrieval times[12] or to retrieval times that they cannot predict[13]. Because a ”Fan In” system uses a function to pull together activities for an activity stream the retrieval time might be fast or slow depending on the current load on the machine it is running on. A read operation in ”Fan out” system should both be faster and less variable in it’s retrieval time.

- How important is scalability? YouPic.com’s ambition is to grow as much as possible so scaling the activity engine is an important concern. Using a ”Fan In” system would make scaling trickier than using a ”Fan Out” system. In a ”Fan Out” system machines could be added incrementally to store the new users’ activity streams.

Taking into account the way in which the ”FanFlow” activity stream is to be used the decision has been made to use a ”Fan Out” approach to implementing the activity engine.

3.4 Activity Streams

The key to performance in a ”Fan Out” system is choosing a good data structure to represent activity streams.

The database used to store activity streams will be the in memory advanced key value database Redis. Redis is already used by YouPic.com for several other services and is ideal for activity streams because it provides several data structures that can be used to implementent them.

As activity streams are simply collections of activities a natural data structure to represent them is a list. A list is a data structure that can represent a collection of arbitrary length. Redis has an implementation of lists as doubly linked lists. Doubly linked lists can be traversed from either the start or the end, but cannot be randomly accessed.

The two main advantages of using a list is that it can be of arbitrary length and that inserting elements at the ends is cheap.

Two disadvantages of using lists is the fact that ranking is cumbersome and that a list can contain duplicate activities. Ranking is difficult because inserting anywhere but the two ends of the list requires extra steps, so if an activity of lower rank than the current head comes in a computation needs to be done to find where it should go in the list. The only ranking that works well with lists is a chronological one, where activities are simply added to the beginning of the list. The other problem is that lists can contain duplicates and therefore a prudent *put* algorithm needs to check whether the list already contains the activity before inserting it, and checking if a list contains an activity is an expensive operation. Checking if an item is a member of a list is a $O(n)$ operation as every item in the list needs to be compared.

The problems with using lists to store activity streams can be solved by using a set data structure. Redis provides two of these, one is an unorder set and the other is sorted set. The sorted set is the better choice because it solves both the problem of ranking and the duplicated activites. An entry in Redis sorted set consist of an item and a score, the item in our case will be the *photo_id* and score is the rank of the corresponding activity. Pseudocode for get and put can be seen in figure 3.2 and 3.1.

```
function put(x, A)
    score := rank(x)
    return redis.ZADD(x, score, A)
end
```

Figure 3.1: Put method.

```
function get(x, n, A)
    i := redis.ZREVRANK(A, x)
    return redis.ZREVRANGE(A, i, i+n)
end
```

Figure 3.2: Get method.

3.5 Routing

The activities in the FanFlow are the photos uploaded by the users a user is a fan of. The fan relation is unidirectional and is stored as a $from\_id$ and a *to_id* in a relational database. This makes YouPic.com's social graph a directed graph where the nodes are users and the edges are fan relationships. The routing of activities is based on the in-degree of the uploading user. In a directed graph the in-degree is the number of incoming relations to a node.

When a user uploads a photo the *route* method of the activity engine is called with the *photo_id* as an argument. The algorithm for *route* works by first retrieving all of the fans of the user that has uploaded a photo and then adding *photo_id* to their FanFlow activity stream by using the *put* method. The route algorithm is described in psuedocode in figure 3.3.

```
function route(x, As)
    for A in As
        put(x, A)
end
```

Figure 3.3: Route method.

3.6 Ranking

The rank of an activity should be based on the relevancy of the activity to the consuming user[14]. The crux is defining what relevancy is in the context that the activity stream is used. One nearly universal factor of relevancy though is that it is timed based: an activity that happened recently is more relevant than an activity that happened a year ago. The second aspect of relevancy is the importance of the activity. Importance is of course a nebulous concept but an illustration of it can be made

by thinking about activities on Facebook, what is more important your cousin twice removed liking a picture of a cat or your partner changing his/her relationship status? Of the two aspect of ranking an activity (time and importance) importance is the more complex construction but time is the more important factor. For example Twitter's aptly named Timelines are only ranked in chronological order.

3.6.1 EdgeRank

A good example of how an actual activity ranking algorithm works is Facebook's News Feed ranking algorithm EdgeRank[7]. EdgeRank takes three inputs to rank an activity or edge as Facebook calls them, affinity score (u_e), edge weight (w_e) and time decay (d_e).

The affinity score is a quantification of the relationship between the user receiving the activity and the user the creating it. The score becomes higher the more the receiving user interacts with the creating users activities. So if the receiving user has in the past commented or liked activities from that user the affinity score will be high and if they have ignored their activities the score will be low.

Edge weight is determined by the type an activity has. Facebook has several activity types such as statuses, likes, comments and photo uploads. Usually the activities that take more effort to create have a higher edge weight so a comment will be scored higher than a like for example.

The last input to EdgeRank is time decay which is calculated as $1/t$ where t is time since activity was created. This input keeps the News Feed from becoming stale with lingering old activities.

3.6.2 Static and continuous ranking

An important question to ask about a ranking function is whether it is static or continuous.

A static ranking function only needs to be applied once to an activity when it is created. Ranking activities by time is a simple static ranking function because the time the activity was created never changes.

A continuous ranking function on the other hand needs to be update every time the score changes. For example if a ranking function was defined in terms of the number of Favorites a photo has it would need to be reapplied every time the number Favorites on a photo changes.

As continuous functions are computationally expensive the ranking functions that will be used will all be static. This circumvents the problem of having to recalculate the entire activity stream every time a new activity is added.

3.6.3 Time as the base of ranking

Almost all ranking algorithms take into account how long ago an activity was created. EdgeRank does this with its time decay input which is defined as $1/t$. The problem is that "time since activity was created" needs to be updated to accurately reflect time since the activity was created and having to recalculate an entire activity stream every time an activity is added is computationally expensive.

As time is the foundation of all the ranking algorithms that are going to be tested, expressing the importance part of the ranking score in terms of time would make it so the activity only has to be updated whenever the importance changes. That means that an activity will "time-travel" forward because the importance score will be added to the time. This means that an activity with a high importance might go forward by a day, while a low ranking one only by a minute or two.

3.6.4 Ranking 1: Chronological

The first ranking algorithm is a simple chronological ranking were the score is the time (the timestamp) when the activity was created. This algorithm will act as control for the other ranking algorithms we are going to test. Psuedocode for the chronological ranking is in 3.4.

```
function rank(x)
    return x.timestamp
end
```

Figure 3.4: Chronological ranking.

3.6.5 Ranking 2: Favorites

Another way to to rank activities is to look at the history of the viewing user and whether that user has positively interacted with the creating users' activities before. On YouPic.com a way to positively interact with photos is to favorite them. With this we can create a ranking that gives priority to photos from users that have received many favorites from the viewing user. Psuedocode for the favorite ranking algorithm can be found in figure 3.5.

3.6.6 Ranking 3: Category

The third ranking algorithm investigated uses the number of favorites per photo category instead of favorites per user. The logic being that a

```
function rank(x)
    ratio := favs(x.user) / total_favs
    return (24*60*60*ratio) + x.timestamp
end
```

Figure 3.5: Favorite ranking.

user will be interested in photos that have the same category of photos they have favorited before. Psuedocode describing this ranking algorithm is in figure 3.6.

```
function rank(x)
    ratio := category(x.user) / total_favs
    return (24*60*60*ratio) + x.timestamp
end
```

Figure 3.6: Category ranking.

3.6.7 Ranking 4: Reputation

On YouPic.com users are given a rank using a metric called reputation which is based on a modified version of the Hirsch index[15].

The Hirsch index is an academic impact metric which is designed to measure the productivity and the impact of a scholars published works[16]. To compute it a list of the papers a scholar has published and the number of citations to each paper is needed. Then the Hirsch index is defined as the number of papers h that have been cited at least h times in other papers.

The Hirsch index is adapted to YouPic.com by substituting photographs for papers and favorites for citations. The metric derived is called ”reputation” and stands for the how much impact a photographer has on the YouPic community.

The last ranking uses the ”reputation” metric to rank photos. The importance of a photo on YouPic.com is determined by the reputation of the photographer. The idea being that a photo uploaded by somebody with a high reputation will be of higher quality. Psuedocode for this ranking can be found in figure 3.7.

```
function rank(x)
    rep := reputation(x.user)
    return 60*rep + x.timestamp
end
```

Figure 3.7: Reputation ranking.

Chapter 4

Analysis

In this chapter an analysis of the implementation from the previous chapter with both measurements of performance and theoretical analysis of the algorithms and data structures. In the first section we go through each method of the activity engine and determine its complexity class. In the next section each method is subject to performance measurements with different inputs.

4.1 Complexity

Determining the complexity class of each method is important because this can tell us about how the activity engine will work at scale. The time complexity of the activity engine is mostly determined by the data structures we have chosen to use in the Redis database. The complexity of different Redis operations can be found in the Redis documentation[17].

The implementation of the *put* method maps directly to the Redis command $ZADD$ which according to the Redis documentation has a time complexity of $O(log(N))$ where N is the number of elements in the activity stream.

To make *get* comply with its definition the index of pivot element x must first be found using the Redis command $ZREVRANK$, which has the time complexity $O(log(N))$ where N is the number of elements in the activity stream. Using the index of the pivot element the correct slice of the activity stream is retrieved by using the Redis command $ZREVRANGE$ with the index of x as an argument and $x + n$ as the offset. The $ZREVRANGE$ command has the time complexity $O(log(N) + M)$ where N is the number of elements in the activity stream and M is the number of elements returned which is necessarily always less than n. Putting these two together the complexity of the

implementation of *get* is $O(log(N) + M) + O(log(N))$.

The *route* method iterates over all the fans of the user that uploaded activity x applying the *put* method to each FanFlow activity stream. This loop has the time complexity $O(M)$ where M is the number of fans the user that uploaded x has. As the *put* method has time complexity $O(log(N))$ the total time complexity is $O(M * log(N))$.

The *rank* method is $O(1)$ as it is simply an arithmetic between the reputation integer and the time x was uploaded. Although the computation of the reputation is $O(N) + O(M)$ where N is the number of photos and M the number of favorites.

4.2 Memory

The amount of memory the activity engine uses is determined by how many references to activities are stored in the activity streams. Knowing how much memory the activity engine is going to use is crucial as the Redis database is strictly an in memory database.

A quick estimate of how many references there are in the system is just taking the median number of relationships each user has and the median number of photos each user has uploaded. Having these two numbers an estimation can be calculated by the simple formula $R * P * U_tot$ where R is the median number of relations, P the median number of photos each user has uploaded and U_{tot} the total number of users in the system.

The maximum number of references happens when the social graph between users is fully connected i.e. everyone is a fan of everyone else. In that case the number of references to activities is $(U_tot - 1) * P * U_tot$ where P is the median number of uploaded photos per user and U_tot is the total number of users in the system.

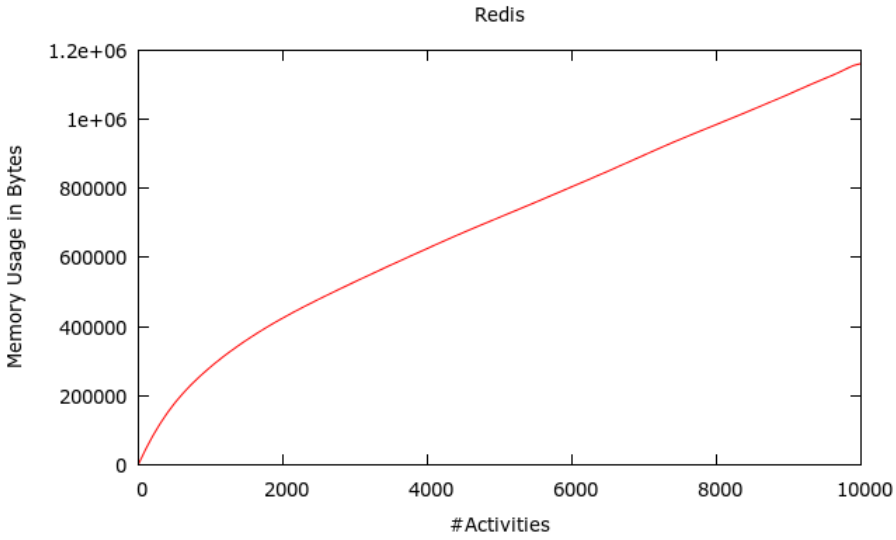
Because Redis is an in memory database unbounded growth of activity streams can be dangerous as when the database runs out of memory it becomes unresponsive or shuts down. Therefore on production systems the length of activity streams is usually capped at some fixed number. In this case the maximum number of references in the system is $L * U_tot$ where L is the maximum length of an activity stream. This is beneficial because the cap L can be decreased as the number of users increase, thus thwarting memory problems.

4.3 Performance

In this section the performance of the implemented activity engine will be evaluated using test data. The performance is divided into two

parts: one is the time it takes to route the activities and the other is the amount of space that is taken up by the stored activities. The script that evaluates the activity engine takes three arguments: the number of users in the system, the number of relationships in the system and the number of activities to be published.

Figure 4.1: Space performance with 1000 users and 1000 relationships



In figure 4.1 we can see the progression of memory usage when we increase the number of activities in the system. As we can see from the graph the memory usage is linear with the number of activities. The number of users in the system is a 1000 and the number of relationships are 1000 making the median *outdegree* of the social graph 1.

Figure 4.2 shows the usage of memory when we increase the number of fan relationships in the system. Just like when we increase the activities, the memory usage increases linearly as predicted by the analysis of the system. The number of users in the system is 1000 and the number activities created in the system is 1000. The space consumed increases because the activities are copied into each users activity stream.

Figure 4.3 shows the same test as in figure 4.1 but instead of space the time it takes to route the activities is measured. The time to route the activities in the system increases linearly.

Figure 4.3 shows the same test as in figure 4.2 with the time measured instead of the space.

Figure 4.2: Space performance with 1000 users and 1000 activities

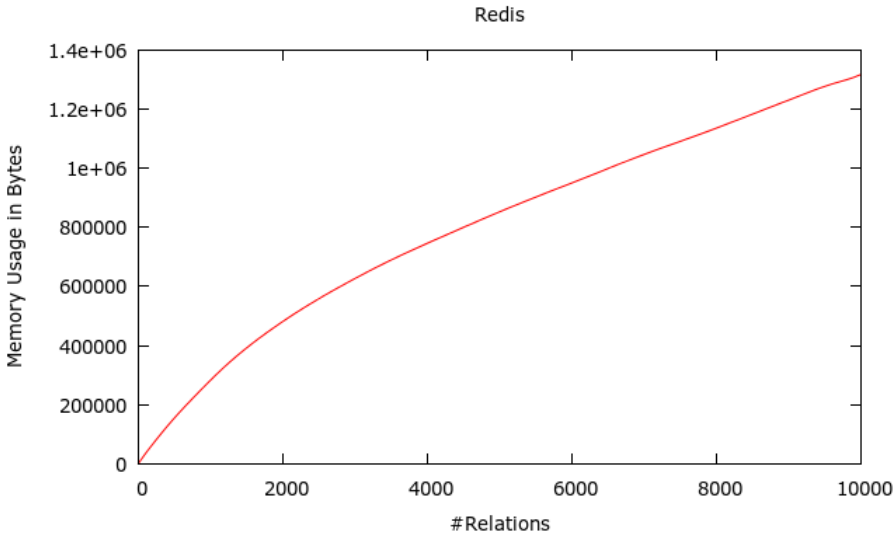


Figure 4.3: Time performance with 1000 users and 1000 relations

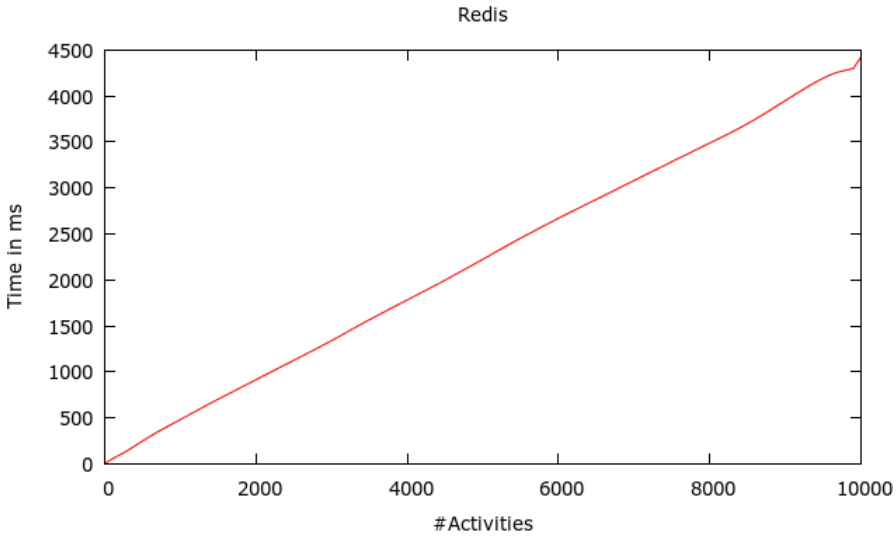
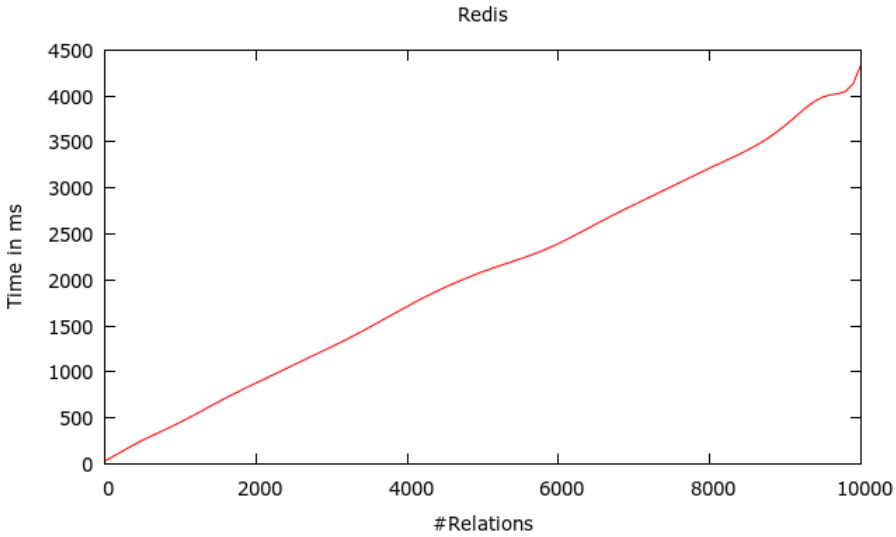


Figure 4.4: Time performance with 1000 users and 1000 activities



4.4 Evalution

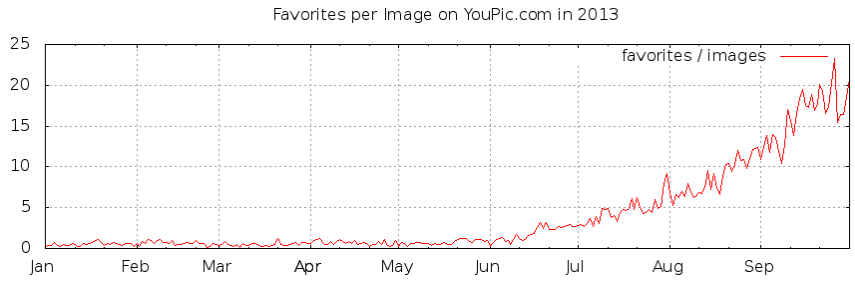
4.4.1 User engagement on YouPic.com

The overarching goal of an activity engine is to increase user engagement with content on the service. In the context of YouPic.com engagement with content can be said to be how many ”favorites” a photo gets. If this metric increases when the activity engine is introduced then we can say that the activity engine has been successful. This is of course easier said than done because YouPic.com is constantly changing its user interface. This iteration of the UI adds confounding factors in the data, making it difficult to parse out the trend for the activity engine.

Nevertheless an attempt has to be made to measure if the activity engine has had any positive effect on the user engagement. To do this I have decided to measure the favorites per photo before and after the introduction of the FanFlow activity stream. Taking May 27, 2013 as the date at which FanFlow was added to YouPic.com we can see from Figure 4.5 that the favorites per photo has increased substantially.

The increase in engagement with content by activity streams is supposed to come from the fact that the activity streams serve up more relevant content. The hypothesis being that a user is more prone to engage with content that he/she finds relevant. As we can see this is what has happened at YouPic.com since the introduction of the FanFlow activity stream. Users are getting photos from other users they

Figure 4.5: Favorites per photo



have actively shown interest in by becoming fans of them, presumably because they like their photos. Increasing this type of engagement makes users more likely to come back to the site because they get more feedback on their photos and more interaction with other users[18].

Taking into account the hypothesis effect of the activity engine and the actual result we can determine that the activity engine has at least been correlated with an increase in favorites on YouPic.com. The user engagement increased in the way we hoped and the activity engine performed well.

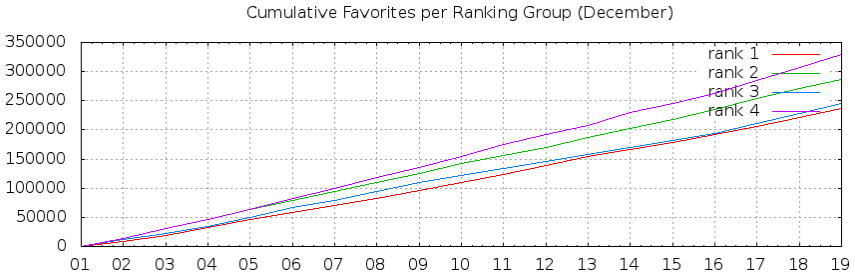
4.4.2 Ranking algorithms

In section 3.6 we define four different ranking algorithms that are going to be evaluated. The way chosen to evaluate the ranking algorithms is by doing what is called an A/B test. An A/B test is a simple randomized experiment where a users are segmented into equal groups each given a variant of the experiment. When the A/B test is concluded some metric is measured to see which of the variants score the highest. Testing the ranking algorithms is done by dividing the user base into four equal groups and assigning them each on of the ranking algorithms. The groups are created by taking the *user_id* modulus 4, if the result is 0 then that user is assigned ranking method 1 and so on.

The performance of the different ranking algorithms is charted in Figure 4.6. The cumulative number of favorites is used to show the different impact the ranking algorithms have on the number of favorites. At the 19th of December the top ranking and the bottom ranking differ by 38% favorites.

The ranking algorithm that resulted in the most favorites was the one that took into account the user's reputation score. As the reputation metric gives a higher score to those that have contributed a lot of good quality photos. This points to the fact that the reputation metric

Figure 4.6: Evaluation of the different ranking algorithms described in chapter 3.



might be a good indicator whether future photos from a user will result in more engagement. The ranking algorithm that took into account the amount of favorites that a user had given a category did poorly and didn't not beat the control chronological ranking.

Chapter 5

Discussion

This chapter discusses the implementation and the results of the analysis. The goals set up in the introduction will serve as an outline for this chapter.

5.1 General framework

One of the goals of this project was to find a set of simple methods that can be implemented to create an activity engine. For these methods to be considered good they need to be implementation agnostic. This means that they are general enough so that they can implemented in several programming languages and computer architectures. The methods also have to be precisely described so as there is no ambiguity as to what they are intended to do.

I would argue that if the methods described in chapter 2 are implemented as per their definition, a correct activity engine would result. Also that the definitions are flexible enough that they can have plurality of implementations.

5.2 Performance

Two questions need to be answered about the performance of an activity engine. The first is whether it copes well with the current input, and the second being at what size input does the performance become unacceptable.

The activity engine that is running on YouPic.com is currently handling the input very well. Every day there are about 5000 photos uploaded to the service and with an average *indegree* of 3.473 that would make an average of 17358 references added to the activity engine per day.

An activity engine becomes unusable when the time it takes to process an activity is larger than the interval at which activities are added to it. This means that if it took 10 second for the activity engine to process a photo on YouPic.com and photos were uploaded every 9 seconds then the activity engine would be broken because photos would be queued up in perpetuity waiting to the processed.

The above scenario is simplistic though because it assumes that only one activity is processed at a time, in fact several *route* methods can be run in parallel because they do not depend on each other or in which order they are executed. As long as the *route* method has the activity and the list of activity streams it can run at anytime.

Another important performance concern is whether the activity engine is horizontally scalable[19]. Being horizontally scalable means that the activity engine can run on an arbitrary number of computers and is not dependent on running in a single unified environment. A simple way to scale the activity engine in practice would first be to distribute the keys used by the redis database to several redis instances. This could be accomplished by using consistent hashing[20] of the keys to determine which instance has which keys. Thereafter the routing and reading of activities be done by any number of machines because both of these methods are not dependent on the order which they are executed.

5.3 Future work

Future work that is hinted at in the discussion is the issue of scalability and making the activity engine work in a distributed manner across several machines. With the ”Fan Out” approach described in this report scaling the system should be straight forward because the activity streams do not depend on each other. One could imagine simply distributing activity streams on to other machines using for example consistent hashing[20].

Another area that can be expanded on is the ranking. The ranking algorithms used in this project are adequate but basic and do not incorporate all the data about a user and their preferences. There are many opportunities to create new ranking algorithms, one interesting method would be to apply the concepts from machine learning. With machine learning a ranking algorithm could be trained on the preferences of a user and in doing so create personalized experience. For example one could create a classifier and uses the favorite photos of a user as the training set. This classifier could then be used to rank the photos in the activity stream. It would be a challenging task to make such a system fast and scalable.

Information filtering systems such as recommendation systems could also be adapted to ranking activity streams. Recommendation systems try to predict the rating a user would give to an item. In this project where there are no ratings and the users only positivly engage with photos through ”favorites” the best approach would probably be to use an item-to-item recommendation system. One such widley used system is Amazon.com’s collaborative filtering[21] which could be adapted to this project use case.

Chapter 6

Conclusions

The overall purpose of this project has been to explore ranked activity streams. This has been done first by asking the question what abstract methods are needed to create a system to handle ranked activity streams? An answer to this question is given in chapter 2 where four methods are described that when implemented will create an activity engine. The abstract methods are then given concrete implementations in chapter 3. In this chapter four ranking algorithms are also implemented that are are later evaluated.

The rest of the report is dedicated to analysis and discussion about the implemented activity engine. In chapter 4 the activity engines performance is analysed looking at both the theoretical performance using complexity theory and at the empirical performance with sample datasets. The analysis shows that the activity engine performs acceptably for the scale it was intended.

In chapter 4 we also discuss the evaluation of the four ranking metrics implemented in chapter 3. The evaluation is done by partitioning the user base into four equal groups and seeing which group has the most favorites. The algorithm that was the clear winner was the one that took into account the reputation of the uploading user. This result should point readers in the direction of what would be a good ranking of activities in a content based activity stream.

Chapter 5 is a discussion about the goals that an activity engine should achieve. Answering questions such as, has the activity stream increased engagement? Or whether the general interface defined in chapter 2 is viable? From data gathered at YouPic.com we can see that engagement has increased, but whether this is due to the activity stream or other changes to the website is inconclusive.

Bibliography

[1] D. Boyd, Streams of Content, Limited Attention: The Flow of Information through Social Media , EDUCAUSE Review 45 (5) (2010) 26–28.
URL http://www.editlib.org/p/110306

[2] A. Snell, M. Norris, D. Wilkinsion, JSON Activity Streams 1.0, Accessed: 2014-05-26.
URL http://activitystrea.ms/specs/json/1.0/

[3] C. Castillo, M. Mendoza, B. Poblete, Information credibility on twitter, in: S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, R. Kumar (Eds.), WWW, ACM, 2011, pp. 675–684.

[4] A. Asemanfar, Timelines @ Twitter, Presentation at QCon LondonAccessed: 2014-05-26.
URL http://www.infoq.com/presentations/Timelines-Twitter

[5] K. Koroleva, H. Krasnova, O. Günther, 'STOP SPAMMING ME!' - Exploring Information Overload on Facebook, in: M. Santana, J. N. Luftman, A. S. Vinze (Eds.), AMCIS, Association for Information Systems, 2010, p. 447.

[6] M. Naaman, J. Boase, C.-H. Lai, Is it really about me?: message content in social awareness streams, in: K. I. Quinn, C. Gutwin, J. C. Tang (Eds.), CSCW, ACM, 2010, pp. 189–192.

[7] J. Kincaid, EdgeRank: The secret sauce that makes Facebook's news feed tick, TechCrunch, April.

[8] K. Hamilton, K. Karahalios, C. Sandvig, M. Eslami, A path to understanding the effects of algorithm awareness, in: M. Jones,

P. A. Palanque, A. Schmidt, T. Grossman (Eds.), CHI Extended Abstracts, ACM, 2014, pp. 631–642.

[9] K. Bontcheva, G. Gorrell, B. Wessels, Social Media and Information Overload: Survey Results, CoRR abs/1306.0813.

[10] S. Berkovsky, J. Freyne, G. Smith, Personalized Network Updates: Increasing Social Interactions and Contributions in Social Networks, in: J. Masthoff, B. Mobasher, M. C. Desmarais, R. Nkambou (Eds.), UMAP, Vol. 7379 of Lecture Notes in Computer Science, Springer, 2012, pp. 1–13.

[11] I. Guy, I. Ronen, A. Raviv, Personalized activity streams: sifting through the river of news, in: Proceedings of the fifth ACM conference on Recommender systems, ACM, 2011, pp. 181–188.

[12] J. Ramsay, A. Barabesi, J. Preece, A psychological investigation of long retrieval times on the World Wide Web, Interacting with Computers 10 (1) (1998) 77–86.

[13] R. Thomaschke, C. Haering, Predictivity of system delays shortens human response time, International Journal of Human-Computer Studies 72 (3) (2014) 358–365.

[14] J. Chen, R. Nairn, E. H. hsin Chi, Speak little and well: recommending conversations in online social streams, in: D. S. Tan, S. Amershi, B. Begole, W. A. Kellogg, M. Tungare (Eds.), CHI, ACM, 2011, pp. 217–226.

[15] J. E. Hirsch, An Index to Quantify an Individual's Scientific Research Output That Takes into Account the Effect of Multiple Coauthorship, Scientometrics 85 (3) (2010) 741–754, Accessed: 2014-05-26.
URL `http://dx.doi.org/10.1007/s11192-010-0193-9`

[16] L. Egghe, The Hirsch index and related impact measures, Annual review of information science and technology 44 (1) (2010) 65–114.

[17] S. Sanfilippo, P. Noordhuis, Redis, Accessed: 2014-05-26.
URL `http://redis.io/`

[18] A. M. Kaplan, M. Haenlein, Users of the world, unite! The challenges and opportunities of Social Media, Business Horizons 53 (1) (2010) 59 – 68, Accessed: 2014-05-26.
URL `http://www.sciencedirect.com/science/article/pii/S0007681309001232`

[19] J. Guitart, V. Beltran, D. Carrera, J. Torres, E. Ayguadé, Characterizing Secure Dynamic Web Applications Scalability, in: IPDPS, IEEE Computer Society, 2005, p. 108a.

[20] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, D. Lewin, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, in: F. T. Leighton, P. W. Shor (Eds.), STOC, ACM, 1997, pp. 654–663.

[21] G. Linden, B. Smith, J. York, Amazon.com Recommendations: Item-to-Item Collaborative Filtering, IEEE Internet Computing 7 (1) (2003) 76–80.