Scopely Challenge

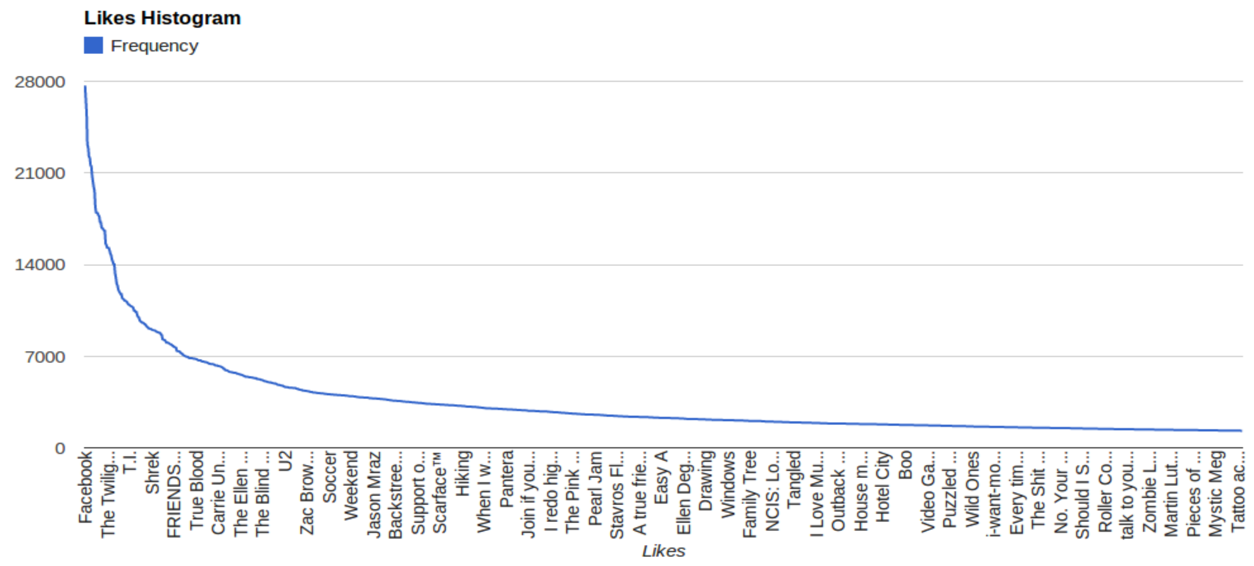
Puneet Singh Ludu

Part 1: Analysis

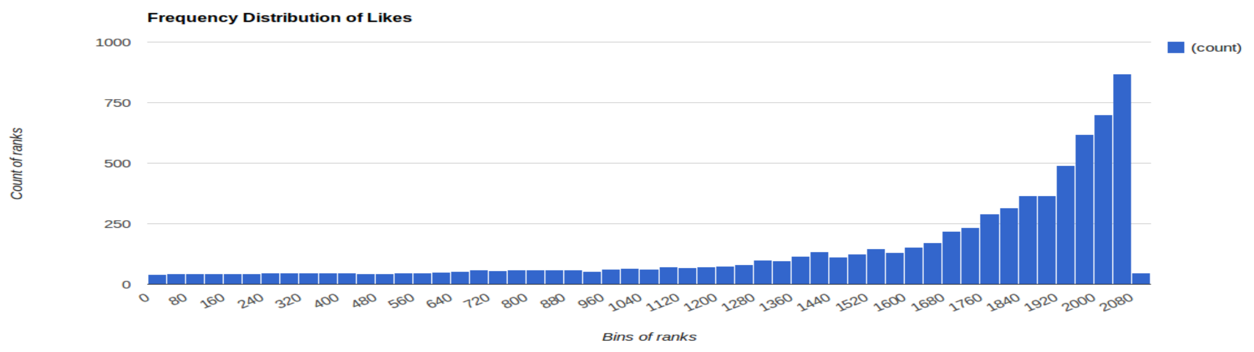
Q1. Histogram of distribution of likes

What was your approach for building this histogram?

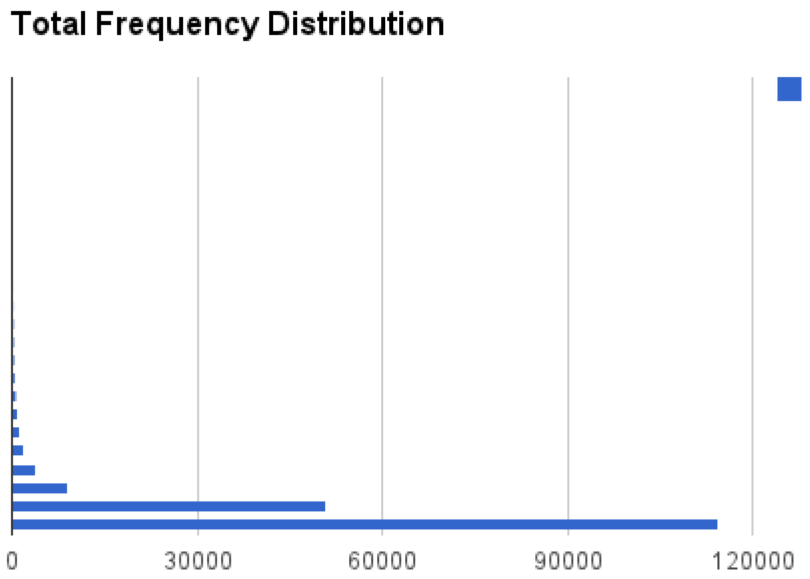
The dataset contains approximately 3558265 unique likes. Pareto principle(80-20 rule) states that top 20% of the data is more useful than rest of the 80% long tail. For generating 'Like' Histogram I used fairly simple approach. I flattened the 'Likes' of each tuple using Perl script and stored it in a different file. I used Pig script (which converts it to Map-Reduce code) to count Frequency of unique likes. I used threshold of 1300 count of likes to cut off the long tail.



This chart shows line chart for top 1463 likes. I chose 1300 to be the threshold because in a bin of 80 ranks of likes 1280-1360 got 50% of the ranks repeated. After this repetition start growing exponentially. Following graph shows "Frequency Distribution" of likes and how tail is growing.



Above graph is truncated at rank 3000, as after that the bar of the graph will rise exponentially and we will not be able to see the reason behind the threshold ‘1300’.



Here is a graph for total frequency distribution, shows exponential rise in the frequency.

How does your approach scale to 1 terabyte of like and interest data?

Since flattening of a like data can easily be ported to Hadoop, and frequency counting problem is pretty standard in Hadoop, this approach would scale linearly.

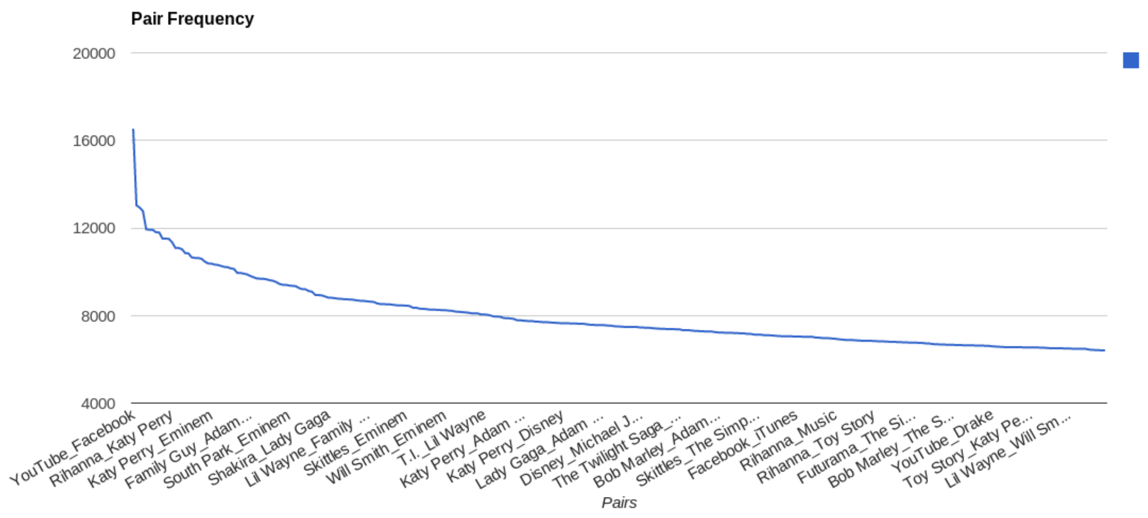
Frequency	Like
27674	Facebook
26626	Eminem
25581	YouTube
23161	Family Guy
22948	Rihanna
22230	Harry Potter
22162	The Simpsons
21583	Music
21505	Michael Jackson
20768	Lady Gaga

Table showing top 10 likes with its frequency.

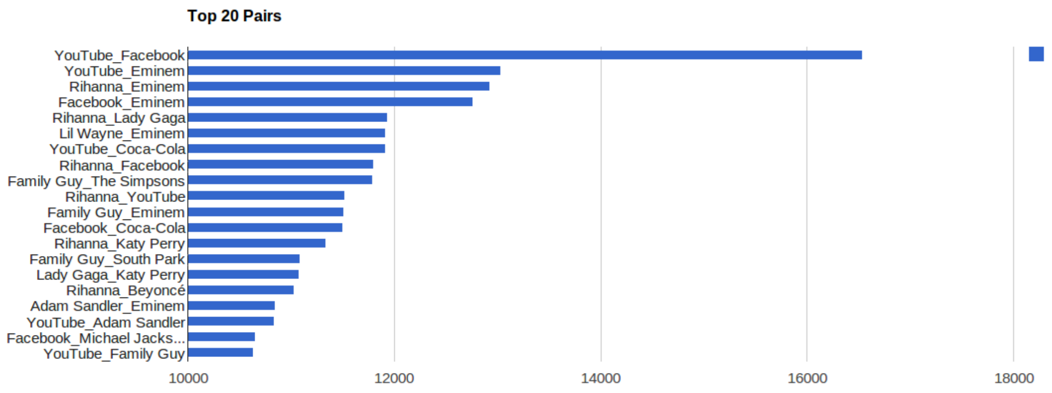
Q2. Histogram of pairs of likes

What was your approach for building this histogram?

Since generating pairs is an $O(N^2)$ complex operation and time to calculate pairs from each tuple may vary, such as some tuple have more than 1000 likes which would generate 1 Million pairs. Since I had single machine and limited resources and time, I chose hybrid approach (Kind of LSH) to solve it.
I generated pairs using Perl script (via its Math::Combinatorics library) for all the tuples having likes less than 100. This approximately covered 70% of the total tuples. I also took out many 1% of the tuples samples randomly from the data and generated pairs. Then, I combined these pairs from both approaches and counted frequency of top 2000 pairs using a pig script similar to Q1. This gave me a fair idea of possible top 2000 pairs but not their actual frequency. So I simply calculated frequency of these 2000 pairs in complete data, and ranked them.



Above figure shows line chart of top 1977 like pairs. Figure Below shows top 20 like pairs.



How does your approach scale as the number of likes increases linearly?

As already discussed above, when likes scale linearly, pairs would scale quadratically (N choose 2). Thus it would be difficult for my approach with the kind of system I have to solve this using my algorithm. Hypothetically I can solve this using parallel computing, or more sophisticated random sampling. It must be noted that our target is not to generate pairs but to calculate like pair histogram. And this approach which uses hybrid of normal pair generation with LSH and other sophisticated sampling techniques can scale and give fairly decent results.
There are many people who tried to solve word co-occurrence problem using Hadoop(Map-Reduce) which scales fairly well for example this on GitHub.

Part 2: Recommendation Systems

Q3. Recommended likes

Please describe your approach

Initially I tried to use Apriori algorithm to generate rules such as:
Facebook, Eminem => Youtube
But since the data is too large it became really time consuming to calculate even simple rules.
So, I chose something very similar to apriori, in which I generated a mapping for each like which maps to frequencies of co-occurring likes. Here is an example:

Suppose we have following data:
user1 : Facebook, Google, Yahoo, Youtube
user2 : Facebook, Cricket, Linux, Youtube
user3 : Linux, Google, Cricket

So my mapping would look like this:
Facebook : Google{1}, Yahoo{1}, Youtube{2}, Cricket{1}, Linux{1}
Google : Facebook{1}, Yahoo{1}, Youtube{1}, Cricket{1}, Linux{1}
Yahoo : Facebook{1}, Google{1}, Youtube{1}
Youtube : Google{1}, Yahoo{1}, Facebook{2}, Cricket{1}, Linux{1}
Cricket : Google{1}, Youtube{1}, Facebook{1}, Linux{2}
Linux : Google{1}, Youtube{1}, Facebook{1}, Cricket{2}

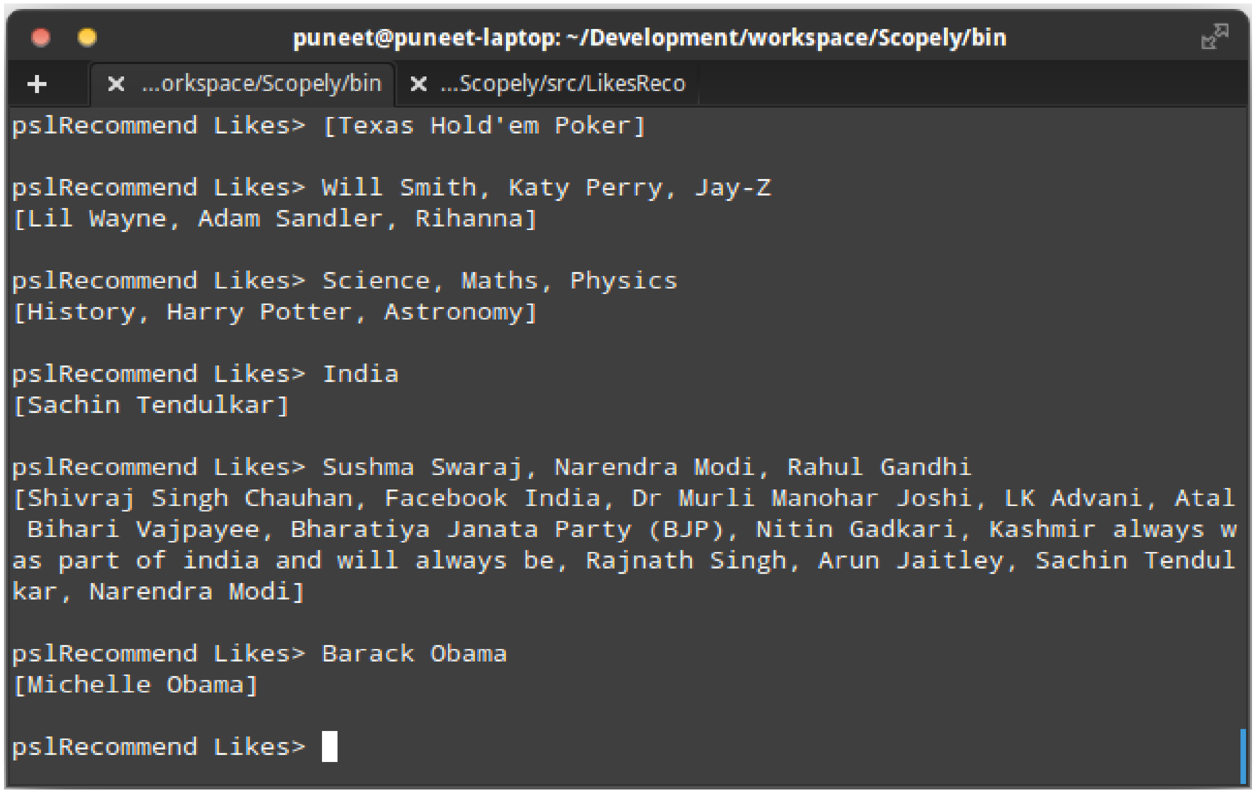
Since this approach also requires to generate pairs to calculate co-occurrence, I used the similar hybrid approach I used in Q2. This type of structure can easily be maintained in <Key, Value> type of data structure (in case of small file, a HashMap; in case of huge file, hBase or Hive)
So suppose the user inputs following to my API:
> Facebook, Linux
we would add both value sets for Facebook and Linux which would create a superset
Google{2}, Yahoo{1}, Youtube{3}, Cricket{2}, Linux{1}, Facebook{1}
Linux and Facebook would be removed from this set
and we can now serve the reply with **Youtube** as it has the highest combined frequency, in case of tie I serve both, but this can be eliminated by randomly selecting one of those.

How does your algorithm change as new data is brought into the system?
As new data is brought in random sampling and Hybrid approach would scale fairly well and the mapping can be updated in real time. For e.g.
user3 started liking Yahoo as well, now all we need is to update transactions related to user3's tuple.

All I need to update are only the mapping containing Yahoo, Linux, Google, Cricket

Updated mapping would look like this (updates in bold):
Facebook : Google{1}, Yahoo{1}, Youtube{2}, Cricket{1}, Linux{1}
Google : Facebook{1}, Yahoo{1}, **Youtube{2}**, Cricket{1}, Linux{1}
Yahoo : Facebook{1}, **Google{2}**, Youtube{1}, **Linux{1}, Cricket{1}**
Youtube : Google{1}, Yahoo{1}, Facebook{2}, Cricket{1}, Linux{1}
Cricket : Google{1}, Youtube{1}, Facebook{1}, Linux{2}, **Yahoo{1}**
Linux : Google{1}, Youtube{1}, Facebook{1}, Cricket{2}, **Yahoo{1}**

How real-time can your system be? For example, suppose the site gets popular in a new country and an influx of like data comes in that you hadn't seen, how would your system react? What are the limitations?
Although this algorithm does not do online learning over streams, but I believe with the basic mapping in place my algorithm can work in near real time. updates in small tuples would not take much time, and updates in large tuples can be handled with LSH or random sampling.
In case of really high influx this can be handled using Apache Storm (a distributed realtime computation system).
One limitation of this system is it trades between Precision vs Recall and Computation time. Thus, to be in favor of good recall with fairly decent precision we can achieve near real time results. Hence, we would sometimes have to compromise with poor precision for some recommendations.



```
pslRecommend Likes> [Texas Hold'em Poker]

pslRecommend Likes> Will Smith, Katy Perry, Jay-Z
[Lil Wayne, Adam Sandler, Rihanna]

pslRecommend Likes> Science, Maths, Physics
[History, Harry Potter, Astronomy]

pslRecommend Likes> India
[Sachin Tendulkar]

pslRecommend Likes> Sushma Swaraj, Narendra Modi, Rahul Gandhi
[Shivraj Singh Chauhan, Facebook India, Dr Murli Manohar Joshi, LK Advani, Atal
 Bihari Vajpayee, Bharatiya Janata Party (BJP), Nitin Gadkari, Kashmir always w
as part of india and will always be, Rajnath Singh, Arun Jaitley, Sachin Tendul
kar, Narendra Modi]

pslRecommend Likes> Barack Obama
[Michelle Obama]

pslRecommend Likes> 
```

Screen shot of the API in action, giving results in 10 milliseconds.

Q4. Recommended users

Please describe your approach

For this system I created an inverted index of the data and then created a mapping of use to user co-occurrence frequency.
For e.g for the following data:
user1 : Facebook, Google, Yahoo, Youtube
user2 : Facebook, Cricket, Linux, Youtube
user3 : Linux, Google, Cricket

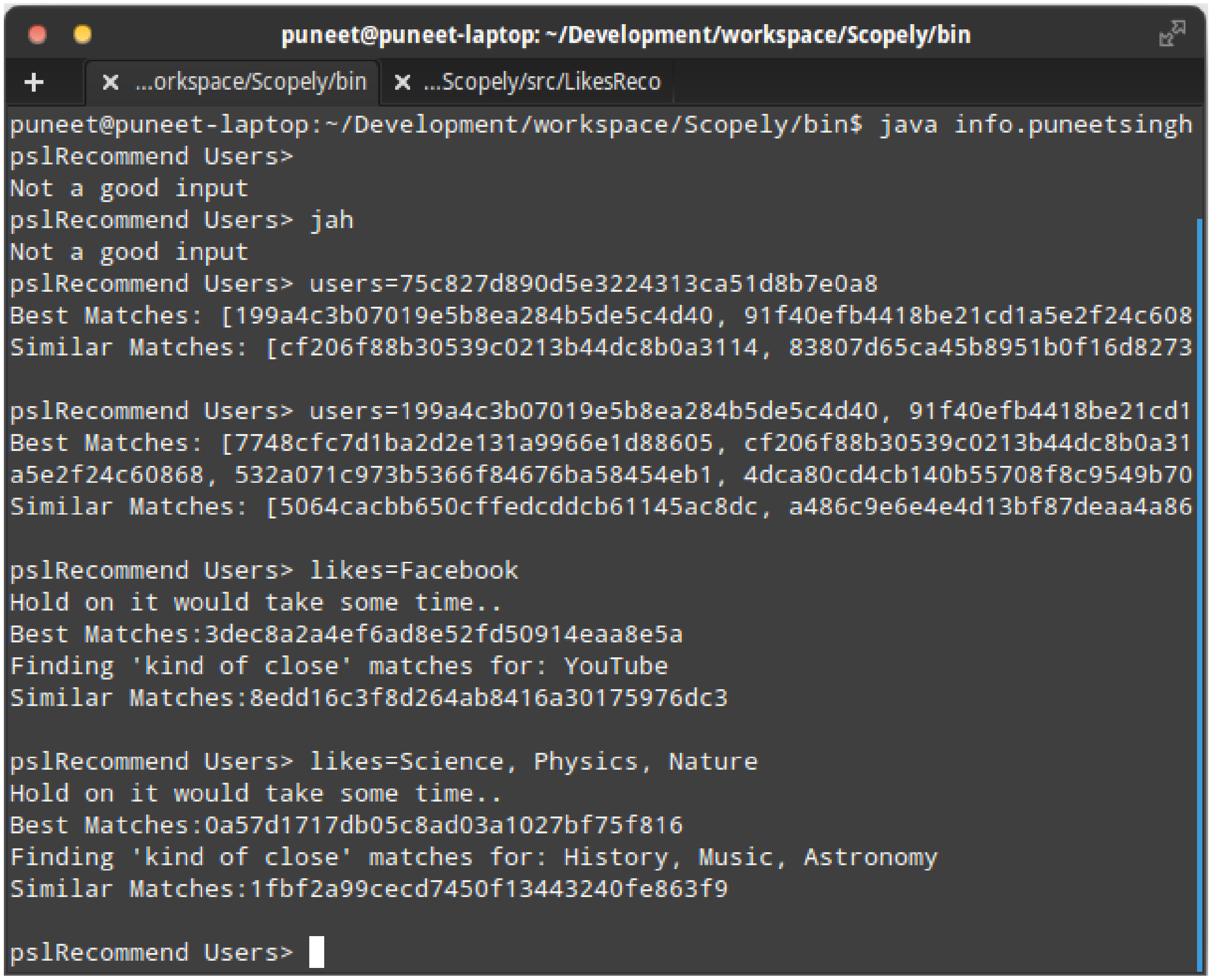
Following inverted index was generated:
Facebook : user1, user2
Google : user1, user3
Yahoo : user1
Youtube : user1, user2
Cricket : user2, user3
Linux : user2, user3

which subsequently generated similar kind mapping used in Q3.

user1 : user2{2}, user3{1}
user2 : user1{2}, user3{2}
user3 : user1{1}, user2{2}

thus for query:
> user3
we would get **user2** as response of the API.
incase of recommending users based on likes, we can use search in inverted index itself. For an example query:
> Facebook, Yahoo
for exact match we would simply add up and chose the best, here:
user1{2},user2{1}
Thus for exact match we would get
user2 as API response.

and for close matches we would use Q3 API to get closest matches Thus Facebook, Yahoo would become **Youtube**, for which we could suggest any of the **user1** or **user2** depending on who did the query.



```
puneet@puneet-laptop:~/Development/workspace/Scopely/bin$ java info.puneetsingh
pslRecommend Users>
Not a good input
pslRecommend Users> jah
Not a good input
pslRecommend Users> users=75c827d890d5e3224313ca51d8b7e0a8
Best Matches: [199a4c3b07019e5b8ea284b5de5c4d40, 91f40efb4418be21cd1a5e2f24c608
Similar Matches: [cf206f88b30539c0213b44dc8b0a3114, 83807d65ca45b8951b0f16d8273

pslRecommend Users> users=199a4c3b07019e5b8ea284b5de5c4d40, 91f40efb4418be21cd1
Best Matches: [7748cfc7d1ba2d2e131a9966e1d88605, cf206f88b30539c0213b44dc8b0a31
a5e2f24c60868, 532a071c973b5366f84676ba58454eb1, 4dca80cd4cb140b55708f8c9549b70
Similar Matches: [5064cacbb650cffedcddcb61145ac8dc, a486c9e6e4e4d13bf87deaa4a86

pslRecommend Users> likes=Facebook
Hold on it would take some time..
Best Matches:3dec8a2a4ef6ad8e52fd50914eaa8e5a
Finding 'kind of close' matches for: YouTube
Similar Matches:8edd16c3f8d264ab8416a30175976dc3

pslRecommend Users> likes=Science, Physics, Nature
Hold on it would take some time..
Best Matches:0a57d1717db05c8ad03a1027bf75f816
Finding 'kind of close' matches for: History, Music, Astronomy
Similar Matches:1fbf2a99cecd7450f13443240fe863f9

pslRecommend Users>
```

Above is the screenshot for API in action

How does your algorithm change as new data is brought into the system? How real-time can your system be? For example, suppose the site gets popular in a new country and an influx of like data comes in that you hadn't seen, how would your system react? What are the limitations?

Maintaining an inverted index in realtime is a challenge, Lucene or Solr can come handy for such problem. Once inverted index is maintained in real time it becomes a similar problem as was in Q3 and we can achieve near realtime response.
One of the main limitation is while suggesting users based on likes, we had to search in the inverted index which in log(N) complexity problem, searching in real time for M users in real time would mean M*log(N) complexity which would be difficult to scale unless distributed with load balancing.

Also it would make sense, in case to huge new data influx, to use log-likelihood based similarity using Mahout as the probability of finding another user that shares a couple of obscure likes would increase and this user would be a really good recommendation.

PS: I would be attaching my code with this mail in the src folder.