Google News Personalization: Scalable Online Collaborative Filtering

Abhinandan Das
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
abhinandan@google.com

Mayur Datar
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
mayur@google.com

Ashutosh Garg
Google Inc.
1600 Amphitheatre Pkwy,
Mountain View, CA 94043
ashutosh@google.com

Shyam Rajaram
University of Illinois at Urbana Champaign
Urbana, IL 61801
rajaram1@ifp.uiuc.edu

ABSTRACT

Several approaches to collaborative filtering have been studied but seldom have studies been reported for large (several million users and items) and dynamic (the underlying item set is continually changing) settings. In this paper we describe our approach to collaborative filtering for generating personalized recommendations for users of Google News. We generate recommendations using three approaches: collaborative filtering using MinHash clustering, Probabilistic Latent Semantic Indexing (PLSI), and covisitation counts. We combine recommendations from different algorithms using a linear model. Our approach is content agnostic and consequently domain independent, making it easily adaptable for other applications and languages with minimal effort. This paper will describe our algorithms and system setup in detail, and report results of running the recommendations engine on Google News.

Categories and Subject Descriptors: H.4.m [Information Systems]: Miscellaneous

General Terms: Algorithms, Design

Keywords: Scalable collaborative filtering, online recommendation system, MinHash, PLSI, Mapreduce, Google News, personalization

1. INTRODUCTION

The Internet has no dearth of content. The challenge is in finding the right content for yourself: something that will answer your current information needs or something that *you* would love to read, listen or watch. Search engines help solve the former problem; particularly if you are looking for something specific that can be formulated as a keyword query. However, in many cases, a user may not even know what to look for. Often this is the case with things like *news*, *movies* etc., and users instead end up browsing sites like news.google.com, www.netflix.com etc., looking around for things that might "interest them" with the attitude: *Show me something interesting*. In such cases, we would like to present recommendations to a user based on her interests as demonstrated by her past activity on the relevant site.

Collaborative filtering is a technology that aims to learn user preferences and make recommendations based on user and community data. It is a complementary technology to content-based filtering (e.g. keyword-based searching). Probably the most well known use of collaborative filtering has been by Amazon.com where a user's past shopping history is used to make recommendations for new products. Various approaches to collaborative filtering have been proposed in the past in research community (See section 3 for details). Our aim was to build a scalable online recommendation engine that could be used for making personalized recommendations on a large web property like Google News. Quality of recommendations notwithstanding, the following requirements set us apart from most (if not all) of the known recommender systems:

Scalability: Google News (*http://news.google.com*), is visited by several million unique visitors over a period of few days. The number of items, news stories as identified by the cluster of news articles, is also of the order of several million.

Item Churn: Most systems assume that the underlying item-set is either static or the amount of churn is minimal which in turn is handled by either approximately updating the models ([14]) or by rebuilding the models ever so often to incorporate any new items. Rebuilding, typically being an expensive task, is not done too frequently (every few hours). However, for a property like Google News, the underlying item-set undergoes churn (insertions and deletions) every few minutes and at any given time the stories of interest are the ones that appeared in last couple of hours. Therefore any model older than a few hours may no longer be of interest and partial updates will not work.

For the above reasons, we found the existing recommender systems unsuitable for our needs and embarked on a new approach with novel scalable algorithms. We believe that Amazon also does recommendations at a similar scale. However, it is the second point (item churn) that distinguishes us significantly from their system. This paper describes our approach and the underlying algorithms and system compo-

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

nents involved. The rest of this paper is organized as follows: Section 2 describes the problem setting. Section 3 presents a brief summary of related work. Section 4 describes our algorithms; namely, user clustering using Minhash and PLSI, and item-item covisitation based recommendations. Section 5 describes how such a system can be implemented. Section 6 reports the results of comparative analysis with other collaborative filtering algorithms and quality evaluations on live traffic. We finish with some conclusions and open problems in Section 7.

2. PROBLEM SETTING

Google News is a computer-generated news site that aggregates news articles from more than 4,500 news sources worldwide, groups similar stories together and displays them according to each reader's personalized interests. Numerous editions by country and language are available. The home page for Google News shows "Top stories" on the top left hand corner, followed by category sections such as *World, U.S. , Business,*, etc. Each section contains the top three headlines from that category. To the left of the "Top Stories" is a navigation bar that links each of these categories to a page full of stories from that category. This is the format of the home page for non signed-in users[1]. Furthermore, if you sign-in using your Google account and opt-in to the "Search History" feature that is provided by various Google product websites, you enjoy two additional features:

(a) Google will record your search queries and clicks on news stories and make them accessible to you online. This allows you to easily browse stories you have read in the past.

(b) Just below the "Top Stories" section you will see a section labeled "Recommended for *youremailaddress*" along with three stories that are recommended to you based on your past click history.

The goal of our project is to present recommendations to signed-in users based on their click history and the click history of the community. In our setting, a user's click on an article is treated as a positive vote for the article. This sets our problem further apart from settings like Netflix, MovieLens etc., where users are asked to rate movies on a 1-5 scale. The two differences are:

1. Treating clicks as a positive vote is more noisy than accepting explicit 1-5 star ratings or treating a purchase as a positive vote, as can be done in a setting like amazon.com. While different mechanisms can be adopted to track the authenticity of a user's vote, given that the focus of this paper is on collaborative filtering and not on how user votes are collected, for the purposes of this paper we will assume that clicks indeed represent user interest. [2]

2. While clicks can be used to capture positive user interest, they don't say anything about a user's negative interest. This is in contrast to Netflix, eachmovie etc. where users give a rating on a scale of 1-5.

[1]The format for the web-site is subject to change.

[2]While in general a click on a news article by a user does not necessarily mean that she likes the article, we believe that this is less likely in the case of Google News where there are clean (non-spammy) snippets for each story that the user gets to see before clicking. Infact, our concern is that often the snippets are so good in quality that the user may not click on the news story even if she is interested in it; she gets to know all that she wants from the snippet

2.1 Scale of our operations

The Google News website is one of the most popular news websites in the world receiving millions of page views and clicks from millions of users. There is a large variance in the click history size of the users, with numbers being anywhere from zero to hundreds, even thousands, for certain users. The number of news stories[3] that we observe over a period of one month is of the order of several million. Moreover, as mentioned earlier, the set of news stories undergoes a constant churn with new stories added every minute and old ones getting dropped.

2.2 The problem statement

With the preceding overview, the problem for our recommender system can be stated as follows: Presented with the click history for N users ($\mathcal{U} = \{u_1, u_2, \ldots, u_N\}$) over M items ($\mathcal{S} = \{s_1, s_2, \ldots, s_M\}$), and given a specific user u with click history set C_u consisting of stories $\{s_{i_1}, \ldots, s_{i_{|C_u|}}\}$, recommend K stories to the user that she might be interested in reading. Every time a signed-in user accesses the home-page, we solve this problem and populate the "Recommended" stories section. Similarly, when the user clicks on the Recommended section link in the navigation bar to the left of the "Top Stories" on home-page, we present her with a page full of recommended stories, solving the above stated problem for a different value of K. Additionally we require that the system should be able to incorporate user feedback (clicks) instantly, thus providing instant gratification.

2.3 Strict timing requirements

The Google News website strives to maintain a strict response time requirement for any page views. In particular, home-page view and full-page view for any of the category sections are typically generated within a second. Taking into account the time spent in the News Frontend webserver for generating the news story clusters by accessing the various indexes that store the content, time spent in generating the HTML content that is returned in response to the HTTP request, it leaves a few hundred milliseconds for the recommendation engine to generate recommendations.

Having described the problem setting and the underlying challenges, we will give a brief summary of related work on recommender systems before describing our algorithms.

3. RELATED WORK

Recommender systems can be broadly categorized into two types: **Content based** and **Collaborative filtering**. In content based systems the similarity of items, defined in terms of their content, to other items that have been rated highly by the user is used to recommend new items. However, in the case of domains such as news, a user's interest in an article cannot always be characterized by the terms/topics present in a document. In addition, our aim was to build a system that could be potentially applied to other domains (e.g. images, music, videos), where it is hard to analyse the underlying content, and hence we developed a content-agnostic system. For the particular application of

[3]As mentioned earlier, the website clusters news articles from different news sites (e.g. BBC, CNN, ABC news etc.) that are about the same story and presents an aggregated view to the users. For the purpose of our discussion, when we refer to a news story it means a cluster of news articles about the same story as identified by Google News.

Google News recommendations, arguably content based recommendations may do equally well and we plan to explore that in the future. Collaborative filtering systems use the item ratings by users to come up with recommendations, and are typically content agnostic. In the context of Google News, item ratings are binary; a click on a story corresponds to a 1 rating, while a non-click corresponds to a 0 rating. Collaborative filtering systems can be further categorized into types: memory-based, and model-based. Below, we give a brief overview of the relevant work in both these types while encouraging the reader to study the survey article [1]

3.1 Memory-based algorithms

Memory-based algorithms make ratings predictions for users based on their *past* ratings. Typically, the prediction is calculated as a weighted average of the ratings given by other users where the weight is proportional to the "similarity" between users. Common "similarity" measures include the Pearson correlation coefficient ([19]) and the cosine similarity ([3]) between ratings vectors. The pairwise similarity matrix $w(u_i, u_j)$ between users is typically computed offline. During runtime, recommendations are made for the given user u_a using the following formula:

$$r_{u_a,s_k} = \sum_{i \neq a} I_{(u_i,s_k)} w(u_a, u_i) \qquad (1)$$

Note that this formula applies to our setting where the ratings are binary. The indicator variable $I_{(u_i,s_k)}$ is 1 if the user u_i clicked on the story s_k and 0 otherwise. The predicted rating r_{u_a,s_k} can be binarized using an appropriate threshold.

Memory-based methods have grown in popularity because of their simplicity and the relatively straightforward training phase. However, as noted in [23], one of the biggest challenges is to make memory-based algorithms more scalable. In fact [23] focusses on instance selection to reduce the training set size, as means to achieve this scalability. However, their techniques are not applicable in our scenario due to the large item churn. For instance, one of their methods (TURF1) tries to compute for each item, a subset of training users that are sufficient to predict any given users rating on this item. Clearly this wont't work for Google News since an old news item, for which this computation can be offline, is typically too stale to recommend anyway.

A variation of the memory-based methods [21], tries to compute the similarity weight matrix between all pairs of items instead of users. The similarity is computed based on the ratings the items receive from users and measures such as Pearson correlation or vector similarity are used. During the testing phase, recommendations are made to users for items that are similar to those they have rated highly.

3.2 Model-based algorithms

In contrast to the memory-based algorithms, model-based algorithms try to model the users based on their past ratings and use these models to predict the ratings on *unseen* items. One of the earliest examples of this approach, include [3] which proposes two alternative probabilistic models: cluster models and Bayesian models. The shortcoming of this paper was that it only categorized each user into a single class while intuitively a user may have different tastes corresponding to different topics. Similar to our approach, most of the recent work in model-based algorithms captures multiple interests of users by classifying them into multiple clusters or classes. Model-based approaches include: latent semantic indexing (LSI) [20], Bayesian clustering [3], probabilistic latent semantic indexing (PLSI) [14], multiple multiplicative Factor Model [17], Markov Decision process [22] and Latent Dirichlet Allocation [2]. Most of the model-based algorithms are computationally expensive and our focus has been on developing a new, highly scalable, cluster model and redesigning the PLSI algorithm [14] as a MapReduce [12] computation to make it highly scalable.

4. ALGORITHMS

We use a mix of memory based and model based algorithms to generate recommendations. As part of model-based approach, we make use of two clustering techniques - PLSI and MinHash and as part of memory based methods, we make use of item covisitation. Each of these algorithms assigns a numeric score to a story (such that better recommendations get higher score). Given a set of candidate stories, the score (r_{u_a,s_k}) given by clustering approaches is proportional to

$$r_{u_a,s_k} \propto \sum_{c_i : u_a \in c_i} w(u_a, c_i) \sum_{u_j : u_j \in c_i} I_{(u_j,s_k)}$$

where $w(u_a, c_i)$ is proportional to the fractional membership of the user u_a to cluster c_i. The covisitation algorithm assigns a score to each candidate story which is proportional to the number of times the story was covisited with the other stories in the user's click-history.

The scores given by each of these algorithms are combined as $\sum_a w_a r_s^a$ (where w_a is the weight given to algorithm a and r_s^a is the score given by algorithm a to story s) to obtain a ranked list of stories. Top K stories are chosen from this list as recommendations for the user. The weights used in combining the individual algorithm scores (w_a's) are learned by exploring a pre-selected discrete parameter space (possible combinations of weights) and for each point in the parameter space running a live experiment (see section 6.5) to see which one performs the best. In future we plan to explore using SVM [7] (with linear kernel) to learn these weights. Next we describe each of these algorithms in detail.

4.1 MinHash

MinHashing is a probabilistic clustering method that assigns a pair of users to the same cluster with probability proportional to the overlap between the set of items that these users have voted for (clicked-on). Each user $u \in \mathcal{U}$ is represented by a set of items (news stories) that she has clicked on, i.e her click history C_u. The similarity between two users u_i, u_j is defined as the overlap between their item sets given by the formula $\mathcal{S}(u_i, u_j) = \frac{|C_{u_i} \cap C_{u_j}|}{|C_{u_i} \cup C_{u_j}|}$. This similarity measure, also known as the Jaccard coefficient, takes values between 0 and 1 and it is well known that the corresponding distance function $\mathcal{D}(u_i, u_j) = 1 - \mathcal{S}(u_i, u_j)$ is a metric [6]. As a thought experiment, given a user u_i, conceptually we would like to compute the similarity of this user, $\mathcal{S}(u_i, u_j)$, to all other users u_j, and recommend to user u_i stories voted by u_j with weight equal to $\mathcal{S}(u_i, u_j)$. However, doing this in real-time is clearly not scalable; one could imagine simple pruning techniques such as using a hash table to find out users who have at least one vote in common, but even doing so is not going to reduce the number of candidates to a manageable number due to the presence of popular stories. Offline computation is also infeasible for such a large number of user pairs. Not suprisingly, what comes to our rescue is a provably sublinear time near-neighbor search technique called Locality Sensitive Hashing (LSH) [16].

4.1.1 LSH

The LSH technique was introduced by Indyk and Motwani [16] to efficiently solve the near-neighbor search problem and since then has found applications in many fields [13, 9, 5]. The key idea is to hash the data points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. LSH schemes are known to exist for the following distance or similarity measures: Hamming norm [13], L_p norms [13, 11], Jaccard coefficient [4, 8], cosine distance and the earth movers distance (EMD) [6]. Our similarity measure, the Jaccard coefficient, thankfully admits a LSH scheme called Min-Hashing (short for Minwise Independent Permutation Hashing) that was first introduced by Cohen [8] to estimate the size of transitive closure and reachability sets (see also Broder [4]).

The basic idea in the Min-Hashing scheme is to randomly permute the set of items ($\mathcal{S}$) and for each user u_i compute its hash value $h(u_i)$ as the index of the *first* item under the permutation that belongs to the user's item set C_{u_i}. It is easy to show ([8, 4, 9]) that for a random permutation, chosen uniformly over the set of all permutations over $\mathcal{S}$, the probability that two users will have the same hash function is exactly equal to their similarity or Jaccard coefficient. Thus, we can think of min-hashing as a probabilistic clustering algorithm, where each hash bucket corresponds to a cluster, that puts two users together in the same cluster with probability equal to their item-set overlap similarity $\mathcal{S}(u_i, u_j)$. Similar to [16], we can always concatenate p hash-keys for users, where $p \geq 1$, so the probability that any two users u_i, u_j will agree on the concatenated hash-key is equal to $\mathcal{S}(u_i, u_j)^p$. In other words, by concatenating the hash-keys we make the underlying clusters more refined so that there are more of these clusters and the average similarity of the users within a cluster is greater. From the perspective of finding near neighbors for a given user, these refined clusters have high precision but low recall. We can improve the recall by repeating this step in parallel multiple times, i.e. we will hash each user to q clusters where each cluster is defined by the concatenation of p MinHash keys. Typical values for p and q that we have tried lie in the ranges $2-4$ and $10-20$ respectively.

Clearly, generating random permutations over millions of items and storing them to compute MinHash values is not feasible. Instead, what we do is generate a set of independent, random seed values, one for each MinHash function (as per the discussion above, $p \times q$), and map each news-story to a hash-value computed using the Id of the news story and the seed value. The hash-value thus computed serves as a proxy for the index in the random permutation. By choosing the range of the hash-value to be $0 \ldots 2^{64}-1$ (unsigned 64 bit integer) we ensure that we do not encounter the "birthday paradox" [18] as long as the item set is less than 2^{32} in size, thereby having a small chance of collision. The (approximate) MinHash values thus computed have properties similar to the ideal MinHash values [15]. Next, we describe how we can compute the MinHash values in a scalable manner, over millions of users and items, using the Mapreduce computation framework.

4.1.2 MinHash clustering using MapReduce

MapReduce [12] is a very simple model of computation over large clusters of machines that can handle processing of large amounts of data in relatively short periods of time and scales well with the number of machines. Tens or hundreds of Terabytes of data can be processed with thousands of machines within hours. The computation works in the following three phases:

Map inputs to key-value pairs: In the Map phase, we read the input records independently, in parallel, on different machines and map each input to a set of zero or more key-value pairs. In our case, each input record (one for every user u_i) is a user's click history C_{u_i}. We iterate over the user's click history and compute $p \times q$ MinHash values for this user. Computing a single MinHash value is very easy: we hash each item in the history using the item's Id and the random seed corresponding to the hash function [4] and maintain the minimum over these hash values. Finally, we bunch the MinHash values in q groups of p MinHash values each. For each group, we concatenate the MinHash values to obtain the cluster-id corresponding to this group. The key-value pair that is output (one for each cluster that the user belongs to) is the cluster-id (key) and the user-id (value).

Partition and Shuffle the key-value pairs: In this phase, the key-value pairs output at the end of the Map phase are split into partitions (shards), typically based on the hash value of the keys. Each shard is sorted on the keys so that all the key-value pairs for the same key (in our case the cluster-id) appear together.

Reduce key-value pairs: In the reduce phase, we obtain for each cluster-id the list of user-ids that belong to this cluster (membership list) and prune away clusters with low membership.In a separate process, we also invert the cluster membership and maintain for each user the list of clusters that she belongs to, along with her click history. The user information (cluster-ids and click history) is stored in a Bigtable [10] keyed by the user-id. (See description of User Table **UT** in section 5.2 for more details).

4.2 PLSI

PLSI was introduced in [14], where Hofmann developed probabilistic latent semantic models for performing collaborative filtering. It models users ($u \in \mathcal{U}$) and items ($s \in \mathcal{S}$) as random variables, taking values from the space of all possible users and items respectively. The relationship between users and items is learned by modeling the joint distribution of users and items as a mixture distribution. A hidden variable Z (taking values from $z \in \mathcal{Z}$, and $\|\mathcal{Z}\| = L$) is introduced to capture this relationship, which can be thought of as representing user communities (like-minded users) and item communities (genres). Formally, the model can be written in the form of a mixture model given by the equation:

$$p(s|u;\theta) = \sum_{z=1}^{L} p(z|u)p(s|z). \tag{2}$$

[4]Each mapper machine has an identical copy of the random seed values.

The model is completely specified by parameters θ representing conditional probability distributions (CPDs) $p(z|u)$ and $p(s|z)$. The key contribution of the model is the introduction of the latent variable Z, which makes users and items conditionally independent. The model can also be thought of as a generative model in which state z of the latent variable Z is chosen for an arbitrary user u based on the CPD $p(z|u)$. Next, an item s is sampled based on the chosen z from the CPD $p(s|z)$.

4.2.1 Mapreducing EM Algorithm

Learning the co-occurrence model from training data of size T involves estimating the CPDs $p(z|u)$ and $p(s|z)$ such that the product of conditional likelihood over all data points is maximized, equivalently minimizing the empirical logarithmic loss given by the following equation:

$$\mathrm{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\log(p(s_t|u_t;\theta))$$

Expectation Maximization (EM) is used to learn the maximum likelihood parameters of this model. The details of the actual EM algorithm and its derivation can be found in [14]. The algorithm is an iterative one with each iteration consisting of two steps: The E-Step involves the computation of Q variables (i.e. the a-posteriori latent class probabilities) given by the following equation:

$$q^*(z;u,s;\hat{\theta}) := p(z|u,s;\hat{\theta}) = \frac{\hat{p}(s|z)\hat{p}(z|u)}{\sum_{z\in\mathcal{Z}}\hat{p}(s|z)\hat{p}(z|u)}$$

and the M-step uses the above computed Q function to compute the following distributions:

$$p(s|z) = \frac{\sum_u q^*(z;u,s;\hat{\theta})}{\sum_s\sum_u q^*(z;u,s;\hat{\theta})}, \quad (3)$$

$$p(z|u) = \frac{\sum_s q^*(z;u,s;\hat{\theta})}{\sum_z\sum_s q^*(z;u,s;\hat{\theta})}. \quad (4)$$

Note, in the equations above, $\hat{p}$ values stand for the parameter estimates from the previous iteration of the EM algorithm[5]. Executing the EM algorithm on a single machine becomes infeasible when dealing with our large scale: To get an idea on the space requirements of loading the model into main memory, let $M = N = 10$ million and $L = 1000$. In this case, the memory requirement for the CPDs is $(M{+}N){\times}L{\times}4 \sim 80GB$ (with 4 bytes to represent a double value). Next, we demonstrate how the EM algorithm for computing PLSI parameters can be parallelized, using the Mapreduce [12] framework, to make it scalable. The insight into using mapreduce for the EM algorithm comes from rewriting the equations as

$$q^*(z;u,s;\hat{\theta}) = p(z|u,s;\hat{\theta}) = \frac{\frac{N(z,s)}{N(z)}\hat{p}(z|u)}{\sum_{z\in\mathcal{Z}}\frac{N(z,s)}{N(z)}\hat{p}(z|u)}, \text{where}$$

$$N(z,s) = \sum_u q^*(z;u,s;\hat{\theta})$$

[5]For the first iteration, we set $\hat{p}$ to appropriately normalized random values that form a probability distribution.

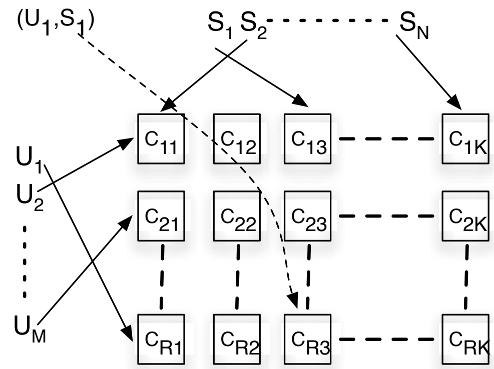


Figure 1: Sharding of users and items for mapreducing EM algorithm

$$N(z) = \sum_s\sum_u q^*(z;u,s;\hat{\theta})$$

$$\hat{p}(z|u) = \frac{\sum_s q^*(z;u,s;\hat{\theta})}{\sum_z\sum_s q^*(z;u,s;\hat{\theta})}$$

Given a user-story pair (u,s) the sufficient statistics from the previous iteration that are needed to compute $q^*(z;u,s;\hat{\theta})$ include: $\hat{p}(z|u)$, $N(z,s)$ and $N(z)$. Lets assume that these statistics are available for every user u and story s at the begining of a new EM iteration. The important observation is that given the sufficient statistics, the computation of the $q^*(z;u,s;\hat{\theta})$ can be done independently and parallely for every (u,s) pair observed in the click logs. We will describe how a single iteration (next iteration) can be executed as a Mapreduce computation. Consider a grid of size $R\times K$ of mapper computers (Figure 1). Users and items are sharded into R, K groups respectively (as a function of their Ids) and click data corresponding to the (u,s) pair is sent to the appropriate (i,j)th machine from the grid where i is the shard that u belongs to and j is the shard that s belongs[6]. Note that the (i,j)th machine only needs to load CPDs and sufficient statistics corresponding to the users in ith shard and items in jth shard respectively. This drastically reduces the memory requirement for each machine since it has to load $1/R$th of the user CPDs and $1/K$th of the item CPDs. Having computed $q^*(z;u,s;\hat{\theta})$, we output three (key, value) pairs in the Mapper: $(u,\,q^*)$, $(s,\,q^*)$, and $(z,\,q^*)$.

The reducer shard that receives the key-value pairs corresponding to the item s computes $N(z,s)$ (for all z values) for the next iteration. The reducer shard that receives the key-value pairs corresponding to the user u computes $p(z|u)$. $N(z)$ is computed by the reduce shard that receives the key-value pairs corresponding to z[7]. Note that the computation in all the reduce shards is a simple addition.

[6]The click data does not change between iterations and needs to be sharded only once at the start

[7]The reduce shards corresponding to the z values receive a lot of data (one entry for each click pair (u,s) and if aggregating this data in a single reducer becomes a bottle neck we can perform some preprocessing in the shuffle stage of the Mapreduce.

4.2.2 Using PLSI with Dynamic Datasets

While the past research ([14]) shows that PLSI fairs well in comparison to other algorithms, it suffers from the fundamental issue that every time new users/items are added, the whole model needs to be retrained. By parallelizing the model, we can learn the probability distributions quickly. However, the model still suffers from the fact that it is not real time. Some heurisitcs (approximations) have been provided in the literature which allow one to update the model in the case of few additions of items and users. While the number of new users that are added daily is a small fraction, the set of stories has a big overhaul each day rendering such update heuristics ineffective. To get around this, we use an approxiate version of PLSI that makes use of $P(z|u)$ values learned from the above model. Z values are treated as clusters and the distribution as giving the fractional cluster memberships. We keep track of the activity observed from each cluster for every story. When a user clicks on a story, we update the counts associated with that story for all the clusters to which the user belongs (weighted count is updated based on the membership confidence). This weight matrix is normalized to give the distribution $P(s|z)$. Note, $P(s|z)$ thus computed can be updated in real time. However, this model still suffers from the fact that the new users cannot be added. While there are a few heuristics which can be employed to do this, we defer that to future work. For new users, we rely on the story-story covisitation algorithm, that is described next, to generate useful recommendations based on their limited history.

4.3 Using user clustering for recommendations

Once the users are clustered, we maintain the following statistics for each cluster at serving time: the number of clicks, decayed by time, that were received on different stories by members of this cluster. In case of PLSI, the count of clicks is further weighted by the fractional cluster membership $P(z|u)$. Note that since these clusters are refined, the number of users that belong to each is limited and hence the number of unique stories that are clicked by the cluster's members is typically small (few thousand at the most). When evaluating a candidate news story s for possible recommendation to a user u, we compute an unnormalized story score based on clustering as follows: fetch the clusters that this user belongs to, for each cluster lookup how many times (discounted by age) did members of this cluster click on the story s (normalized by the total number of clicks made by members of this cluster), finally add these numbers to compute the recommendation score. The recommendation scores thus obtained are normalized (by simple scaling) so that they all lie between 0 and 1. We compute these normalized scores based on MinHash and PLSI clustering separately.

4.4 Covisitation

Our item based technique for generating recommendations makes use of covisitation instances, where covisitation is defined as an event in which two stories are clicked by the same user within a certain time interval (typically set to a few hours). Imagine a graph whose nodes represent items (news stories) and weighted edges represent the time discounted number of covisitation instances. The edges could be directional to capture the fact that one story was clicked after the other, or not if we do not care about the order. We maintain this graph as an adjacency list in a Bigtable ([10]) that is keyed by the item-id (see description of Story Table **ST** in section 5.2). Whenever we receive a click from user u_i on item s_k, we retrieve the user's recent click history C_{u_i} and iterate over the items in it. For all such items $s_j \in C_{u_i}$, we modify the adjacency lists for both s_j and s_k to add an entry corresponding to the current click. If an entry for this pair already exists, we update the age discounted count. Given an item s, its near neighbors are effectively the set of items that have been covisited with it, weighted by the age discounted count of how often they were covisited. This captures the following simple intuition: "Users who viewed this item also viewed the following items".

For a user u_i, we generate the covisitation based recommendation score for a candidate item s as follows: We fetch the user's recent click history C_{u_i}, limited to past few hours or days[8]. For every item s_i in the user's click history, we lookup the entry for the pair s_i, s in the adjacency list for s_i stored in the Bigtable. To the recommendation score we add the value stored in this entry normalized by the sum of all entries for s_i. Finally, all the covisitation scores are normalized to a value between 0 and 1 by linear scaling.

4.5 Candidate generation

So far we have assumed that when asked to generate recommendations we will also be provided with a set of candidate items. These candidates can be generated in two ways: The News Frontend (NFE) may generate a list of candidates based on factors such as the news edition, language preferences of the user, story freshness, customized sections selected by the user etc. The exact scoring method used to generate the candidate set is independent of our recommender system. Alternately, candidates for a given user can be generated as follows: consider the union of all stories that have been clicked by the members of the clusters that this user belongs to and the set of stories that have been covisited with the set of stories in the user's click history. As per our algorithm, only stories from this set will get non-zero score, and therefore this set is a sufficient candidate set.

5. SYSTEM SETUP

Putting together the above algorithms into a real time recommendation system requires the following three main components: An offline component that is responsible for periodically clustering users based on their click history; a set of online servers responsible for performing two main types of tasks: (a) Updating user and story statistics each time a user clicks on a news story, and (b) Generating news story recommendations for a given user when requested; and two types of data tables: a user table **UT** indexed by user-id that stores user click history and clustering information, and a story table **ST** indexed by story-id that stores real time click counts for every story-story and story-cluster pair. We describe each of these components in more detail below:

5.1 Offline processing

Log analysis is performed periodically as MapReduces over user click-history stored in the user table UT. During this

[8] We consider covisitation based recommendations as those that are a function of user's short term behavior (click history in the last few hours) while user-based recommendations (MinHash and PLSI) as those that are a function of user's long term behavior.

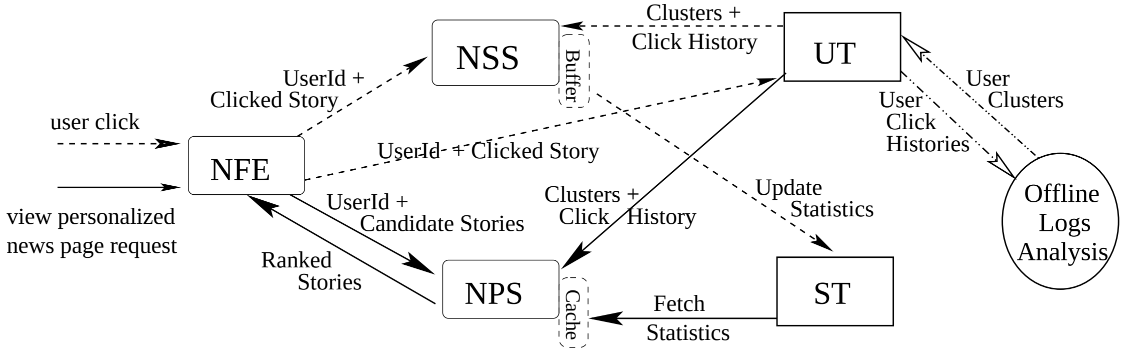


Figure 2: System Components

step we look at the clicks made by users over a time window consisting typically of a few months and cluster the users using the MinHash and PLSI clustering algorithms described in sections 4.1 and 4.2. The user clusters thus computed are then written to the UT as part of the *user information* that will be used for generating recommendations.

5.2 Data tables

The user table **UT** and story table **ST** are conceptually two dimensional tables that are used for storing various kinds of statistics on a per user and per story basis. The rows of the user table are indexed by user-id, and for each user-id, two kinds of information are stored in the table:
(a) Cluster information: A list of MinHash and PLSI cluster-ids that the user belongs to, and
(b) Click history: The list of news story-id's that the user has clicked on.

These two sets of items collectively represent all the user-specific information used to generate recommendations.

The rows of the story table are indexed by story-id, and for each row corresponding to a story S, there are two main types of statistics that are maintained in different columns:
(a) Cluster statistics: How many times (weighted by the user's fractional membership $p(z|u)$ in case of PLSI) was story S clicked on by users from each cluster C. Here C may either be a MinHash or a PLSI cluster.
(b) Covisitation statistics: How many times was story S co-visited with each story S'. Conceptually, the covisitation statistics represent the adjacency list for the covisitation graph described in section 4.4.

For each of the above two types of statistics, we also need to maintain some *normalization* statistics: For every cluster C we need to maintain the total number of clicks made by users belonging to that cluster, and for every story S we need to maintain the total number of story covisitation pairs where this story was one of the covisited pair.

In order to emphasize more recent news story interests expressed by users, and to discount the fact that older stories are likely to have higher click counts simply because of their age, the various counts described above are not maintained as simple counts. Rather, we maintain time decayed counts which give more weight to user clicks from the recent past.

For making the recommendation system an online one, the data tables UT and ST need to provide a mechanism for fast real time update and retrieval (of the order of a few milliseconds) of the statistics corresponding to either a row (e.g. user-id lookup in UT), or a (row , column) pair (e.g. story-id, cluster-id lookup in ST). A suitable candidate for storing these two data tables is the Bigtable infrastructure (c.f. [10]) which is a distributed persistent storage system for structured data that is designed to scale to a very large amount of data across thousands of commodity servers.

5.3 Real time servers

We require online servers to perform two main types of functions: Updating the various statistics and information stored in the data tables whenever a user clicks on a news story, and generating a ranked list of recommended news stories for a given user-id when requested by the user.

Figure 5 gives one possible implementation, where the news statistics server (**NSS**) is responsible for updating statistics in the ST when informed of a user click by the news webserver, referred to as the news front end (**NFE**). The news personalization server (**NPS**) is responsible for generating news story recommendations when requested by the NFE. The front end NFE serves as a proxy through which the personalization servers interact with a user.

5.4 Putting the components together

The various components of the news recommendation system described above mainly interact with each other as part of the work-flow for handling two different kinds of requests initiated by NFE in response to user actions: A request to recommend stories (which the NFE forwards to an NPS), and a request to update click statistics when a user clicks on a news story (which the NFE forwards to an NSS). The overall steps involved in handling each of these two types of requests are outlined below:
1. Recommend request (solid arrows in Figure 5): When a user requests personalized news story recommendations, the NFE contacts an NPS with the user-id and a list of candidate stories to be scored (see Section 4.5 on candidate generation). On receiving this request, the NPS needs to fetch the user information (cluster-id's + recent click history) from the UT, followed by the click counts corresponding to the (MinHash and PLSI) clusters that the user belongs, and the covisitation counts for the stories in her click history. These latter statistics are fetched from ST, and locally cached by the NPS with a suitable expiry window to improve performance. Based on these statistics, as described in Section 4, NPS computes three recommendation scores (cluster-story score based on MinHash and PLSI, and story-story covisitation score) that are linearly combined to

obtain a final score for each of the candidates that is eventually sent back to NFE.

2. Update statistics request (dashed arrows in Figure 5): When a user clicks on a news story, this information is recorded in her click-history stored in UT. The NFE also contacts an NSS with a request to update any statistics that may change as result of this click. In order to update the statistics, the NSS needs to fetch the users information (cluster-ids and click-history) from UT. For every (MinHash and PLSI) cluster that the user belongs to, the corresponding click count for the cluster for this story needs to be updated, weighted by $p(z|u)$ in case of PLSI. Additionally, we need to update the covisitation count for every story in the user's (recent) click-history with the story corresponding to the latest click. These counts, along with the appropriate normalization counts in ST are updated by the NSS. Again, for performance reasons, NSS may choose to buffer these updates and write them out to ST periodically.

One advantage of separating the two functionalities listed above into separate servers is that even if the statistics server NSS fails, although the system will not be able to update statistics for clicks during the downtime, the personalization server NPS can continue to function normally and even generate recommendations using the stale statistics present in the ST. This allows for a graceful degradation in quality of recommendations over the duration of server downtime.

Since the click recording and statistics are updated and retrieved in real time (e.g. via the Bigtable infrastructure mentioned earlier), the system described above is an *online* one, offering 'instant gratification' to users. Thus, every click made by a user affects the scores assigned to different candidate news stories for that user, and potentially changes the set of recommended stories seen by the user in real time. The online nature of the system also allows us to deal with the high item churn associated with news stories by enabling us to recommend relevant news stories to users shortly after they appear in various news sources. In order to limit the actions of "spammy" users from biasing the statistics and possibly affecting the quality of news recommendations for other users, clicks received from users with abnormally high click rates can be ignored when updating statistics as well as when clustering users based on their click history.

6. EVALUATION

In this section, we present the quality evaluation for our individual algorithms and our overall recommendation scheme. In the first part of this section, we compare our individual algorithms, in particular our model-based algorithms, with each other and with the best known memory based algorithm in collaborative filtering research, using test datasets. The purpose of this evaluation is to answer the following question: How does our new algorithm (MinHash based user-clustering) compare, in terms of quality of recommendations, with PLSI and the best known memory based algorithm? It is evident that our implementations for MinHash and PLSI are scalable and that the memory based algorithms do not scale as well.

The second part presents our success metric for the live running system, namely presentation unbiased relative click-through rates, as compared with a simple but competitive recommendation strategy of simply recommending the most popular stories. This is an implicit evaluation of our live system by our users, measured by what they click on, and as part of this evaluation we would like to compare the three individual algorithms (MinHash, PLSI, Covisitation) with each other and also compare our overall algorithm to a natural benchmark – the popular stories that are being clicked by users at any time.

6.1 Test Datasets

We use three test datasets for our comparative study. The first dataset, **MovieLens** dataset, consists of movie rating data collected using a web-based research recommender system. The dataset, after some pruning to make sure that each user has at least a certain number of ratings, contains 943 users, 1670 movies, and about $54,000$ ratings, on a scale from 1 to 5. The second dataset consists of a subset of clicks received on the Google News website over a certain time period, from the top[9] 5000 users (top as sorted by the number of clicks.) There are about $40,000$ unique items that are part of this dataset and about $370,000$ clicks. We refer to this as the **NewsSmall** dataset. The third dataset, **NewsBig**, as the name suggests, is similar to the second one (in fact a superset), and just contains more records: $500,000$ users, $190,000$ unique items and about $10,000,000$ clicks. In order to have uniformity in comparisons, we binarize the first dataset as follows: if the rating for an item, by a user, is larger than the average rating by this user (average computed over her set of ratings) we assign it a binary rating of 1, 0 otherwise.

6.2 Evaluation methodology and metrics

Similar to most machine learning evaluation methodologies, we randomly divide the datasets into a training set and a test set. The training set is to used to learn and predict what other items a user would click and compare the predicted set with the actual set of clicks from the test set or hold-out set. Note, this is done in an offline manner. The split is in the ratio $80\% - 20\%$ (train to test) and done for each user. The numbers reported here are average over numerous such splits. The numbers that we report are the precision (what fraction of the recommendations were actually clicked in the hold-out or test set[10]) and recall (what fraction of the clicks in the hold-out set were actually recommended) fractions over the test set.

6.3 Algorithms

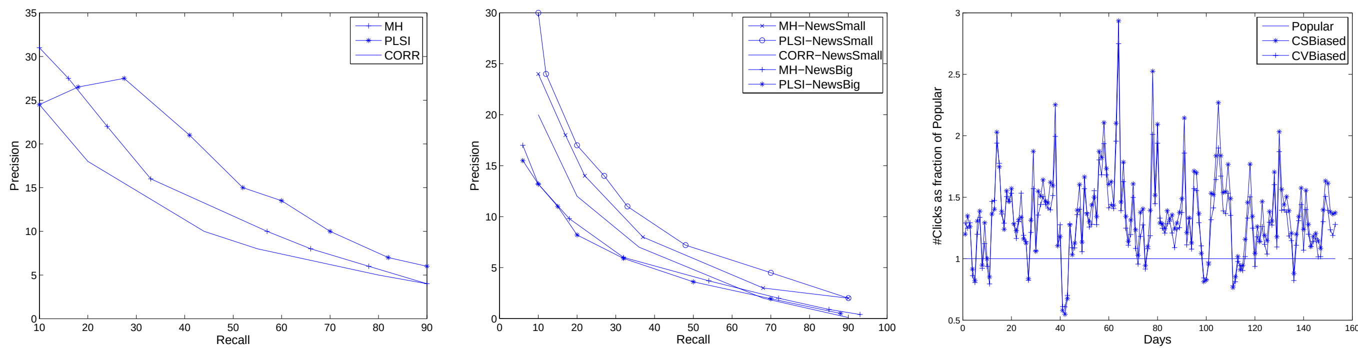
For the sake of clarity we briefly describe the three algorithms we compare:

MinHash: In the training phase, each user is clustered into 100 clusters based on her clicks. In the test phase, the inferred weight of a user u for an item s is computed as $\sum_{u_i} w(u, u_i) I_{(u_i,s)}$, where the sum is over all other users u_i, the function $w(u, u_i)$ is the similarity between two users u, u_i (proportional to the number of MinHash clusters they are together in, normalized to sum to 1 over all users for the given user u), and the indicator function $I_{(u_i,s)}$ is 1 if the user u_i has clicked on the item s and 0 otherwise.

Correlation: This memory based technique computes the similarity measure between every pair of user and combines ratings from other users weighted by similarity. The in-

[9]We ignore the top 1000 since they have an unusually large number of clicks and are suspected to be bots as opposed to real humans

[10]Precision is undefined when recall is zero and no items are recommended



(a) (b) (c)

Figure 3: (a) Precision recall curves for the MovieLens dataset. (b) Precision recall curves for the GoogleNews dataset. (c) Live traffic click ratios for different algorithms with baseline as Popular algorithm.

ferred weight of a user u for an item s, is computed as $\sum_{u_i} w(u, u_i) I_{(u_i,y)}$, where $w(u, u_i)$ is the vector similarity measure, defined as the cosine of the angle between the vectors representing the users, and the $I_{(u_i,s)}$ is the same as that described earlier for MinHash.

PLSI: In the case of PLSI, the inferred rating is simply the conditional likelihood computed as $p(s|u) = \sum_z p(s|z)p(z|u)$ (see Section 4.2)

In all of the algorithms described above, the inferred rating is a fraction between 0 and 1 that is binarized to 1 if it exceeds a certain threshold, 0 otherwise. The threshold is chosen from the set $\{10^{-x} | x \in \{0.1, 0.2, \ldots, 4\}\}$. Varying the threshold gives a precision vs. recall trade-off – the higher the threshold, higher the precision and the lower the recall.

6.4 Evaluation Results

Figures 3 (a) and (b) show the precision-recall curves for the three datasets: MovieLens, NewsSmall and NewsBig. For the **NewsBig** dataset we were unable to run the memory based algorithm as it would not scale to these numbers; keeping the data in memory for such a large dataset was not feasible, while keeping it on disk and making random disk seeks would have taken a long time. One observes that the PLSI always does the best, followed by MinHash, followed by Correlation. This shows that our algorithms, although more scalable, do not incur a loss in quality, and on the contrary do better in terms of quality. Another trend to note is that the difference in quality reduces with growing amounts of data.

6.5 Evaluation on live traffic

The previous set of evaluations were not for a dynamic itemset, which is one of the important distinguishing factors of our application. Moreover, the individual algorithms that were compared are slightly different from those that are used in our system, modified to incorporate the churn in itemset and also to have a common framework for combining different recommendation algorithms. We would like to evaluate how the overall recommendation algorithm and its individual components fare over the live traffic and also compare them with a natural candidate algorithm: recommending popular stories. As explained in Section 4, each algorithm generates a recommendation score for the candidate items that are linearly combined with appropriate weights to get the overall recommendation score. Unless otherwise specified, the weights used for combining the three individual algorithms (MinHash, PLSI, Covisit), are 1.0 for all the algorithms. An individual algorithm, in the comparisons below, is simply the overall algorithm with weights set to 1.0 for that algorithm and zero for the rest. The simple **Popular** algorithm assigns recommendation score to candidates that is equal to their age discounted click count, i.e. recent popularity. This provides a natural but fairly high benchmark to evaluate against.

How does one compare two or more recommendation algorithms on a subset of the live traffic? This is one way in which we do it: We generate a sorted ranked list from each of these algorithms and then interlace their results(e.g. to compare two algorithms A and B, we take the first result from A, followed by first result of B, followed by second result of A, and so on, removing duplicates if required), and present the interlaced list as the final recommended list to the users. To account for the position bias (users are more likely to click higher positioned stories), we cycle through the order in which we interlace the results, i.e. which algorithm goes first. The advantage of this approach is that it removes any presentation bias or position bias. We then measure which of the algorithms gets more clicks (i.e., clicks on the stories recommended by this algorithm). The premise is that users will click on stories they like and hence the ratio of the number of clicks measures their relative quality.

The click-through numbers that we report are from running our experiments over a large user fraction of the entire Google News traffic (millions of users) over a period of 5-6 months. Figure 3 (c) shows the ratio of the clicks for three competing algorithms: Overall algorithm with higher weight for covisitation (2.0 instead of 1.0) (defn. **CVBiased**), overall algorithm with higher weight for PLSI and MinHash (2.0 instead of 1.0) (defn. **CSBiased**), and the baseline **Popular** algorithm. We observe that, on an average, **CVBiased** and **CSBiased** are better by 38% as compared with the baseline **Popular**. Occasionally, when you get a popular juicy story out of Hollywood that is irresistible to our readers (e.g. Katie Holmes and Tom Cruise affair), we see that the baseline **Popular** algorithm does better.

Figure 4 shows the results of comparing **PLSI** and MinHash (**MH**) with the baseline that combines the scores from both algorithms in equal ratios. We observe that the individual algorithms are almost always better than combining them, although there is no conclusive winner between the individual ones.

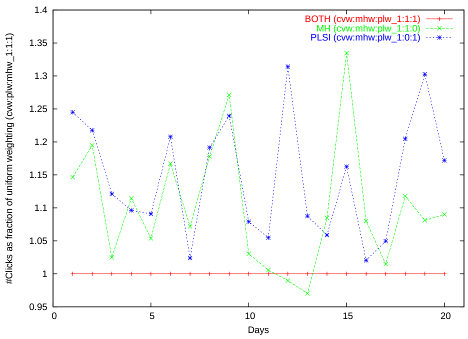


Figure 4: Live traffic click ratios for comparing PLSI and MinHash algorithms

7. CONCLUSION

In this paper, we present the algorithms behind a scalable real time recommendation engine and give the results of its evaluation on Google News. The high item churn and massive scale of the datasets involved sets our system apart from existing recommendation systems. We presented novel approaches to clustering over dynamic datasets using MinHash and PLSI, both of which were adapted to scale arbitrarily using the Mapreduce framework developed at Google. Our experimental results on several real life datsets show that this scalability does not come at the cost of quality. We experimentally demonstrated the efficacy of our recommendation engine via evaluation on live traffic, using user clickthroughs as a quality metric. The evaluation on live traffic was done over a large fraction of the Google News traffic over a period of several days. This was a clear demonstration of the scalability of the system. Our approach, which is based on collaborative filtering, is content agnostic and thus easily extendible to other domains. Future directions to explore include using suitable learning techniques to determine how to combine scores from different algorithms, and exploring the cost-benefit tradeoffs of using higher order (and directional) covisitation statistics.

8. ACKNOWLEDGMENTS

The authors would like to thank Jon McAlister for his reviews and suggestions on the overall system design and significant contributions in helping build parts of the system. The authors also thank Megan Nance for valuable suggestions on early drafts of this paper.

9. REFERENCES

[1] G. Adomavicius, and A. Tuzhilin Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. In *IEEE Transactions on Knowledge And Data Engineering*, Vol 17, No. 6, June 2005

[2] D. Blei, A. Ng, and M. Jordan Latent Dirichlet Allocation In *Journal of Machine Learning Research*, 2003.

[3] J. Breese, D. Heckerman, and C. Kadie Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proc. of the 14th Conf. on Uncertainty in Artifical Intelligence*, July 1998.

[4] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, 1998, pp. 21–29.

[5] J. Buhler Efficient large-scale sequence comparison by locality-sensitive hashing. In *Bioinformatics*, Vol. 17, pp 419 –428, 2001.

[6] M. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing*, STOC (2002).

[7] N. Cristianini, and J. Shawe-Taylor An Introduction to Support Vector Machines and Other Kernel-based Learning Methods Cambridge University Press, 1st edition (March 28, 2000).

[8] E. Cohen. Size-Estimation Framework with Applications to Transitive Closure and Reachability. *Journal of Computer and System Sciences* 55 (1997): 441–453.

[9] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding Interesting Associations without Support Pruning. In *Proc. of the 16th Intl. Conf. on Data Engineering*, (ICDE 2000).

[10] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of the 7th Symposium on Operating System Design and Implementation*, (OSDI 2006).

[11] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proc. of the 20th ACM Annual Symposium on Computational Geometry* (SOCG 2004).

[12] J. Dean, and S. Ghemawat., "MapReduce: Simplified Data Processing on Large Clusters.", In *Proc. of 6th Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, 2004.*

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proc. of the 25th Intl. Conf. on Very Large Data Bases*, VLDB(1999).

[14] T. Hofmann Latent Semantic Models for Collaborative Filtering In *ACM Transactions on Information Systems*, 2004, Vol 22(1), pp. 89–115.

[15] P. Indyk A Small Approximately Min-Wise Independent Family of Hash Functions. In *Proc. 10th Symposium on Discrete Algorithms*, SODA (1999).

[16] P. Indyk and R. Motwani. Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. In *Proc. of the 30th Annual ACM Symposium on Theory of Computing*, 1998, pp. 604–613.

[17] B. Marlin, and R. Zemel The multiple multiplicative factor model for collaborative filtering In *ACM Intl. Conf. Proceeding Series*, Vol. 69, 2004.

[18] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1985.

[19] P. Resnick, N. Iakovou, M. Sushak, P. Bergstrom, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews, In *Proc. of Computer Supported Cooperative Work Conf.* , 1994.

[20] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl Application of Dimensionality Reduction in Recommender Systems – A Case Study In *Proc. of the ACM WebKDD Workshop*, 2000.

[21] B. Sarwar, G. Karypis, J. Konstan, and J. Reidl Item-based collaborative filtering recommendation algorithms. In *Proc. of the 10th Intl. WWW Conf.* , (WWW) 2001.

[22] G. Shani, R. Brafman, and D. Heckerman, An MDP-Based Recommender System In *Proc. of the 18th Conf. Uncertainty in Artificial Intelligence*, Aug. 2002.

[23] K. Yu, X. Xu, J. Tao, M. Ester, and H. Kriegel Instance Selection Techniques for Memory-Based Collaborative Filtering In *Proc. of the Second Siam Intl. Conf. on Data Mining*, (SDM) 2002.