SHRec: Scalable Holistic Recommendation

Ahmed M. Aly, Moustafa A. Hammad, Amr Ahmed

Google Inc.

Mountain View, CA

(aaly,hammad,amra)@google.com

ABSTRACT

The problem of recommending items to users is of high practical importance. For instance, many web services try to find relevant recommendations for the users, e.g., finding relevant movies, social-media friends, restaurants, shopping items, etc. The expansion of the Web and the ever-growing number of people who use web services render the problem of recommendation challenging. The Locality Sensitive Hashing (LSH, for short) is the most known scalable technique for nearest-neighbor search in high dimensional data, and hence the LSH is widely used in most industrial recommendation systems. This paper presents an implementation of the LSH using Google's MapReduce engine. We apply the LSH to a real case study at Google, where we recommend for each web-host a set of outlinks based on the outlink similarity amongst the web-hosts. We identify some performance limitations of the LSH that occur due to specific properties in the data, and that become significant when the scale of the data is large. Furthermore, we present SHRec, a novel technique for scalable recommendation that addresses these performance limitations. Based on real deployment of both SHRec and LSH on Google's infrastructure, and using real data of the crawled Web at Google, where a sample host-level graph of 1.5 Billion web-hosts is extracted, we demonstrate that SHRec is more scalable than LSH. In particular, we show that SHRec is one order of magnitude faster than LSH while achieving better recommendation quality.

CCS CONCEPTS

•Information systems → Recommender systems; Personalization; Collaborative filtering; Clustering; Nearest-neighbor search;

ACM Reference format:

Ahmed M. Aly, Moustafa A. Hammad, Amr Ahmed. 2017. SHRec: Scalable Holistic Recommendation. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017*, 11 pages.
DOI: <http://dx.doi.org/10.1145/3085504.3085523>

1 INTRODUCTION

The unbounded growth of the Internet and its users creates numerous web services in which the content presented to a user needs to be personalized. Examples include, but are not limited to, recommending movies or videos, news feed (e.g., see [10]), shopping items (e.g., see [20]), etc.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SSDBM '17, Chicago, IL, USA

© 2017 Copyright held by the owner/author(s). 978-1-4503-5282-6/17/06...\$15.00
DOI: <http://dx.doi.org/10.1145/3085504.3085523>

User	Items (Movies)
Bob	X-Men, Gladiator, ...
Alice	Gladiator, Brave Heart, ...

Figure 1: The users table. In collaborative filtering, a self-join is required on the users table in order to find the nearest-neighbors of each user.

Collaborative filtering [6] is a technique that has been adopted by several systems to solve the problem of recommending items to users. The technique can be described as follows. Consider a table of users, where each row represents a user and her items of interest. Refer to the example of Figure 1 for illustration. The main idea in collaborative filtering is to find the nearest-neighbors of a user, say u , according to a certain similarity measure, e.g., Cosine similarity. After the nearest-neighbors are found, items are recommended to u based on its nearest-neighbors.

Our goal in this paper is to efficiently determine the users' recommendations all-at-once. The goal is motivated by the fact that most web services have strict response-time constraints, and cannot afford the online computation of recommendations when receiving requests. For these web services, it is common to pre-compute users' recommendations beforehand while periodically refreshing the data. Depending on the web service, the periodic-refresh cycle can be as short as hours or even minutes. Therefore, efficiency of recommendation is always a requirement.

As we adopt the collaborative-filtering approach, the above goal of having all-at-once (i.e., holistic) recommendation is challenged by the efficient search for the nearest-neighbors of each user. One approach to find each user's nearest-neighbors is to evaluate a self join on a database table, i.e., users table of Figure 1. Given the scale of Google's data, with Billions of users and items, performing an efficient scalable join on the users table is quite challenging. Moreover, it is also challenging to have holistic recommendation on such scale, where all the recommendations for Billions of users need to be computed at once. As we demonstrate in the next sections, one can apply several optimizations to enhance the performance of the computations, but these optimizations can lead to poor recommendation quality, or even worse, lead to lost recommendations for some users. Thus, there is always a tradeoff between the efficiency of computation and the quality of recommendation. This paper studies these challenges using real datasets and real MapReduce [11] implementation of collaborative filtering techniques.

Except for [10], most existing systems (e.g., [1, 3, 18–20, 28]) that address the problem of recommendation are not MapReduce-based and do not support the scale of operation we are targeting (Billions of users and items). [10] is a MapReduce-based system for online

recommendation of Google news. The problem addressed in [10] is slightly different from the one we address in this paper. The reason is that [10] focuses on online recommendation, while our focus in this paper is to provide a holistic recommendation model, where all the recommendations are pre-computed efficiently. The work in [10] applies the Locality Sensitive Hashing technique (LSH) using the Min-Hash scheme [7, 8].

Our case study in this paper is to perform link-based recommendation in the Web Graph. In particular, we recommend for each web-host a set of outlinks based on the similarity of the outlinks between the different web-hosts. Our case study is motivated by the fact that both the explicit and recommended link structures of the web-hosts provide a rich set of information to better understand web quality. Moreover, communicating the recommended links to Webmasters helps better structure their websites and improves the returns from their online presence.

In the case study, we use real data of the crawled Web at Google, where we extract a sample host-level graph of 1.5 Billion web-hosts. We apply the LSH to our case study, and monitor the performance. We identify some limitations that occur due to properties of the data as well as its scale. Furthermore, we present SHRec, a new scalable recommendation technique.

Most existing recommendation techniques, including LSH, try to tackle the problem by starting at each user and finding her nearest-neighbors. For instance, LSH clusters the users based on their similarity, and then for each cluster, performs an all-to-all nearest-neighbor search. The problem with this approach when deployed to our case study is that some clusters end up being large, and hence performing an $O(n^2)$ all-to-all nearest-neighbor search at these clusters is quite costly. When analyzing the data of the web-hosts, we find that the Internet contains many *hub* hosts, e.g., Facebook, Twitter, Google+, where most of the other hosts point to these hosts. These hubs lead to similarity between hosts that are not necessarily similar, and hence the clusters of hosts become very big. In particular, in our case study of link-profiling of the Web, we found that, after clustering, many clusters have nearly 10% of the data each. One approach to address this problem is to identify the hubs and isolate them from the data before running LSH. However, the problem of identifying the hubs is challenging by itself, and adds to the complexity and latency of the execution.

In contrast to LSH, SHRec does not start from a *user*, but rather starts from an *item*. The key idea in SHRec is that if two users, say u_1 and u_2 like an item, say x , then u_1 and u_2 are similar to each other with respect to x . For instance, in Figure 2, Alice and Bob both like the Gladiator movie. Hence, we conclude that Alice is a candidate to be amongst the nearest-neighbors of Bob, and at the same time, Bob is a candidate to be amongst the nearest-neighbors of Alice. In SHRec, for each item, say i , we generate the candidacy list of nearest-neighbors for each user who likes i . For each user, we aggregate the frequency count of each candidate. The nearest neighbors of each candidate are found from the most frequent candidates. Thus, in SHRec, we transform the all-nearest-neighbor search, i.e., self join, into an efficient counting process.

Our experimental results demonstrate that, for our case study, SHRec is an order of magnitude faster than LSH. In addition, we show that SHRec achieves more accurate recommendations than

Item	List of Users
X-Men	Bob, ...
Gladiator	Bob, Alice, ...
Brave Heart	Alice, ...

Figure 2: In SHRec, we process the items table instead of the users table.

that of LSH. In particular, we show that SHRec achieves 100% recall for more than 80% of the web-hosts.

The rest of this paper proceeds as follows. Section 2 outlines the scope of the problem. Section 3 describes the related work, and gives a detailed study of LSH when applied in a MapReduce pipeline. Section 4 describes SHRec. Section 5 experimentally evaluates SHRec compared to LSH by applying both techniques to the case study of recommending outlinks to each web-host in the Web graph. Section 6 includes concluding remarks.

2 PROBLEM SETTING

We consider a generic recommendation problem, where the inputs are: a) two tables: Users and Items, and b) a one-to-many relationship between these two tables, i.e., for each user, the relation describes the items that she likes. The goal is to recommend for each user a set of items based on users' similarity. For simplicity, we consider the Jaccard Similarity measure. The Jaccard similarity between two users, say A and B , is the number of common items between A and B divided by the size (i.e., cardinality) of the union of the items of A and B . For instance, if User A is interested in Items x , y , and z , and User B is interested in Items t , x , and l , then the Jaccard similarity between A and B is $\frac{1}{5}$.

We aim at providing a scalable recommendation system that can provide the recommendations to all the users in advance, i.e., precompute all the recommendations in a **holistic** manner. There are several reasons for having this requirement:

- At Google, for instance, the response time required by any of the web-services (e.g., Youtube) has to be very low. Hence, having real-time recommendation for each user is of high importance. One way to achieve that is to precompute the recommendations for all the users (e.g., Youtube users), cache these recommendations, and just retrieve these recommendations as needed.
- In our case study for link-based recommendation in the Web-Graph, having a holistic model of recommendation speeds up offline analysis and data exploration. For instance, we can compute/enhance various important signals, e.g., PageRank [23], based on the recommended links of each document. Also, comparing the recommended links with the already existing links of a document can give useful signals about the quality/trustworthiness of that document. Furthermore, having additional links for a webpage can help improve the quality of that webpage by enabling webmasters to better structure their website through adding new/better links.

3 PRELIMINARIES & RELATED WORK

The problem of recommending items for a database of users has been studied extensively in the literature (e.g., see [2]), where two main approaches have been proposed [4]: content-based filtering and collaborative filtering. In content-based filtering (e.g., [21, 24]), the goal is to recommend items for a user, say u , that are similar to those items that u has liked (or positively rated) in the past. In other words, content-based filtering is more about modeling the content of each item and measuring the similarity between these items. In contrast, in collaborative filtering (e.g., [6]), the goal is to identify those users whose tastes are similar to those of the given user, and hence recommend items that these users have liked (or positively rated). Thus, collaborative filtering is more about modeling each user and finding similarities among users.

Two shortcomings diverge us from applying content-based filtering:

- Because we intend to have a generic system that can be applied to any recommendation domain, e.g., Youtube videos, shopping items, etc., in many of these domains, it is hard to model the content of items. Moreover, in our case study of link-based recommendation in the Web, our goal is to perform recommendation for each document based on its outlinks (not based on the content of a document). Thus, content-based filtering is not applicable to our case study.
- Because in content-based filtering, recommendation is mainly based on a user's previous activities, the user is usually restricted to seeing items similar to what she already liked or positively rated. It is generally desirable to recommend items that a user has never seen before, and that is precisely why collaborative filtering has an advantage over content-based filtering. For a given user, say u , the recommendations in collaborative filtering are based on users other than u , and hence it is highly likely that these recommendations introduce novel items that have not been rated or seen before by u . Mapping this argument to our case study of outlink recommendation for the documents in the web, it is desirable to avoid the content and the outlinks that are controlled by the document, but it rather makes more sense to recommend outlinks for a certain document, say d , that are different from what d already has.

Collaborative filtering techniques can be categorized into two types: model-based (e.g., [5, 13, 14, 16]) and memory-based (e.g., [12, 22]). Below, we briefly explain each of these categories.

Model-based techniques use the past ratings of the users to learn a *model*, which is then used to predict the user ratings for the items that have not been seen by the user. For instance, [6] proposes cluster-based and Bayesian probabilistic models to estimate the user ratings for items. Other research efforts incorporate other probabilistic models, e.g., latent semantic indexing (LSI) [27], probabilistic latent semantic indexing (PLSI) [17], Matrix Factorization [26, 29]. The main drawback of these model-based algorithms is that they are computationally expensive and are not straightforward to debug, i.e., attribute the recommendation to a specific set of users or items.

Memory-based techniques try to make rating predictions for a given user, say u , based on the ratings of the top- k similar users to u . The pairwise similarity between users can be measured in terms of any distance metric, e.g., Cosine similarity, Jaccard similarity, etc. Thus, the main goal in the memory-based techniques is to find the k -nearest-neighbors of each user. This task is very challenging given the scale of the problem, and given our holistic model of computation, where we need to perform recommendations for all users, i.e., perform an all-nearest-neighbor search.

[9] presents a memory-based technique for collaborative filtering that is based on aggregation. The main idea in [9] is to maintain inverted files, i.e., for each item, say i , users who like i belong to i 's inverted file. Given a user, say u , [9] scans the inverted files that contain u 's items, and determines the similarity between u and other users via aggregation. Similar to [9], SHRec employs inverted files. However, SHRec distinguishes itself by: 1) being holistic, i.e., SHRec has the ability to find recommendations for all users rather than a single user as in [9], and 2) being scalable, i.e., SHRec is based on MapReduce and shares computations among users, while the techniques in [9] are based on a single machine implementation. As we demonstrate in Section 4 it is not obvious to extend [9] to support holistic recommendation on a scalable framework like MapReduce because: 1) the information necessary for correctly computing the similarity scores (e.g., Jaccard) among users is hard to maintain and store in a distributed environment and across a pipeline of MapReduce jobs, and 2) the optimizations in [9], e.g., the inverse user frequency and the early termination heuristics, can lead to lost recommendations, which contradicts the motivation of holistic recommendation.

To the best of our knowledge, the most scalable existing solution to the problem of nearest-neighbor search in high-dimensional data is the (LSH) technique, which we explain below.

Locality Sensitive Hashing (LSH)

In high-dimensional data, finding the nearest-neighbors of a certain data point is a challenging task. This is due to the curse of dimensionality [15], where in most of the cases, the entire space needs to be scanned to ensure the correct determination of the nearest-neighbors of a point. To address this challenge, LSH [15] was introduced to efficiently find an approximate solution to the nearest-neighbor search problem. The key idea of LSH is to hash the data points multiple times into multiple buckets, where each bucket corresponds to a different hashing function. The hashing aims at maximizing the likelihood that two different points that are similar will fall within the same bucket. Different LSH schemes exist for the different similarity measures, e.g., Hamming distance, Cosine similarity, Jaccard similarity, etc. Min-Hashing [7, 8] is an LSH scheme that is well-suited for the Jaccard similarity.

The basic idea in the Min-Hashing scheme is to hash the users such that the probability that two users will have the same hash value is equal to their Jaccard similarity. Because multiple hash functions are used and each user is hashed multiple times, Min-Hashing can be viewed as a clustering algorithm, where two users belong to the same cluster (i.e., bucket) with probability equals to their Jaccard similarity.

Being our baseline algorithm, below, we give a detailed description of our implementation of LSH as in [10, 25]. Figure 3 depicts a MapReduce pipeline of our implementation of LSH to find the nearest-neighbors of each user and perform recommendation.

Figure 3: The MapReduce pipeline of LSH.

In the first MapReduce job, the goal is to assign each user to a set of clusters. Given the users data, where for each user, the items she likes are given as a list (i.e., vector), a Min-Hash signature vector is computed for each user using projection. Projection is done by performing a dot product between the liked items vector of the

Figure 4: The banding of the Min-Hash signature matrix in LSH.

given user with a randomly generated vector. The challenge relies in the fact that the length of the randomly generated vector is huge (because the number of items is in the order of Billions). Generating a random vector of such size is quite costly and impractical. To solve this issue, [10] observes that the items ratings vector is sparse, i.e., the number of items liked by a given user is usually low. Hence, it is redundant to generate a random vector of the size of all items while eventually, it will be multiplied mostly by zeros during the dot product operation. Thus, it is sufficient to generate only the required part(s) of the random vector, i.e., those that correspond to non-zeros in the items vector. To achieve that in a parallel platform (i.e., in MapReduce), we use a fixed seed at each mapper to generate the random vector entries. The use of fixed seeds ensures that an item in the items vector gets the same random value in the generated vector at each of the mappers in the Map phase.

LSH divides the signature vectors of all users into bands, and the similarity between the users is determined based on these bands. Refer to Figure 4 for illustration. LSH hashes each user multiple times into clusters based on the values in each band. The rationale behind this banding can be explained as follows. Two users will be hashed to the same cluster if they have the same Min-Hash signatures at any given band. Observe that although u_1 and u_2 in Figure 4 do not have similar signatures in the top band, this does not mean that they will never be hashed to the same cluster. In the lower band, u_1 and u_2 have similar signatures $\langle 1, 0 \rangle$, and hence they will be hashed to the same clusters. Thus, the banding technique allows multiple chances for two similar users to be hashed to the same cluster. Also, it minimizes the chance of two dissimilar users to be hashed to the same cluster.

In the first MapReduce job of the pipeline (see Figure 3(a)), the Map phase determines the Min-Hash signature of each user and outputs for each band, a key-value pair: (cluster id, user id). Procedure 1 gives pseudocode of the Map phase. In the Reduce phase, all the users are grouped by the cluster id.

The second MapReduce job (see Figure 3(b)) reads the contents (users) of each cluster and performs an all-nearest-neighbor search to find for each user her k -nearest-neighbors. Notice that these k -nearest-neighbors are local w.r.t. the given cluster because each user belongs to multiple clusters. Hence, in the Reduce phase, LSH aggregates the local nearest-neighbors of each user and determines the global nearest-neighbors across all clusters. Once the k -nearest-neighbors of a user are determined, recommendation is performed from the items of these neighbors. Procedures 2 and 3 give pseudocode of the Map and Reduce phases.

Procedure 1 LSH - Clustering - Map

Require: Key: a user, say u . Value = $iList$ liked by User u . b : number of bands.

```

1:  $signature \leftarrow minhash(iList)$  // returns a bit-vector
2:  $bandsList \leftarrow$  divide  $signature$  into  $b$  bands
3: for each  $band \in bandsList$  do
4:    $clusterID \leftarrow hash(band)$ 
5:   Output ( $clusterID, u$ )
6: end for
```

Procedure 2 LSH - Nearest-Neighbor Search - Map

Require: Key = $clusterID$, say C . Value = $uList$: list of users in Cluster C . k : value of k in nearest-neighbor search.

```

1: for each user  $u \in uList$  do
2:    $priorityQueue \leftarrow \emptyset$ 
3:   for each user  $v \in uList$  do
4:      $similarity = Jaccard(u, v)$ 
5:      $priorityQueue.insert(v, similarity)$ 
6:   end for
7:   Output ( $u, priorityQueue.getTop(k)$ )
8: end for
```

Two performance issues limit the scalability and practicality of LSH given our problem settings:

- The all-nearest-neighbor search within each cluster is quite costly (of $O(n^2)$, where n is the number of users in the cluster). Observe that all the users within a cluster are processed in one single Map job (refer to Figure 3(b)). Thus, unless the cluster sizes are small, the whole solution loses its parallelism.
- In many application scenarios, the data may not be *separable* into small clusters. Many large clusters emerge, which slow down the all-nearest-neighbor search. In particular, in our case study of link-profiling of the Web, we found that, after clustering, many clusters have nearly 10% of the data each. Because of dealing with Billions of documents, performing a holistic $O(n^2)$ all-nearest-neighbor search at the Mapper corresponding to each of these clusters is impractical.

In the next section, we present our proposed solution that avoids the above limitations.

Procedure 3 LSH - Nearest-Neighbor Search - Reduce

Require: Key = a user, say u . Value = $localsList$: lists of local nearest-neighbors to User u gathered from all clusters. k : value of k in nearest-neighbor search.

```

1: // Get the global nearest-neighbors of  $u$  from the local nearest-
   neighbors at each cluster:
2:  $priorityQueue \leftarrow \emptyset$ 
3:  $recommendation \leftarrow \emptyset$ 
4: for each list  $l \in localsList$  do
5:   for each user  $v \in l$  do
6:     //  $v.similarity$  with  $u$  is already computed by Line 4 in
       Procedure 2
7:      $priorityQueue.insert(v, v.similarity)$ 
8:   end for
9: end for
10:  $nearestNeighbors \leftarrow priorityQueue.getTop(k)$ 
11: for each  $neighbor \in nearestNeighbors$  do
12:   // Add recommendations from items liked by neighbors.
13:    $recommendation.add(neighbor.iList)$ 
14: end for
15: Output ( $u, recommendation$ )
```

4 SHREC

In this section, we present our new technique SHRec, a scalable solution for the user-item recommendation problem. Unlike LSH, where processing starts with a user, in SHRec, processing starts with an item. This model of processing enables SHRec to find the nearest-neighbors of each user without the need to perform the costly $O(n^2)$ user-to-user comparisons as in LSH. In contrast to LSH, SHRec counts the common items between the users, which is an efficient operation.

The key idea of SHRec can be described as follows. If a certain item, say i , is liked (or positively rated) by two (or more) different users, say u_1 and u_2 , then we conclude that u_1 and u_2 are similar, and hence are candidates to be nearest-neighbors to each others. In other words, an item that is *common* between two (or more) different users contributes to their similarity. The more the common items are, the higher the Jaccard similarity is between the users. SHRec efficiently aggregates the common items between the different users through simple counting. Processing in SHRec is divided into two stages; namely the Inversion stage and the Aggregation stage, where each stage is implemented as a MapReduce job. Figure 5 gives an overview of the MapReduce pipeline of SHRec.

Inversion

Given the users and the list of items that each user likes, the goal of this stage of processing is to determine the inverse relation, i.e., for each item, the users who like it. This stage is implemented as a MapReduce job, where in the Map phase, all users are processed; given a user, say u and her liked items list, e.g., (a, b, c) , the Map phase outputs pairs on the form (a, u) , (b, u) , and (c, u) . In the Reduce phase, all pairs are grouped by the item id. The output of this MapReduce job is for each item, say i , the users who like i . We refer to the list of users who like an item as the users-like list.

Figure 5: The MapReduce pipeline of SHRec.

Note that the number of items that each user likes is encoded for each user in the users-like list. This information is used in the Aggregation stage (that we describe next) in order to compute the Jaccard similarity between the different users. In a similar way, this information can also be used to compute other similarity measures, e.g., Cosine similarity.

Figure 6: The k -nearest-neighbor (k NN, for short) candidates of Users u_1, u_2 , and u_3 who like Item i .

Observe that in some cases, the output of the Inversion stage might be already available depending on how the user-item relationship is represented. In particular, the data might be represented in terms of items, and for each item, the list of users who liked/bought that item is maintained. In this case, the Inversion stage is not needed. However, if the data is represented in terms of users, i.e., the Inversion stage is inevitable, the overall performance of SHRec is better than that of LSH as we demonstrate in Section 5.

Aggregation

The Aggregation stage is the heart of SHRec and is also implemented as a MapReduce job. In the Map phase, given the output of the Inversion stage, we process each item as follows. For each item, say i , and its users-like list, e.g., (u_1, u_2) , we determine that u_1 is a candidate k -nearest-neighbor to u_2 because both u_1 and u_2 like the same given item i . Similarly, we determine that u_2 is a candidate k -nearest-neighbor to u_1 . The idea is generalized for any number of users in the users-like list corresponding to the given item. In particular, we generate all candidates for each user according to the users-like list. Refer to Figure 6 for illustration, where three users u_1, u_2 , and u_3 like the same item, so, we output the k -nearest-neighbor candidates of each user as shown in the figure.

In the Reduce phase, we group the nearest-neighbor candidates based on the user id. For each user, say u , we aggregate (i.e., count) the number of occurrences, say o , of each candidate user, say v , of u . Thus, o represents the number of common items between u , and v .

Computing the Similarity

In order to compute the Jaccard similarity between u and v , we need to divide two components:

- (1) the intersection of the liked items by both u , and v , and
- (2) the union of the liked items by both u and v .

The first component (i.e., the intersection) is available as the number of occurrences o . The second component (i.e., the union) is computed knowing the number of items liked by u , say $|u|$, and the number of items liked by v , say $|v|$. Recall that these numbers are encoded in the users-like list of each item as discussed in the Inversion stage. In particular, we compute the second component (i.e., the union) as $(|u| + |v| - o)$. Thus, the Jaccard similarity between u and v is computed as $\frac{o}{|u| + |v| - o}$. Finally, we output for each user id, the top- k candidates based on the Jaccard similarity measure, and

perform recommendation based on these top-k candidates. Observe that other similarity measures, e.g., the Cosine similarity, can be computed using the same encoded information, but with a different formulae.

We use a combiner to speed up the aggregation (i.e., counting) stage. A combiner can be viewed as a local reducer at each mapper. Usually, a combiner enhances the performance of a MapReduce job by minimizing the communication between the mappers and the reducers. In the Aggregation stage of SHRec, the mappers compute local-aggregate counts for each user, and pass on these aggregates to the Reduce phase, which computes the global aggregates.

Being at the heart of the pipeline of SHRec, the Aggregation stage transforms the problem of finding the similarity between users to a counting process, which is much more efficient than the costly $O(n^2)$ comparisons in LSH. However, when experimenting with our case study, we observe that for a given item, the size of the users-list can be relatively large¹, and hence computing all the possible candidacy pairs is costly. To address this issue, we apply the *inverse user frequency*[6] optimization.

Inverse-User-Frequency. Inverse-user-frequency is a well-known technique that enhances the results of collaborative filtering. The technique relates to the information retrieval domain, where word frequencies are typically modified by the *inverse-document-frequency* in order to reduce the weights of commonly occurring words. The rationale is that a word that exists in any document is not useful to identify the topic of a document. In contrast, a rare word is more representative of a topic, and adds to the uniqueness of the enclosing document. Using the same idea in collaborative filtering, [6] shows that items that are liked by all (or many) users are not helpful in representing the similarity between users. However, items that are less commonly liked amongst users are more representative. [6] defines the inverse-user-frequency of an item, say i , as $f_i = \log(\frac{n}{n_i})$, where n is the total number of users, and n_i is the number of users who like Item i . Observe that the more the users like Item i , the lower the weight of the inverse-user-frequency is, and vice versa. Also, observe that if an item is liked by most (all) users, then its inverse-frequency weight tends to be zero (or negligible).

We apply the concept of inverse-user-frequency to our solution. In particular, we compute the inverse-user-frequency of each item while processing its users-like list. If the number of users who like a certain item is large, we ignore that item because it has negligible weight. We use a user-defined parameter to set the threshold of the number of users according to which an item is ignored. We refer to this parameter as the *ignore-threshold*². If an item is liked by a number of users that is greater than the ignore-threshold, then that item is ignored.

An important observation is that when an item is ignored due to its high user-frequency, its users get no recommendation based on that item. If one or more of these users appear at no other items, then these users will not get any recommendation at all. Unless we carefully enumerate the candidate nearest neighbors for each user,

it is likely that we lose the objective of holistic recommendation. This fact was confirmed in our case study using our real data, where we observed that some web hosts are getting no recommendation for the same reason. To address this problem, whenever an item is ignored due to its high user-frequency, for each user, we choose k random candidates from the users-like list. Hence, each user is guaranteed to get at least k random candidates throughout the execution pipeline. Thus, each user is guaranteed a recommendation, and hence the goal of holistic recommendation is achieved.

Procedures 4 and 5 give pseudocode of the Map and Reduce phases of the Aggregation stage in SHRec.

Procedure 4 SHRec - Aggregation - Map

Require: Key: an item, say i . Value = $uList$: list of users who like Item i . nT : Total number of items in database.

```

1: if  $uList.length < ignoreThreshold$  then
2:   for each user  $u \in uList$  do
3:     for each user  $v \in uList$  do
4:       Output ( $u, v$ )
5:     end for
6:   end for
7: else
8:   for each user  $u \in uList$  do
9:      $rand \leftarrow (k + 1)$  random candidates from  $uList$ 
10:    for each user  $v \in rand$  do
11:      Output( $u, v$ )
12:    end for
13:  end for
14: end if
```

Procedure 5 SHRec - Aggregation - Reduce

Require: Key = u : user. Value = $candidatesList$: list of candidates for u .

```

1: // Aggregate the frequency of occurrence of all candidates into
   a hashtable keyed by the candidate.
2:  $hashtable \leftarrow \emptyset$ 
3: for each candidate  $c \in candidatesList$  do
4:    $hashtable[c]++ = 1$ 
5: end for
6:  $priorityQueue \leftarrow \emptyset$ 
7: for each candidate  $c \in candidates$  do
8:    $intersection \leftarrow hashtable[c]$  // i.e.,  $|c \cap u|$ 
9:    $similarity \leftarrow \frac{intersection}{|u| + |c| - intersection}$  // Apply the formulae
10:   $priorityQueue.add(c, similarity)$ 
11: end for
12:  $nearestNeighbors \leftarrow priorityQueue.getTop(k)$ 
13: for each  $neighbor \in nearestNeighbors$  do
14:   // Add recommendations from items liked by neighbors.
15:    $recommendation.add(neighbor.iList)$ 
16: end for
17: Output ( $u, recommendation$ )
```

¹The size of the users-like list reaches more than 1 Million in our case study. For instance, a popular web-host, e.g., Youtube.com, has Millions of web-hosts pointing to it.

²Empirically, we consider a threshold of 1000 in our case study.

Distributed Lookups

Observe that the final step in SHRec's pipeline is to recommend items for a user, say u , based on the liked items of u 's nearest-neighbors. Refer to Line 15 in Procedure 5. In order to achieve this, the items liked by each of these nearest-neighbors need to be available at this point in the pipeline. One solution is to maintain the list of items liked by each user throughout the pipeline, i.e., the Inversion and Aggregation stages. Although it is temporary, maintaining such large amount of data throughout the pipeline requires significant amount of temporary storage as well as processing time to read/write such data.

To address the above issue, instead of maintaining the list of items liked by each user throughout the pipeline, the items liked by each user are stored in a distributed hash table, where the key is the user id and the value is the list of liked items. During the recommendation step in SHRec, i.e., Line 15 in Procedure 5, we do hash lookups to determine the items liked by each user.

Observe that at the final stage of the LSH pipeline, recommendation is performed. To have a fair comparison with LSH, we also apply the distributed lookups optimization to LSH pipeline.

5 EXPERIMENTS

5.1 Case Study

Our case study in this paper is to perform link-based recommendations for the documents in the Web-Graph. As discussed in Section 2, having recommendations for each web-document in the web has several applications, e.g., computing various signals about the web-documents, and also enhancing the quality of websites by introducing new/recommended outlinks.

The Web-Graph can be viewed as a set of documents and the interconnecting links between these documents. Each document has an id and a set of outlinks to other documents. Our goal is to perform recommendations based on the outlinks comprising the inter-connectivity of the graph. We apply the Jaccard Similarity to measure the similarity between the documents in the Web-Graph. In the context of outlinks, the Jaccard Similarity between two documents, say A and B , is the number of common outlinks between A and B divided by the size (i.e., cardinality) of the union of the outlinks of A and B .

In order to have meaningful and useful recommendations, we do not consider the document-level Web-Graph, but rather consider the host-level Web-Graph. We do not directly consider the outlinks between individual web-pages; instead, we consider the outlinks between different hosts. For instance, $http : //xyz.com/1/2/abc.html$ and $http : //xyz.com/3/1/lmn.html$ are considered as two different nodes in the document-level graph, but are considered as only one node in the host-level graph because they belong to the same host $xyz.com$. The reason for choosing to employ the host-level graph in recommendation is that web-pages, that are within the same host, usually link to each other. Hence, two documents within the same host have many common links, i.e., have high similarity. Any collaborative filtering technique that is to be applied on the document-level graph will recommend for each document, say d , outlinks that are within d 's host.

To determine the host-level graph, we process the document-level graph and aggregate the outlinks based on each distinct host.

In particular, we apply a MapReduce job on all the documents. In the Map phase, we read all the documents in the document-level graph, where the key is a document, and the value is the document's outlinks. We extract the host information from the url of each document, e.g., $xyz.com$ from $http : //www.xyz.com/1/2.html$. If a document, say $http : //xyz.com/1.html$ has an outlink to another document, say $http : //abc.com/2.html$, then we determine that there is an outlink from Host $xyz.com$ to Host $abc.com$, and hence we emit key-value pair $(xyz.com, abc.com)$. In the Reduce phase, we aggregate all the outlinks for each distinct host as a list of strings.

Figure 7: A web-host, its original outlinks, and its recommended outlinks.

Figure 8: The outlink frequency histogram of the host-level Web-Graph.

Given the host-level graph, we recommend for each web-host, a set of other web-hosts based on the Jaccard Similarity amongst the web-hosts. Figure 7 gives an overview of our case study. For the given web-host of Figure 7(a) that has outlinks to web-hosts a , b , and c , the recommended outlinks are a , y , z , and c . Some of the existing outlinks (e.g., a and c) may be in the recommended links, while others (e.g., b) may not. Also, observe that introducing new outlinks (e.g., y and z) in the recommended links can be useful to detect new signals about the web-hosts.

Although reducing the problem to the host-level graph decreases the scale of the problem, the scale challenge still exists. Figure 8 depicts the outlink frequency histogram of a sample host-level Web-Graph that has a total of almost 1.5 Billion web-hosts. The figure shows how many web-hosts exist (y-axis) for each number of outlinks (x-axis). For instance, more than 1.4 Billion web-hosts have 1 to 10 outlinks, about 48 Million web-hosts have 10 to 100 outlinks, etc.

5.2 Experimental Setup

We implemented both the LSH and SHRec techniques as explained in Sections 3 and 4, respectively. Experiments are conducted at Google using a cluster of 2000 machines running Google’s MapReduce [11], where each machine has 2 GB of main-memory and 5 GB of disk space. The dataset we are using is based on Google’s crawled web-documents, out of which, we extract a sample host-level graph of 1.5 Billion nodes.

In all the experiments, we set the value of k to 10 in the nearest-neighbor search. We set the required number of recommended outlinks per web-host to be 130% of the existing number of outlinks in a web-host. For instance, if a certain web-host, say h , already has 10 outlinks, then we recommend 13 outlinks for h from its 10 nearest-neighbors. If the nearest-neighbors of h have less than 13 outlinks, then we just copy all the outlinks in the nearest-neighbors as recommended outlinks for h .

For LSH, we represent the Min-Hash signature as a bit vector. In our experiments, we set the number of bits in the signature to be 500 bits, and the number of bands to be 10. We tried different combinations for the signature size and the number of bands in the LSH technique, and these values were the ones that achieved the best possible performance for the given data. In SHRec, we set the ignore-threshold to 1000.

5.3 Scalability

In this experiment, we measure the execution time of SHRec and LSH. We execute both LSH and SHRec on the 1.5 Billion nodes host-level graph, and monitor the execution time. We found LSH to be slow due to its $O(n^2)$ all-nearest-neighbor search that is performed at each of the clusters without any parallelism (i.e., at a single mapper) (refer to Figure 3(b)). We analyzed the cluster sizes in LSH. Figure 9 gives the histogram of the cluster sizes reported by LSH. The majority of clusters are of small size; more than 86 Million clusters were of size < 10 . However, more than 100 thousand clusters are of size > 100 . Furthermore, more than 5000 clusters are of size > 1 Million.

Table 1: The execution time of SHRec and LSH. SHRec is almost an order of magnitude faster than LSH.

Number of Web-Hosts	Execution Time	
	SHRec	LSH
1.5 Billion	5 hrs, 53 min	41 hrs, 16 min

To speedup the execution of LSH, we decided to avoid the costly $O(n^2)$ search in each of the large clusters. In particular, for clusters of size > 10000 , instead of computing the nearest-neighbors for each web-host by checking it against all the web-hosts in the cluster, we simply select k random web-hosts as the k -nearest-neighbors. However, even with this change, SHRec is found to be an order of magnitude faster than LSH as Table 1 demonstrates.

Figure 9: Cluster Size Histogram in LSH. More than 5000 clusters are of size > 1 Million each, i.e., contain more than 1 million hosts each. Performing an $O(n^2)$ all-nearest-neighbor search in each of these clusters is costly.

In contrast to LSH, SHRec avoids the quadratic search by transforming the problem to a counting problem, where we simply aggregate the candidate pairs corresponding to each item, and hence SHRec is efficient.

5.4 Quality of Recommendation

In this experiment, we measure the quality of the recommendation of SHRec and LSH. Our performance measure in this experiment is the recall. Measuring the recall is challenging because there is no ground-truth in recommendation. To have a meaningful evaluation, we consider the existing links in each web-host as our ground-truth. In particular, for a given web-host, say h , the recall is computed as $\frac{n_e}{n_r}$, where n_e is the number of outlinks that already exist in h and also exist in the recommended outlinks for h , while n_r is the number of outlinks in h . The rationale is that if the recommended outlinks contain the existing links (i.e., high recall), then this adds to the trustworthiness of our recommendation. In contrast, if the recommended links are oblivious to the existing links, then this should indicate poor recommendation.

Table 2 gives the recall values for SHRec compared to those of LSH. The figure exploits four different categories for the different outlink frequencies per web-host, i.e., for web-hosts with 1-10 outlinks, web-hosts with 10-100 outlinks, etc. Each entry in the figure corresponds to the percentage of web-hosts for which a certain recall value is achieved. Each percentage value is measured as the

Table 2: Recall of SHRec and LSH for different outlink-count categories. Each entry is the ratio of the web-hosts achieving a certain recall value. The ratio is computed relative to the total number of web-hosts (1.5 Billion). SHRec achieves a recall of 1 for more than 80% of the web-hosts, while LSH achieves a recall of 1 for only 72% of the hosts.

Recall Histogram											
Outlink-Count Category											
		1 - 10		10 -100		100 - 1000		1000 - 10000		SUM	
		SHRec	LSH	SHRec	LSH	SHRec	LSH	SHRec	LSH	SHRec	LSH
Recall	0.01	8.145%	21.261%	0.153%	1.647%	0.012%	0.191%	0.000%	0.016%	8.310%	23.115%
	0.1	0.015%	0.016%	0.243%	0.832%	0.084%	0.159%	0.004%	0.008%	0.346%	1.016%
	0.2	0.182%	0.388%	0.479%	0.283%	0.076%	0.012%	0.004%	0.001%	0.741%	0.684%
	0.3	0.521%	0.547%	0.475%	0.083%	0.048%	0.003%	0.003%	0.000%	1.047%	0.634%
	0.4	0.787%	0.514%	0.384%	0.028%	0.040%	0.002%	0.003%	0.000%	1.213%	0.544%
	0.5	0.327%	0.077%	0.286%	0.013%	0.031%	0.002%	0.002%	0.000%	0.647%	0.092%
	0.6	2.875%	1.046%	0.256%	0.012%	0.024%	0.002%	0.002%	0.000%	3.157%	1.060%
	0.7	1.274%	0.216%	0.193%	0.013%	0.020%	0.002%	0.002%	0.000%	1.488%	0.231%
	0.8	0.506%	0.068%	0.159%	0.017%	0.017%	0.003%	0.001%	0.000%	0.683%	0.088%
	0.9	0.391%	0.064%	0.164%	0.030%	0.017%	0.004%	0.001%	0.000%	0.573%	0.099%
SUM		96.406100%		3.110245%		0.406202%		0.026048%		100%	

Table 3: Recall of SHRec and LSH for different outlink-count categories. Each entry is the ratio of the web-hosts achieving a certain recall value. The ratio is computed relative to the number of web-hosts in each outlink-count category.

Recall Histogram									
Outlink-Count Category									
		1 - 10		10 -100		100 - 1000		1000 - 10000	
		SHRec	LSH	SHRec	LSH	SHRec	LSH	SHRec	LSH
Recall	0.01	8.44%	22.04%	4.91%	52.95%	3.02%	47.01%	15.43%	62.58%
	0.1	0.02%	0.02%	7.81%	26.74%	20.59%	39.24%	7.34%	32.13%
	0.2	0.19%	0.40%	15.40%	9.10%	18.60%	3.01%	9.19%	1.98%
	0.3	0.54%	0.57%	15.27%	2.67%	11.75%	0.80%	11.00%	0.45%
	0.4	0.82%	0.53%	12.35%	0.89%	9.75%	0.51%	12.17%	0.20%
	0.5	0.34%	0.08%	9.21%	0.42%	7.58%	0.52%	10.52%	0.16%
	0.6	2.98%	1.08%	8.22%	0.39%	5.99%	0.46%	8.08%	0.17%
	0.7	1.32%	0.22%	6.21%	0.42%	4.82%	0.49%	6.84%	0.19%
	0.8	0.52%	0.07%	5.10%	0.54%	4.23%	0.71%	6.08%	0.29%
	0.9	0.41%	0.07%	5.26%	0.97%	4.23%	1.10%	5.55%	0.50%
SUM		100%	100%	100%	100%	100%	100%	100%	100%

number of web-hosts achieving a certain recall value divided by the total number of web-hosts (1.5 Billion). As the figure demonstrates, SHRec achieves better recall than LSH. On the one hand, for a recall

of 0.01 (i.e., low recall), there is about 8% of the web-hosts in SHRec, and about 23% of the web-hosts in LSH. On the other hand, for a recall of 1 (i.e., high recall), there is about 81% of the web-hosts

in SHRec, and about 72% of the web-hosts in LSH. In other words, for low recall values, SHRec has less percentage of web-hosts than LSH, and for high recall values, SHRec has higher percentage of web-hosts than LSH; this occurs at (almost) each entry in the figure.

Similarly to Table 2, Table 3 gives the recall values for SHRec compared to those of LSH, but each percentage value in the figure is measured as the number of web-hosts in a certain outlink-count category achieving a certain recall divided by the number of web-hosts within that category. For instance, for the category of hosts with outlink-count 1-10, SHRec achieves a recall of 0.01 for 8% of the web-hosts in this category, while LSH achieves a recall of 0.01 for 22% of the web-hosts in this category. As the figure demonstrates, SHRec achieves better recall than LSH for each outlink-count category. On the one hand, for recall of 0.01 (i.e., low recall), SHRec has (8%, 4%, 3%, and 15%) of the web-hosts in each of the (1-10, 10-100, 100-1000, and 1000-10000, respectively) categories, while LSH has (22%, 52%, 47%, and 62%) of the web-hosts in each of these categories. On the other hand, for recall of 1 (i.e., high recall), SHRec has (84%, 10%, 9%, and 7%) of the web-hosts in each of the (1-10, 10-100, 100-1000, and 1000-10000, respectively), while LSH has (74%, 4%, 6%, and 1%) of the web-hosts in each of these categories. Observe that the majority (96%) of web-hosts fall in the outlink-count category of (1-10), and that SHRec is superior to LSH in this category w.r.t. the recall values. Furthermore, SHRec has better recall values at each of the other three categories.

One reason for the relatively low accuracy of LSH can be attributed to the way we handle large clusters, where for each web-host in a large cluster (i.e., of size > 10000) we select k random web-hosts as the k -nearest-neighbors in order to avoid the costly $O(n^2)$ nearest-neighbor search. This random selection has a negative impact on the accuracy of LSH.

6 CONCLUDING REMARKS

This paper presents an industrial MapReduce-based implementation of the LSH technique that is applied to a real host-level graph of 1.5 Billion hosts crawled at Google. The paper identifies the performance limitations of LSH, and presents SHRec, a new scalable technique that addresses these limitations. Leveraging the advantages of nearest-neighbor collaborative filtering and MapReduce, SHRec is composed of two stages; namely the Inversion stage and the Aggregation stage. The main insight behind SHRec is to transform (i.e., invert) the representation of the data into pairs of (item, users-like-list) instead of the traditional representation of (user, items-list) pairs. This transformation enables SHRec to efficiently aggregate the common items between the users (in the Aggregation stage), and hence determine the nearest-neighbors of each user through a simple counting procedure. In our case study, we compare SHRec against LSH, and we find that SHRec is an order of magnitude faster while achieving better recommendation quality.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support provided by Andrei Broder, D. Sivakumar, and Raghu Hulikal Keshavan during paper preparation.

REFERENCES

- [1] Zeinab Abbassi and Laks V. S. Lakshmanan. 2009. On Efficient Recommendations for Online Exchange Markets. In *ICDE*. 712–723.
- [2] Gediminas Adomavicius and Alexander Tuzhilin. 2005. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *IEEE Trans. Knowl. Data Eng.* 17, 6 (2005), 734–749.
- [3] Sihem Amer-Yahia, Alban Galland, Julia Stoyanovich, and Cong Yu. 2008. From del.icio.us to x.qui.site: recommendations in social tagging sites. In *SIGMOD*. 1323–1326.
- [4] Marko Balabanović and Yoav Shoham. 1997. Fab: Content-based, Collaborative Recommendation. *Commun. ACM* 40, 3 (1997), 66–72.
- [5] Daniel Billsus and Michael J. Pazzani. 1998. Learning Collaborative Information Filters. In *ICML*. 46–54.
- [6] John S. Breese, David Heckerman, and Carl Myers Kadie. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *UAI*. 43–52.
- [7] Andrei Z. Broder. 1997. On the Resemblance and Containment of Documents. In *In Compression and Complexity of Sequences (SEQUENCES'97)*. 21–29.
- [8] Edith Cohen. 1997. Size-estimation Framework with Applications to Transitive Closure and Reachability. *J. Comput. Syst. Sci.* 55, 3 (Dec. 1997), 441–453.
- [9] Rickard Cöster and Martin Svensson. 2002. Inverted file search algorithms for collaborative filtering. In *SIGIR*. 246–252.
- [10] Abhinandan Das, Mayur Datar, Ashutosh Garg, and ShyamSundar Rajaram. 2007. Google news personalization: scalable online collaborative filtering. In *WWW*. 271–280.
- [11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*.
- [12] Joaquin Delgado and Naohiro Ishii. 1999. Memory-Based Weighted-Majority Prediction for Recommender Systems. In *SIGIR*.
- [13] L. Getoor and M. Sahami. 1999. Using probabilistic relational models for collaborative filtering. In *Workshop on Web Usage Analysis and User Profiling (WEBKDD'99)*.
- [14] Kenneth Y. Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. 2001. Eigentaste: A Constant Time Collaborative Filtering Algorithm. *Inf. Retr.* 4, 2 (2001), 133–151.
- [15] Sarel Har-Peled, Piotr Indyk, and Rajeev Motwani. 2012. Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. *Theory of Computing* 8, 1 (2012), 321–350.
- [16] Thomas Hofmann. 2003. Collaborative filtering via gaussian probabilistic latent semantic analysis. In *SIGIR*. 259–266.
- [17] Thomas Hofmann. 2004. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.* 22, 1 (2004), 89–115.
- [18] Bhargav Kanagal, Amr Ahmed, Sandeep Pandey, Vanja Josifovski, Jeffrey Yuan, and Lluís Garcia Pueyo. 2012. Supercharging Recommender Systems using Taxonomies for Learning User Purchase Behavior. *PVLDB* 5, 10 (2012), 956–967.
- [19] Georgia Koutrika, Benjamin Bercovitz, and Hector Garcia-Molina. 2009. FlexRecs: expressing and combining flexible recommendations. In *SIGMOD*. 745–758.
- [20] G. Linden, B. Smith, and J. York. 2003. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE* 7, 1 (Jan 2003), 76–80.
- [21] Raymond J. Mooney and Lorie Roy. 2000. Content-based book recommending using learning for text categorization. In *ACM DL*. 195–204.
- [22] Atsuyoshi Nakamura and Naoki Abe. 1998. Collaborative Filtering Using Weighted Majority Prediction Algorithms. In *ICML*. 395–403.
- [23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. Previous number = SIDL-WP-1999-0120.
- [24] Michael J. Pazzani and Daniel Billsus. 1997. Learning and Revising User Profiles: The Identification of Interesting Web Sites. *Machine Learning* 27, 3 (1997), 313–331.
- [25] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA.
- [26] Ruslan Salakhutdinov and Andriy Mnih. 2008. Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In *ICML 2008*. 880–887.
- [27] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Riedl. 2000. Application of Dimensionality Reduction in Recommender Systems. In *ACM WebKDD Workshop*.
- [28] Mohamed Sarwat, James Avery, and Mohamed F. Mokbel. 2013. A RecDB in Action: Recommendation Made Easy in Relational Databases. *PVLDB* 6, 12 (2013), 1242–1245.
- [29] Jianping Shi, Naiyan Wang, Yang Xia, Dit-Yan Yeung, Irwin King, and Jiaya Jia. 2013. SCMF: Sparse Covariance Matrix Factorization for Collaborative Filtering. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13)*. 2705–2711.