Recommendations Made Easy *

[Extended Abstract] †

Darren Guinness
Baylor University
One Bear Place 97356
Waco, TX 76798
{darren_guinness

Paniz Karbasi
Baylor University
One Bear Place 97356
Waco, TX 76798
paniz_Karbasi

Rovshen Nazarov
Baylor University
One Bear Place 97356
Waco, TX 76798
rovshen_nazarov

Greg Speegle
Baylor University
One Bear Place 97356
Waco, TX 76798
Greg_Speegle}@baylor.edu

ABSTRACT

Fueled by ever-growing data, the need to provide recommendations for consumers, and the considerable domain knowledge required to implement distributed large scale graph solutions we sought to provide recommendations for users with minimal required knowledge. For this reason in this paper we implement a generalizable 'API-like' access to collaborative filtering. Three algorithms are introduced with three execution plans in order to accomplish the collaborative filtering functionality. Execution is based on memory constraints for scalability and our initial tests show promising results. We believe this method of large-scale generalized 'API-like' graph computation provides not only good tradeoff between performance and required knowledge, but also the future of distributed graph computation.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Distributed Databases;Distributed Networks;Distributed Applications; D.2 [**Software Engineering**]: Data Abstraction; Information Hiding

General Terms

Collaborative Filtering, Distributed Computation, API,

1. INTRODUCTION

Making recommendations of products like music, movies, toys, and other products is a common operation for businesses today. For businesses like Netflix, and Amazon these recommendations are a critical part of their services. Collaborative Filtering is a technique used to provide these recommendations based on users previous history [7]. However this process of making recommendations equates to a graph problem. Because of this fact the seemingly small scale operation of providing product recommendations to users quickly turns into a "Big Data" problem of finding recommendations within large graphs. This gives rise to the need for large scale solutions to Collaborative Filtering.

In the 'Big Data' era, easily writing applications which process huge sets of data in a quick, reliable manner is of great importance. Many 'Big Data' problems including Collaborative Filtering require simple graph operations to be done across a data set that is too large to fit into current memory constraints. In previous years developers would require "Super Computers" to perform large scale graph processing, which was too costly for most researchers, and businesses [10]. Because of the need for larger data queries, businesses have been constructing and maintaining data centers for large scale requests, and processing. These data centers combine readily available computers into large systems that can operate in parallel to perform much of what previously was performed in "Super Computers" [10].

These new data centers created a need for software to take advantage of their large scale distributed architecture. Because of this frameworks like PEGASUS,Stratosphere, Hama, Giraph, Graphlab, and Hadoop were developed [2], [5]. Each of these frameworks offer up large scale distributed processing on data, or graphs. However these tools are very new and documentation tends to be poor, offering a challenge to businesses trying to provide recommendations. Conversely operations on these huge graphs such as Collaborative Filtering are becoming more and more common and requires many developers that smaller scale businesses cannot afford. With these constraints in mind we see the need for collaborative filtering toolkits that require minimal programming domain knowledge.

*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM_PROC_ARTICLE-SP.CLS. Supported by ACM.

†A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2ε and BibTeX* at `www.acm.org/eaddress.htm`

In this paper we attempt to provide this scalable 'API-like' collaborative filtering using industry standard toolkits. We have selected from the previously mentioned frameworks Hadoop for it's advantages in large scale data processing [13], and GraphLab for it's ability to perform fast graph operations [2]. We then combine these large scale processing tools to provide a method of large scale collaborative filtering and offer access to recommendations with minimal necessary knowledge to remove much of the need for learning these difficult algorithms and tools.

2. RELATED WORK

In this section we introduce the tools leveraged in our project specifically Hadoop and GraphLab, as well as summarize related work in the applications of these tools. We then attempt to show concepts that our project uses out of each area.

2.1 Hadoop

Hadoop MapReduce is a software framework that allows for easily writing applications which process vast amounts of data in-parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner. Hadoop uses a new type of file system, known as Hadoop Distributed File System (HDFS) which distributes data across multiple machines that allows for file system access. Hadoop is not replacement for conventional Relational Database Management System (RDBMS), but a supplement to it. It is mainly designed to process extremely large data sets in batches. Hadoop's final dataset results are expected to be read many times, but written only once. Hadoop is designed to work well on unstructured or semi-structured data, which is not very suitable for RDBMS [6].

In the paper "Parallel Data Processing with MapReduce" [6] the author discusses advantages of the Hadoop MapReduce functionality. He lists immense data processing, simplicity and ease of use highlighting the facts that developers are only required to write are map and reduce tasks, flexibility for input type which can be either irregular or unstructured data, storage independence meaning that it can use with different storage tables, fault tolerance due to replications, and high scalability as more nodes can easily be added as needed.

One interesting paper similar in spirit to the project we are working on was by Cornell University researchers [12]. They too have discussed the possibility of combining bulk data processing powers of Hadoop and the in memory iterative processing of data in GraphLab. The interest in such combination was motivated by GraphLab's capability to capture interesting data dependencies. The problem that this paper addressed related to processing huge graphs, which are harder to process as they cannot fit in memory.

2.2 GraphLab

As mentioned earlier, one of the major tools in our project is GraphLab. In 2010 GraphLab, a parallel framework machine learning was developed by Yucheng Low et al. [9]. GraphLab targets improving upon an abstraction like MapReduce by expressing iterative algorithms with sparse computational dependencies and achieving a high degree of parallel performance. The paper also demonstrated the expressiveness of the GraphLab framework by designing and implementing parallel versions of belief propagation, Gibbs sampling, Co-EM, Lasso and Compressed Sensing [9]. They showed that using GraphLab, a good parallel performance is achieved on large-scale real-world problems using a 16-core computer with 4 AMD opteron 8384 processors and 64 GB of RAM. In their experiments, they developed an optimized shared memory implementation which is now known as GraphLab [9].

GraphLab, supports the representation of structured data dependencies, iterative computation, and flexible scheduling. The GraphLab abstraction uses a data graph to encode the computational structure and data dependencies of the problem. It also represents local computation in the form of update functions which transform the data on the graph. Since it is possible that update functions modify overlapping states, the GraphLab framework provides a set of data consistency models which enables the user to specify the minimal consistency requirements of their application without having to build their own complex locking protocols. To manage sharing and aggregation of global state, GraphLab provides a powerful sync mechanism. Also, for managing the scheduling of dynamic iterative parallel computation, GraphLab provides a collection of of parallel schedulers encompassing a wide range of ML algorithms [9].

2.3 Applications

In this section, we describe some of the most important applications of large-scale graph applications in different areas.

2.3.1 Big graph mining: algorithms and discoveries

[4] founded PEGASUS, one of the first major frameworks built on top of the MapReduce platform. PEGASUS used Hadoop's MapReduce to create and implement large scale graph mining algorithms that linearly scale on the number of machines [4]. This work is important to our research because it demonstrates that MapReduce can be leveraged for large scale graph operations, this is necessary for our pure Hadoop approach described later.

2.3.2 Empirical Analysis of Predictive Algorithms for Collaborative Filtering

In this paper, Breese, Heckerman, and Kadie examine common collaborative filtering techniques and algorithms including Correlation, Vector Similarity, Bayesian network, and Bayesian Clustering [1]. The paper also describes "Default Voting" a method utilized in our collaborative filtering implementation to rank common items between user sets when only one user has reviewed the item. Each algorithm was tested on the MS Web, Neilson Ratings, and EachMovie standard data sets for accuracy in collaborative filtering. Correlation and Bayesian Networks performed the best and won 10 out of the 16 cases [1]. For this reason we chose a naive version of Correlation collaborative filtering with default voting for our recommendation system.

2.3.3 An Evaluation Study of BigData Frameworks for Graph Processing

[2] provides the first evaluation study of the newest big data frameworks of Stratosphere, Hama, Giraph, Graphlab, compared to the traditional Map-Reduce paradigm. The study is conducted to test these frameworks against each other with the task of finding a k-core decomposition of a large graph using Amazon's EC2 service, a common industry option for big data processing [2]. This study showed some interesting results. First Hadoop proved to be inferior to the new graph based frameworks, and second GraphLab was the fastest of all evaluated frameworks [2]. For this reason our project one of our execution plans uses Hadoop's Map-Reduce only for pre-processing and graph creation and GraphLab for fast graph operations.

3. ALGORITHMS

We constructed three algorithms in order to perform Collaborative Filtering. Input to these algorithms was the QueryUser who we were trying to find recommendations for, the number of recommendations requested, and the raw review data. The first algorithm was a pre-processing step that extracted unique key-value pairs from Amazon customer review data. The algorithm would output two key-value pair files the first being product centered meaning that the keys consisted of products with corresponding degree 1 connected users as values. The second file was user centered meaning that the keys were users with degree 1 connected products as values. The second algorithm used these files to generate a bipartite graph of users with degree one connections to the QueryUser. The third algorithm used the Bipartite graph generated to find recommendations based on similar users which were degree 2 connections in the bipartite graph. These similar users were then sorted based on the Jaccard Index between the similar user and the QueryUser. After the sorting the set difference was calculated between the top most similar users and the QueryUser the results were then recommended to the QueryUser until the number of recommendations had been satisfied, or we had exhausted all similar users's products. The algorithms and their execution plans are described in more detail below.

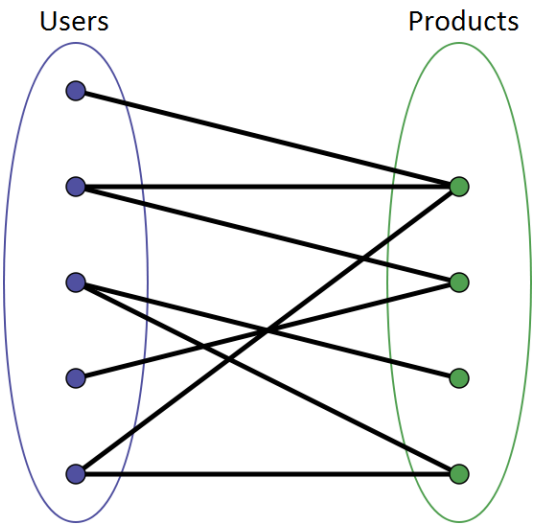

Figure 1: Bipartite Graph of Users and Products modified from the Public Domain

3.1 Pre-Processing

The first is a pre-processing algorithm designed to take in Amazon customer reviews and extract the items required for the computation. The Amazon customer review data consisted of 10 lines of data per review. The data lines were productID, productTitle, price, userID, profileName, helpfulness, rating, time, summary, and text. From this we needed productID, userID, and rating so we created a pre-processing algorithm using Hadoop MapReduce to extract this data. Hadoop's Map Phase was used to create key value pairs where there were non-unique keys. The Reduce phase took keys that were non-unique and merged them until we had a key-value pairs. There were two key-value pair files created in this step. The first had products as keys, and users as values. This product centered graph is shown in figure 2 The second had users as keys, and products as values, an example of this user centered graph is shown in figure 3. The resulting data was then ready to be input into the Bipartite Graph constructor shown in algorithm 2.

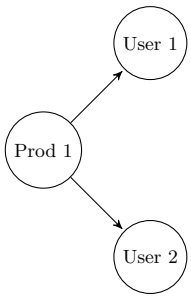

Figure 2: Product Centered graph

3.2 Bipartite Graph Generation

The second algorithm depicted in 2 generated the Bipartite graph. The input was the QueryUser, and the two key-value pair files(Product centered, and user centered files)

Data: Product Review raw data
Result: prodCentered, userCentered
while *not reviewRaw.end* **do**
  userIdToIntMap=<uID, uniqueIntUId>
  productIdToIntMap=<pID, uniqueIntPId>
end
foreach *user and product <user id, product id, rating>* **do**
  uniqueIntUId=userIdToIntMap.find(user)
  uniqueIntPId=productIdToIntMap.find(product)
  write to cleanedReviewFile: <uniqueIntPId, uniqueIntUId, rating>
end
while *not cleanedReviewFile.end* **do**
  MapPhase Get <Product ID, User ID>
  conetxt.write(< Product ID , User ID >)
  ShuffleSort < Product ID , {User} >
  ReducePhase result.write(< Product ID , {User ID} >)
end
while *not cleanedReviewFile.end* **do**
  MapPhase Get <Product ID, User ID>
  conetxt.write(< User ID , Product ID >)
  ShuffleSort < User ID , {Product ID} >
  ReducePhase result.write(< User ID , {Product ID} >)
end

Algorithm 1: Preprocessing Algorithm

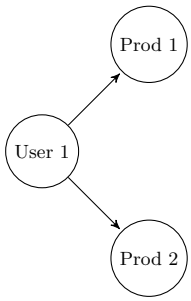

Figure 3: User centered graph

generated in the previous step. The algorithm searched the user centered file for the QueryUser, obtained the products he/she had purchased and then proceeded to search the product centered file for the users associated with these products. The algorithm's output was the generated the Bipartite graph and a file of similar users.

3.3 Collaborative Filtering

The final algorithm was the collaborative filtering algorithm. The input to this algorithm was the Bipartite Graph and the similar user file generated in the previous step, and the number of recommendations requested. The algorithm would then calculate the Jaccard index on each similar user and sort descending the users based on the index obtained. The algorithm would then perform a set difference on each similar user and recommend the resulting items to the QueryUser. This was repeated until the number of recommendations was met or the set of similar users was exhausted. Code for this algorithm is shown in algorithm 3.

3.4 Implementation and Communication

Data: QueryUser,Product Centered file, User Centered file
Result: Bipartite Graph, SimilarUser file
while *notprodCentered.end* **do**
  MapPhase - Get <Product ID, User ID>
  foreach *currUserID in {UserID}* **do**
    if $currUserID == queryUserID$ **then**
      conetxt.write(< queryUserID, { Product ID : {User ID}} >)
    else
    end
  end
  ShuffleSort < Product ID , {User} >
  ReducePhase result.write(< queryUserID {Product ID , {User ID};} >)
end
compute {relatedUserID} based on Bipartite Graph
while *not userCentered.end* **do**
  MapPhase - Get <User ID, Product ID>
  foreach *curUserID in {relatedUserID}* **do**
    if $curUserID == UserID$ **then**
      multiFile.write(fileName=UserID, < UserID, { Product ID} >)
    else
    end
  end
end

Algorithm 2: Bipartite Graph Constructor

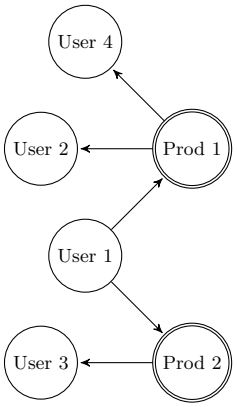

Figure 4: Generated Bipartite Graph

The algorithms were implemented on a 64-bit Linux cluster running kernel 2.6.32. We limited the number of CPU's to 2 and had 4 gigabytes of available memory. Hadoop 2.2 and GraphLab 2.2 were used. The installation of both of these tools was challenging process. Documentation was poor for both with missing critical items such as navigating to the debug directory before the makes in GraphLab. Documentation seemed to also use deprecated functionality showing that the updates to the code base were ahead of the documentation. This created some challenges for us when attempting to learn the tools and only confirmed our belief that the 'API-like' large data solutions like the one proposed in this paper were necessary. We installed the Hadoop MR and Graphlab distributed highly parallelized toolkits on the single machine cluster with the specifications defined above. This is somewhat limited our testing and performance anal-

Data: Bipartite Graph (Fig. 1), similarUsers, numRecommendations
Result: Recommended products
foreach *user,* $u \in similarUsers$ **do**
  compute $similarityRating[u] \equiv \frac{|q \cap g|}{|q \cup g|}$
end
sort similarityRatingMap <user ID, rating> by rating in desc order
foreach $relatedUser \in similarUsers$ **do**
  $queryUser - queryUser \cap relatedUser$
end
while $i < numRecommendations$ **do**
  foreach $user \in similarityRatingMap$ **do**
    foreach *product* $\in$ *the set diff of similar user* **do**
      if $i \geq numReccomendations$ **then**
        Do not add more products
      else
        $recommendedProducts.add(p)$
        $i++$
      end
    end
  end
end

Algorithm 3: Collaborative Filtering Algorithm

Data: similarUsersList, queryUser
Result: similarityRatingMap <user ID, rating>
foreach $similarUser \in similarUsersList$ **do**
  unionSet $\leftarrow queryUser \cup similarUser$
end
foreach $similarUser \in similarUsersList$ **do**
  intersectionSet $\leftarrow queryUser \cap similarUser$
end
foreach $similarUserID \in similarUsersList$ **do**
  jaccardIndex $\equiv \frac{|intersectionSet|}{|unionSet|}$
  simRatingMap.put(similarUserID, jaccardIndex)
end

Algorithm 4: Jaccard Index Algorithm

ysis capabilities as those tools are designed to be run in the cluster environment with more than single node. However, we believe based on other researchers experiments that our solution will perform as good or better in the distributed environment with multiple nodes for data processing.

After the installation the implementation of the hadoop part of the query processing was written in JAVA. Several jobs were written for different steps of the algorithms described above for data pre-processing and for query processing. GraphLab was implemented in C++. For the sake of generalization we sought to make future changes easy using reusable code. For Graphlab part of our model rather than writing a fully optimized collaborative filtering algorithm, we have extended GraphLab functionality by creating Set Union, Set Difference, and Set Intersection distributed toolkits which were all used when computing the Jaccard index in the query processing algorithm 3. As the GraphLab toolkit operated on integers only and our data used strings for both user and product ID's, a basic mapping table was built to store the integer keys and the string id they map to for both user and product ID's. Keeping the mapping was necessary to obtain the original raw data item back after the recommendation was made. The mapping algorithm was straight forward, where we just map a unique string id to its index position in the set. Thus we had two mapping tables for user id set and for product id set. Both of the mappings were available in HDFS for later conversion from integer id to string id.

To create our collaborative filtering algorithm we developed several bash scripts that utilize widely available and enabled by default script based linux languages, such as awk and bash. Those scripts were used to process data on HDFS, to move files, prepare intermediate results and print final results. Several scripts were made for each execution plan Hadoop only, Hadoop + GraphLab, and GraphLab only. The decision to run any of the 3 implementations was made based on memory constraints further described below. All scripts were designed to log critical sections for later result analysis. Some automation test scripts for one fold validation were written for Hadoop MR, MR only and Graphlab only algorithms to allow easy testing.

3.5 Execution Plans

We used three execution plans for our collaborative filtering query. Each execution plan was based on memory constraints in order to allow for scaling, and when applicable optimize runtime. The Pure Hadoop (3.1) execution plan is for vast datasets too large to fit into memory and thus uses only Hadoop. The Hadoop + GraphLab (3.2) execution plan used the MapReduce functionality in Hadoop to reduce the data so that it would fit into memory for GraphLab operations. Lastly the Pure GraphLab (3.3) execution plan was executed when the data given is a size that can already be fit into memory. As GraphLab already had a Collaborative Filtering toolkit we used this directly rather than implement our own version. Each approach is described in more detail below.

3.5.1 Pure Hadoop

In this approach the algorithm is executed solely in Hadoop. The input consisting of *QueryUser* or the user we are finding recommendations for, *numRecommendations* being the number of recommendations we would like to obtain, and review data is first fed into Hadoop where we use the MapReduce functionality to extract 2 maps from our review data via the preprocessing algorithm. The first map is "product centered" consisting of products as our keys, and their immediate connected users as the values. The second map is "user centered" with users as our keys, and a list of products as their values. We next use these maps to construct a bipartite graph with each product connected only to users, and each user connected only to products. Then we begin the collaborative filtering algorithm by finding all first degree similar users to *QueryUser* via examining each product that our *QueryUser* has reviewed and making a list of users who have also reviewed that product. Given this list of similar users, we then use a well known similarity index The Jaccard Index [3], a well known function for finding similar sets to obtain the most similar user to the *QueryUser* in our list of similar users. The Jaccard index ranges from 0 to 1 with 0 being the non similar and 1 being completely similar [3]. Once we have computed the similarity index for each user we find the largest indices that are not 1 meaning a user that is most similar but still has items that were not

within *QueryUser's* set to recommend. We then add each of these items to our list of items to recommend until we have reached *numRecommendations* or there is no more products to recommend. We then output the recommendations for the user.

3.5.2 Hadoop + GraphLab

In this approach we utilize both Hadoop and GraphLab to tackle our collaborative filtering problem. This approach is meant to utilize the power in both tools meaning Hadoop's highly distributed large scale data processing, and GraphLab's in memory data processing. Initially, Hadoop is once again used to implement the pre-processing algorithm to construct the two maps, and the bipartite graph from our review data in the same fashion as the above. In the Collaborative Filtering algorithm Hadoop then provides the similar userList to GraphLab. We then use developed for GraphLab vertex intersection and union toolkits to process each similar user graph with query user graph. Then we perform the Jaccard Index of the *QueryUser's* graph and each user's graph in the userList. Again we find the largest similarity indices that are not 1, and append non-shared items from the user to our *recommendedProducts* list and once we have reached our number of Recommendations required. Then we output our recommendations for the user.

3.5.3 Pure GraphLab

In the pure GraphLab approach our review data is small enough to fit into memory to begin with. As GraphLab already had a Collaborative Filtering toolkit we chose to use the toolkit in this approach. We utilized the Alternating Least Squares (ALS) method because of it's ability to provide quick recommendations on relatively large scale data sets. This method is used in the Results section as a baseline comparison for our two custom execution plans.

3.6 GraphLab Toolkits

In Algorithm 3, in order to compute *similarityRating* and $setDifference$, we wrote three GraphLab toolkits - union, intersection, and setDifference.

The intersection toolkit, took two graphs as the input, and calculated the intersection of the graphs into the HDFS. We used the graph_vertex_join class which provides the ability to pass information between vertices of two different graphs.

For the union toolkit, we used the latest feature of GraphLab, the Warp system. The basic design of the Warp system lies in use of fine-grained user-mode threading to hide communication latency of blocking calls; and as such expose a more intuitive and easy to use API Interface. One of the key functions of the Warp engine is a parfor over all vertices, which enables excuting a single function on all vertices. We used this feature in our union toolkit. This toolkit, took two graphs as the input and after finding the union set of the two graphs, once finished the result was written to the HDFS.

The setDifference toolkit, used both the features mentioned above - graph_vertex_join class and the Warp engine. This toolkit took two graphs as input, computed the intersection of the graphs using graph_vertex_join class and then with the help of Warp system, checked the vertices of the second graph, and appended those vertices not in the intersection set, to the set difference set. The result set was once again written to the HDFS.

3.7 Query Processing

In the algorithms described in this section, we used two major components: MapReduce and GraphLab. In order to utilize our 'API-like' framework, we needed an interface for calling our collaborative filtering algorithm. By using a simple Analytic Query Language (AQL) , we were able to effectively reduce the need for collaborative filtering and distributed tool specific knowledge.

The AQL engine inspired by concurrent research in ongoing in our department was extended to accommodate the collaborative filtering queries. The AQL algorithm has syntax close to other Query Languages such as SQL and is designed to hide the complex program and script structures from the user similar to that of SQL.

```
SELECT n recommendation
  FROM {g}
  WHERE userID= 'X,Y'
```

We used HDFS as shared storage between the MapReduce and GraphLab components. Several bash scripts were written to handle the integration between two toolkits. What those scripts do is simply run one tool, then use the output of the first tool as the input to another tool. In some cases result pre-processing or in memory storage was needed. For example the related users's ids are stored in memory while the bash script for related users's graphs is executing. Related users list was also used for accessing each related user's graph while computing Jaccard index and Set difference.

The bash script uses bash commands for generating arrays, copying files, checking file size and file existence. AWK commands were used for processing file per line to retrieve important file related data for later use in the bash processing. Both bash script and awk script were applied to potentially small files, e.g. related user graph, which size depends on the number of products that a user has reviewed.

We have bash script to map AQL command into bash script commands to start an algorithm chosen by the query processing logic. Query processing function chooses the execution algorithm based on the graph size using the logic explained below.

Based on the size of the input graph, one of the three approaches - Pure Hadoop MR, Hadoop MR + GraphLab, or Pure GraphLab was selected. To determine whether the graph would fit into memory (RAM available in the cluster) we used a hard-coded low level threshold value of 1 GB. If the Free memory minus the size of the graph was $\geq$ 1 GB the Graphlab only approach was used, if it was smaller we employed a GraphLab + Hadoop MR approach. If the free memory and graph size difference was below 0 and the graph size was bigger than graph size threshold (could be 20 GB), then Hadoop MR only approach with large enough (based

	MR only		GLMR		ALS Baseline	
Hold out	Index	Score	Index	Score	Index	Score
5915	8	.4	8	0.4	78	-1.784
2781	10	.4	10	0.4	17	1.176
9425	41	.28714	41	0.287	65	-0.952
11071	19	.28714	19	0.287	47	-0.370

Table 1: Cross Validation over one user

on the number of machines) cluster will be used. The hard coded 1 GB however was not an optimal solution, rather the proper strategy would be to compute the following:

$$FreeMem - Mem_g - Mem_{threshold} > 0$$

where $FreeMem$ is the memory available in the cluster system, Mem_g is the size of the graph g, and $Mem_{threshold}$ is a given threshold to allow the OS to perform computation.

4. RESULTS

4.1 Datasets

The primary source of data for our project was Amazon, from which we obtained about 34 million reviews. To extract the reviews data that we needed for experiments, we started with bigger dataset obtained from the Internet Archive footnotehttp://snap.stanford.edu/data/web-Amazon-links.html.

To test the performance of both of our algorithms we conducted two tests. The first was a Leave-One-Out Cross Validation on one user in order to gauge whether the algorithms would recommend the product back to the user. There were 4 rounds where in each round one product was held out and recommendations were made using the resulting subgraph. Table 1 illustrates the results of this test. The second test was designed to assess the runtime of the algorithms. The test measured the runtime of a Leave-One-Out Cross Validation over 3 users. Results of this test are shown in Table 2. We also used another collaborative Filtering technique for comparison known as Alternating Least Squares(ALS). The algorithm won the Netflix contest providing recommendations in 2008 [14]. Our algorithm used the Jaccard index so our scoring ranged from $0 \leq x \leq 1$, where 0 indicates no similarity and 1 indicates maximal similarity. The ALS algorithm uses $-\infty \leq x \leq \infty$, where a negative value indicates non-similarity and a positive value indicate similarity. The number's distance from 0 indicates the degree to which the user would prefer the item. As the two algorithms used different scoring ranges we used recommendation indicies to make a comparison.

4.2 Cross Validation

As each product was purchased by the user and had a rating above 4 as previously noted in the pre-processing stage we expect to be recommended these items when we use Leave-One-Out Cross Validation. Table 1 shows that our algorithms had higher ranked recommendations for each of the held out items than ALS algorithm. Interestingly enough the ALS algorithm had negative scores for most of the items that were held out, indicating a negative suggestion. A potential reason for this could be that the ALS method which compares both users and products does not find a correlation between the users's history. This may be because of the small amount of historical data that was used during the run.

4.3 Processing Time

User ID	Pure Hadoop	HD + GL	ALS	#Products	Related Users
45	467	168	8	5	59
1696	528	188	6	5	67
8509	135	41	6	7	12

Table 2: Cross Validation Duration measured in minutes

As we can see in table 2, the GraphLab only approach is considerably faster in processing time. GraphLab's computation operates primarily in memory [8] and the Hadoop MapReduce algorithm stores intermediate results on HDFS and MR on each iteration preforms expensive set up and tear down. Thus the Graphlab's approach is much faster, if the graph can feet in the memory. For small graphs a GraphLab only approach would be optimal. However this represents a comparison of a parallel in-memory solution versus a distributed big-data processing tool. An in-memory approach is expected to outperform an approach that stores intermediate results in the HDFS and performs computation in parallel in the distributed manner with very little shared data. The distributed nature of MapReduce allows processing big data in parts. The number of parts define how many machines are needed for faster data processing. On the other hand big datasets (in Terabytes or Petabytes) cannot reasonably fit into memory even when using distributed hardware with shared memory as required by Graphlab. For this reason GraphLab will fail when attempting to operate on large graphs that are unable to fit into memory. We saw this in several test cases, where the Hadoop + GraphLab approach was able to run and provide recommendations the GraphLab only approach failed due to memory constraints.

User ID	Pure HD	HD + GL	ALS	# Prod	# of Rel Users
45	99 min	39 min	51 sec	5	59
1696	112 min	51 min	52 sec	5	67
8509	22 min	10 min	52 sec	7	12
Total	233 min	100 min	155 sec	17	138

Table 3: Single User ID based AQL query processing time

As we can see in table 3, the same AQL query:

```
SELECT 100 recommendation
  FROM {Bipartite User Product Graph}
  WHERE userID= '45,1696,8509'
```

The above query was run using three processing algorithms to make a comparison of the running time as shown in 3 table. As we can see from the table the GraphLab only approach is considerably faster in processing time. As GraphLab's

computation operates primarily in memory [8] and the Hadoop MapReduce algorithm processing input in parts on different machines with little shared memory. Moreover, Hadoop MR approach requires file operations in HDFS for either MR only or Graphlab + MR approach to store intermediate results. As we can see the MR only is twice as slow as Mr + Graphlab approach as MR requires more storing intermediate results on HDFS for each iteration, which requires more disk I/O. This results in longer processing per iteration of computing Jaccard index [3], while Graphlab performs in memory computations, which requires less expensive I/Os.

5. DISCUSSION AND FURTHER WORK

The performance of our algorithms versus the GraphLab ALS baseline condition showed adequate results. In terms of runtime the GraphLab ALS algorithm boasted good performance. This however is expected due to the in-memory approach and has been declared infeasible for large scale graph problems [11]. The Hadoop + GraphLab approach had poor performance in runtime. This is due large number of high cost I/O operations and MR per iteration set up and tear down cost. The Pure Hadoop approach performed the worst in terms of runtime. In the Pure Hadoop approach there were even more MR runs, which results in more I/O operations and thus poor performance. It should be noted that MR only or MR + GL algorithms would be able to process larger scale graphs which cannot fit into memory.

In terms of the recommendation performance our algorithms actually beat out the ALS baseline condition. This shows that our algorithms were not only extendable to graphs that are too large to fit into memory, but also that the recommendations when tested using Leave-One-Out cross validation had better estimation of the user's preferences. We expected this was due to the small amount of historical data which poses a problem to the ALS algorithm. Since our algorithm used the Jaccard Index and a Bipatite Graph small amount of training data was necessary to make predictions.

The algorithm is not heavily optimized and thus can be improved further. Further works will optimize graph generation and similar user collection which we noticed to be a large portion of the computation. Also we plan to implement other similarity indices other than the Jaccard index in an effort to provide a more generalized solution.

6. CONCLUSION

In this paper we have presented a new algorithm for collaborative filtering with two execution plans. The algorithm was meant to be a proof of concept that large-scale collaborative filtering can be done in a generalizable API-like manner. We accomplished this using AQL a new query language for distributed large-scale computation which mimicked SQL queries and hid the algorithm and it's execution plans from the user. In doing this we allow non-programmers to provide recommendations to their customers/users with little background knowledge. Overall the algorithms performed adequately but can be further improved in both runtime and recommendation generation. Future large scale graph operations should seek API-like interfaces in an effort to reduce the time and knowledge it takes to implement these solutions.

7. REFERENCES

[1] J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.

[2] B. Elser and A. Montresor. An evaluation study of bigdata frameworks for graph processing. In *Big Data, 2013 IEEE International Conference on*, pages 60–67. IEEE, 2013.

[3] P. Jaccard. *Nouvelles recherches sur la distribution florale*. 1908.

[4] U. Kang and C. Faloutsos. Big graph mining: Algorithms and discoveries. *ACM SIGKDD Explorations Newsletter*, 14(2):29–36, 2013.

[5] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[6] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD Record*, 40(4):11–20, 2012.

[7] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, Jan 2003.

[8] Y. Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, University of California, 2013.

[9] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[10] J. Rattner. Concurrent processing: A new direction in scientific computing. *Managing Requirements Knowledge, International Workshop on*, 0:157, 1985.

[11] A. Schwing, T. Hazan, M. Pollefeys, and R. Urtasun. Distributed message passing for large scale graphical models. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1833–1840. IEEE, 2011.

[12] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[13] T. White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[14] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.