- The recommendation engine runs on a distributed cluster, it takes the following three steps to calculate the predicted ratings: 1) The first step is similarity computation. It computes a partial similarity matrix on every compute node. 2) The second step is sorting. It sorts elements in every column of partial similarity matrix S_i, and prunes the elements that are less than the similarity threshold. The most similar items are selected based on a neighborhood size after sorting. 3) The final step is to compute the potential ratings of unrated items for every user on every compute node.
- The output file represents the prediction of users' potential ratings, which shows users' ids, items' ids and corresponding predicted rating points.

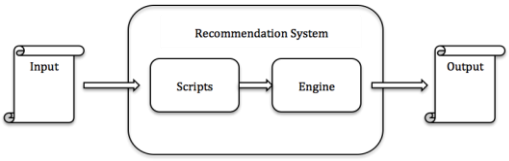

Figure 4. Recommendation system components.

The recommendation engine is implemented in C++ using MPI to realize parallelism on distributed memory cluster and Phoenix++ to execute in-memory MapReduce computation. Figure 5 shows the design of the recommendation engine. *SimMR* and *PtrMR* are two MapReduce classes in the engine, which are inherited from basic class in Phoenix++. Only *split()* and *map()* functions are provided in *SimMR* and *PtrMR*. The *split()* function divides input matrices into row strips and column strips, and assigns to every map task. The *map()* function implements inner product computation. The sort step prunes those dissimilar items as explained earlier.

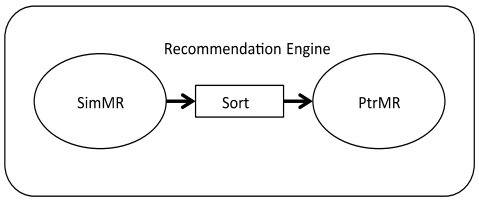

Figure 5. Recommendation engine design.

3.4. Potential issues

3.4.1. Storage efficiency. For the purpose of minimizing the I/O communication in NFS, we send the input matrix to every node, which makes the storage inefficient in the disk. Utilizing a parallel file system is a feasible solution to this problem in the future.

3.4.2. Load balance. Currently, data is divided into column panels evenly, so the scalability of our recommendation system is not close to optimum if ratings are not uniformly distributed in every column. Those nodes receiving popular rated items would perform more computation than others and become the bottleneck. A fine-grained data distribution method like 1-D column cyclic distribution is a feasible way to achieve load balance on in our future work. Figure 6 shows the data distribution differences when using four compute nodes. With 1-D column cyclic distribution, each compute node can receive nearly the same amount data for computation.

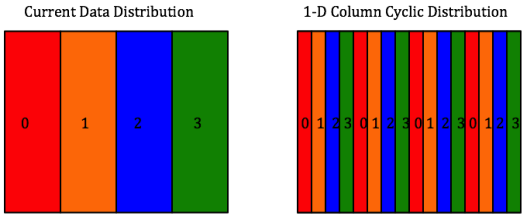

Figure 6. Data distribution optimization

4. Experimental results

In this section we conducted experiments on Amazon EC2 to measure the prediction quality and performance of our recommendation system (we call it dist-phoenix++). We also compared our system with the item-based CF algorithm in Apache Mahout. Both real-world datasets and randomly generated datasets were tested.

4.1. Apache Mahout

Mahout is an open source machine learning library from Apache. It is written in Java and primarily focused on recommendation engines, clustering, and classification. Its core algorithms are implemented on top of Hadoop so that it can scale on large datasets [9]. The item-based CF algorithm implemented in Mahout is compared with our dist-phoenix++.

4.2. Experiment setup

Our experiments were conducted on Amazon EC2. We used one *m1.xlarge* instance as one compute node, which is a 64-bit virtual machine with four virtual CPUs and 15 GB memory.

We built an eight-nodes cluster for dist-phoenix++. NFS was used as file system and MPI version was MPICH2.

For experiments with Mahout, we used the existing Amazon Elastic MapReduce (Amazon EMR) directly. The Hadoop version on Amazon

EMR is 1.0.3. We allocated eight *m1.xlarge* instances as compute nodes and one *m1.xlarge* instance as the master node. Amazon Simple Storage Service (Amazon S3) was the file system used in this Hadoop cluster.

4.3. Prediction quality

To test the quality of recommendation system, we used the MovieLens 100k dataset [13]. This dataset contains 100, 000 ratings by 943 users on 1682 movies. Both Mahout and our dist-phoenix++ system use Pearson Correlation as the similarity measurement, where the similarity threshold is 0.01. The prediction quality is measured by *Mean Absolute Error* (MAE):

$$MAE = \frac{\sum_{i=1}^{N} |p_i - q_i|}{N} \quad (6)$$

Here p_i is the predicted value and q_i is the real value. In Figure 7, 80% of the data was used as a training set and 20% of the data was used as a testing set. Total five different cases were tested. The default recommender in Mahout has the worst accuracy, because, by default, Mahout sets the similarity neighborhood size to 100 and only 10 items are considered for every user when doing prediction. In our dist-phoenix++ recommendation system, we set the similarity neighborhood size to the total number of items and all the items a user has rated will be considered during prediction. The MAE of dist-phoenix++ varies from 0.82 to 0.85. After tuning Mahout with the same setting as dist-phoenix++, its accuracy improves substantially to nearly 0.77.

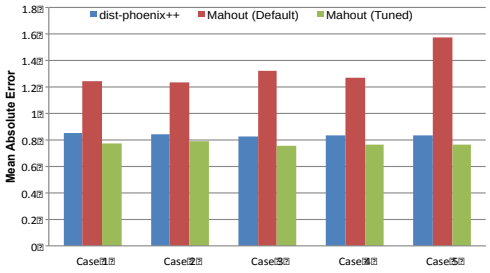

Figure 7. Prediction quality on MovieLens 100k dataset

In the following performance experiments, for both dist-phoenix++ and Mahout, we use 100 as the similarity neighborhood size and 0.01 as the similarity threshold.

4.4. Single compute node performance

We first compare the performance on a single compute node when the data size is small enough to fit into memory. All 100,000 ratings in MovieLens 100k dataset were used as input. The jester-3 dataset was also selected, which contains 24,938 users' ratings on 101 jokes [14]. In dist-phoenix++, all the data was loaded into memory only once and computed by Phoenix++. As shown in Figure 8, dist-phoenix++ outperforms Mahout greatly due to the advantage of in-memory computation.

Note that both Mahout tests ran for more than 10 minutes. This is because the launching overhead of one MapReduce job on Amazon EMR takes around 1 minute. And the item-based CF algorithm in Mahout contains 10 MapReduce jobs. This overhead can be reduced as the input data size increases.

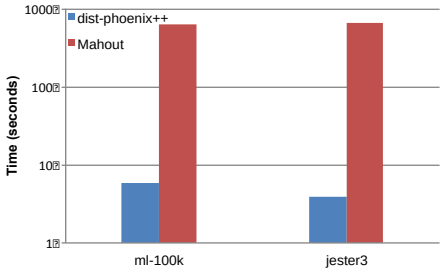

Figure 8. Performance on single compute node when data can fit into memory

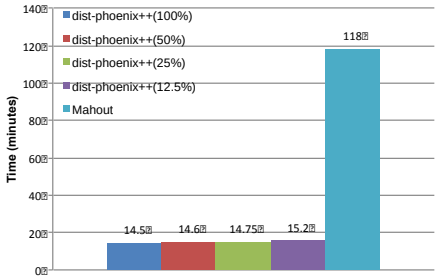

Figure 9. Performance on single compute node for MovieLens 10M dataset when block size changes

To investigate the impact of disk I/O on the overall performance, we varied the size of the data that can be loaded into memory. The dataset we used is MovieLens-10M, which contains 10,000,054 ratings applied to 10,681 movies and 71,567 users [13]. As shown in Figure 9, 100% means all the input data is loaded into memory once, 50% means each time 50% of input data is loaded into memory, and so forth for 25% and 12.5%. When disk I/O increases, the execution time of dist-phoenix++ increases slightly, and all of them are much faster than Mahout.

4.5. Multiple compute nodes performance

To investigate the multiple compute nodes performance of our recommendation system, we increased the number of compute nodes from 1 to 8.

Also, we simply copied MovieLens-10M dataset 10 times to get a 100M ratings dataset applied to 10, 681 movies and 715,670 users. The out-of-core block sizes *block_row* and *block_col* mentioned in section 3.1.3 were set to 2,000 and 1,000 in similarity computation, while 100,000 and 1,000 in prediction computation.

Figure 10 and Figure 11 show the running time and scalability of dist-phoenix++ versus Mahout. For 8 compute nodes, the running time of dist-phoenix++ is 1.07 hours, compared with 2.4 hours in Mahout, it gets a 2.25× speedup.

However, the scalability of dist-phoenix++ is not as good as Mahout. It has a 2.82× speedup when increasing the number of compute nodes from 1 to 8. We believe this is caused by the load balancing issue explained in Section 3.4. In the MovieLens dataset, popular rated movies are associated with a small number of items and are distributed to the first compute node. This makes the first node execute more computation than the other nodes and hence become the bottleneck.

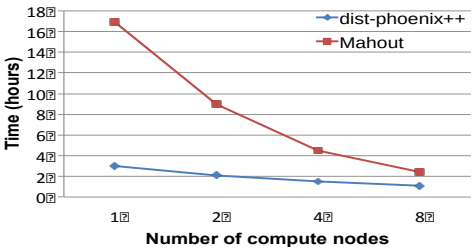

Figure 10. Running time of dist-phoenix++ and Mahout on 100M ratings dataset

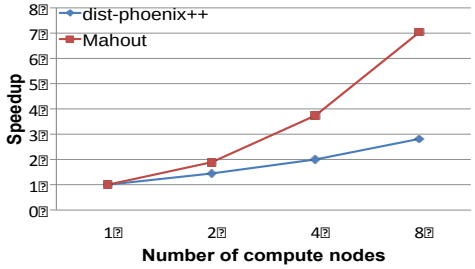

Figure 11. Speedup of dist-phoenix++ and Mahout on 100M ratings dataset

To further investigate the scalability of our dist-phoenix++, we created a uniformly distributed 50M ratings dataset. This dataset was randomly generated and contained 10,000 items and 500,000 users. Ratings were evenly distributed such that each item had 5,000 ratings. The out-of-core block sizes were the same as the previous test. Figure 12 shows that dist-phoenix++ has a linear speedup. With 8 compute nodes, dist-phoenix++ achieves a speedup of 7.44×.

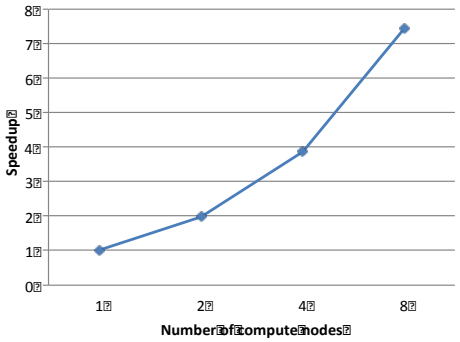

Figure 12. Speedup for a growing number of compute nodes for dist-phoenix++ on a 50M ratings uniformly distributed dataset

5. Related work

There are several shared-memory MapReduce libraries optimized for multicore architectures. Phoenix++ has demonstrated good performance comparable to hand-tuned parallel programs using pthreads [1]. Metis, from MIT, implements a new data structure (i.e., hash table and B+ tree) to group intermediate key/value pairs to provide high performance [15]. Tiled-MapReduce extends the MapReduce model with a tiling algorithm [16]. However, these libraries only work on shared-memory machines and assume the input must fit in the main memory. Our work extends the shared-memory library to support distributed-memory clusters.

Spark is a tool developed at the University of California at Berkeley to support in-memory cluster computing [17]. It provides a number of primitives to allow users to perform data mining efficiently. The primitives include not only map, reduce, but also union, join, filter, sample, and so on. Spark is often applied to iterative data mining applications. It works on distributed-memory clusters but is not designed to handle disk I/O for intermediate results automatically.

Twister is another widely used MapReduce runtime system that supports iterative MapReduce computations [18]. It first reads data from disks to local memories on distributed nodes, then starts to compute iteratively, and finally writes the output to disk. It is possible to extend Twister to store large intermediate results in local disks instead of buffering in memory. Unlike Spark and Twister, our approach is designed to scale up within a single node first, then scales out to many nodes.

There has also been several work on implementing a recommendation system in MapReduce: Schelter et al. presented a recommendation system based on a scalable similarity-based neighborhood method [7]. Jiang et al. presented a scaling-up Item-based CF algorithm using four MapReduce jobs [6]. However all of those

implementations used Hadoop, whereas our work implemented the recommendation system in MapReduce with Phoenix++ on distributed-memory clusters.

6. Conclusion

In this paper, we proposed a method to utilize shared- memory multicore MapReduce systems to implement a distributed high-performance recommendation system. It is inspired by the fact that Phoenix++ can be faster than Hadoop by 28.5 times on Amazon EC2 to run word count. To utilize the sub-module of Phoenix++, we extend the item-based collaborative filtering recommendation algorithm to support both distributed-memory clusters and out-of-core computations. The experimental results show that the recommendation quality of our new system is comparable to that of Apache Mahout, while the performance is faster than Mahout by up to 2.25 times on eight compute nodes. The reason we can achieve better performance is because the shared-memory MapReduce system is highly optimized to conduct in-memory computations and can reduce the number of I/O operations greatly.

7. Future work

We have proposed a new approach for utilizing Phoenix++ to build a high-performance recommendation system. The same approach can be used to build other types of big data applications. Our future work is to make the framework more general so that different application developers can use the framework as a library.

8. References

[1] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular mapreduce for shared-memory systems," in Proceedings of the second international workshop on MapReduce and its applications, 2011, pp. 9-16.

[2] J.Manyika, M.Chui, B.Brown, J.Bughin, R.Dobbs, C.Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," McKinsey Global Institute, June 2011.

[3] A. Rajaraman and J. D. Ullman, Mining of Massive Datasets. New York, NY, USA: Cambridge University Press, 2011.

[4] D. Almazro, G. Shahatah, L. Albdulkarim, M. Kherees, R. Martinez, and W. Nzoukou, "A survey paper on recommender systems," CoRR, vol. abs/1006.5278, 2010.

[5] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in Proceedings of the 10th international conference on World Wide Web, 2001, pp. 285-295.

[6] J. Jiang, J. Lu, G. Zhang, and G. Long, "Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop," in Proceedings of the 2011 IEEE World Congress on Services, 2011, pp. 490-497.

[7] S. Schelter, C. Boden, and V. Markl, "Scalable similarity-based neighborhood methods with mapreduce," in Proceedings of the sixth ACM conference on Recommender systems, 2012, pp. 163-170.

[8] Hadoop. The Apache Software Foundation. http://hadoop.apache.org/, (Access date: 2013-08-02).

[9] Apache Mahout. The Apache Software Foundation. http://mahout. apache.org/, (Access date: 2013-08-03).

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, 2004, pp. 137-150.

[11] T. White, Hadoop: The Definitive Guide, 1st ed. O'Reilly Media, Inc., 2009.

[12] T.Segaran, Programming collective intelligence,1st ed. O'Reilly, 2007.

[13] Movielens Data Sets. GroupLens Research. http://www.grouplens.org/node/73, (Access date: 2013-08-04).

[14] K. Goldberg. Jester Collaborative Filtering Dataset. http://goldberg.berkeley.edu/jester-data/, (Access date: 2013-08-04).

[15] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing mapreduce for multicore architectures," in Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep, 2010.

[16] R. Chen, H. Chen, and B. Zang, "Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling," in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 2010, pp. 523-534.

[17] Spark. UC Berkeley AMPLab. http://spark-project.org/, (Access date: 2013-08-05).

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 810-818.