# What's Next? Music Recommendation System

Stephanie Smith, Sarah Jones, Karl Bo Lopker

## I. Introduction

### A. Motivation/Challenge

Music is easy to listen to, but hard to find. With so many options it is often difficult to know what song to hear next. The main problem is that a good song for one person may be another's "nails on a chalkboard." Sites like Pitchfork and Metacritic provide general music ratings. While these ratings may be a good indicator of a song's quality, a user really wants to know if they will actually enjoy it.
The What's Next? Recommendation System is an application that will tell a user what song they should hear next. The recommendation is based on the user's music tastes, not some faceless critic's.

Recommendation systems are an active topic in research and industry. They are utilized in music, movies, shopping, and many other applications to help consumers find products/services that they are interested in. This provides benefits to both the consumer, who would like to reduce their search time for interesting and relevant products, and to the businesses that are trying to sell their product.
In order for a recommendation system to provide relevant and accurate recommendations a lot of data is necessary to train the system. Processing this much data is computationally expensive and time consuming as well as a challenging problem. The response time of query is an important factor for an acceptable user experience, and accessing and processing that large of a quantity of data can cause a longer than tolerable delay if not implemented well.

The main goal of this project is to predict what music a user would rate well (enjoy); the challenge is this requires processing of a very large data set in order for the prediction to be accurate. Because the data set is large one must consider the format—that is how the data is indexed—when considering the algorithm and implementation. Sequentially this is a very time consuming algorithm. In parallel, the challenge is how to efficiently partition the data for processing. While this is not a novel contribution, this allows us to explore the complexity of a recommendation as well as to interact with Big Data and to create a simple implementation of the recommendation systems that are very influential in today's Internet.

Our contribution is a Java implementation of neighborhood creation parallelized with Hadoop for a music recommendation system.

### B. Nearest Neighbor Algorithm

For this project we will be utilizing the Yahoo! KDD music rating data with a Neighborhood Model. By calculating the k Nearest Neighbors for all songs, we find the top k most similar songs for each song [2]. Given the neighborhoods that we calculate, we can recommend items that are similar to the music items that the user rated well by predicting the user's rating for an item.

First, in order to test the k-NN algorithm that we derived from [2] we will implement the neighborhood algorithm and the query algorithm sequentially. The algorithm calculates the adjusted cosine similarity between all pairs of songs:

$$w_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \overline{r}_u)(r_{uj} - \overline{r}_u)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \overline{r}_u)^2 \sum_{u \in U_{ij}} (r_{uj} - \overline{r}_u)^2}}.$$

*Figure 1. Adjusted Cosine Similarity between songs i and j*
*U is the set of users ($U_{ij}$ is the set of users that have rated both i and j). $r_{ui}$ is the rating for the item i, given by user u. bar($r_u$) is the average rating for a user u. [2]*

# What's Next? Music Recommendation System

Stephanie Smith, Sarah Jones, Karl Bo Lopker

This similarity is combined with the intersection of the songs that a given user has rated and each song's neighborhood in order to predict the ratings the user would give to all other songs. The songs with the highest predicted ratings should be recommended to the user.

$$\widehat{r}_{ui} = \frac{\sum_{j \in R_u \cap N_i} w_{ij} r_{uj}}{\sum_{j \in R_u \cap N_i} |w_{ij}|} .$$

*Figure 2. The predicted rating of item i for user u*
*$R_u$ is the set of all rated songs by user u. $N_i$ is the neighborhood of songs for song i. $w_{ij}$ is the similarity from Figure 1. [2]*

### C. Parallel

In order to scale with the large KDD data set we will use Hadoop in conjunction with MapReduce [3]. The KDD database can be too large for one machine to process. Hadoop allows us to partition the KDD database into logical partitions for parallel processing over several different machines. MapReduce handles the actual processing by abstracting away machine-to-machine communication.

### D. Related Work

K Nearest Neighbor is a common algorithm for classification. Several works have used kNN to implement recommendation systems, including a couple of the papers from the Yahoo! KDD 2011.

[1] designed "an informative ensemble learning framework, which augments model predictions by an additional set of meta features to represent the training instances for ensemble learning." When integrating neighborhood information into their model, they pre-computed the k-Nearest Neighbors using MapReduce.

[2] implemented an ensemble method which combines twelve different algorithms and models to create a better recommendation algorithm. The kNN algorithm is not known to work well by itself ([2] reported that it had the highest RMSE of the 12 algorithms they analyzed), but it complements other algorithms well [1].

To provide a scalable implementation of the kNN, [3] designed a cascading MapReduce algorithm. This approach breaks up the pieces of the similarity computation and thus requires less memory resources from a single machine; however, it requires linking of several MapReduce steps and thus requires a lot of I/O. Alternatively, [4] discusses how to partition the data into chunks and then compare (that is perform the necessary computations with) each chunk to each other chunk to reduce the I/O caused by having to do several map and reduce steps. This approach requires that two chunks be in memory at the same time.

## II. KNN

### A. Parallelizing KNN

In order to use Hadoop we have to instruct it how to partition the database. Ideally, we would split the songs up with each song containing a list of user ID and ratings. Unfortunately, the data is arranged inversely by users with song ID and ratings. To remedy this we will chunk the database into c parts. The ith chunk will contain all users, but only songs and ratings corresponding to the range [i*c, i*c+c-1]. This scheme is shown below in Figure 3.

Splitting the data up in this way allows us to apply the sequential kNN algorithm in parallel. Each machine will be responsible for one chunk of data. The chunking and the creation of the chunk specific neighborhoods are performed in the map phase of the MapReduce process. Following this step, the

shuffle between the map and reduce phases will collect all chunk-specific neighborhoods by each song id. In the reduce phase, each song will choose its actual neighborhood from the intersection of the chunk-specific neighborhoods (which is a superset of the actual neighborhood). At this point the KNN training step is complete and we can query the results.

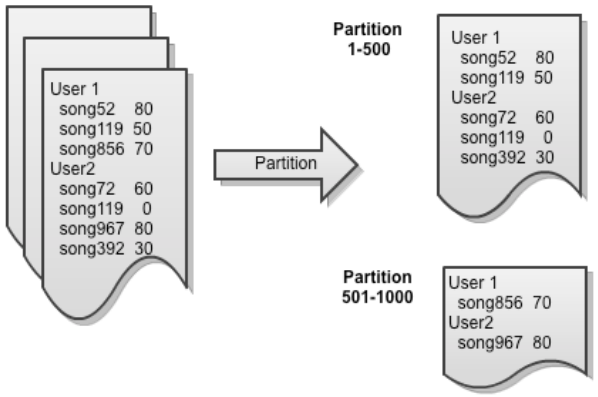


*Figure 3. Partitioning*
*Partition songs into sets of 500*

A query in this system will be made by inputting a user id. The recommendation system will return items that are similar to the music the user rated best.

### B. MapReduce

Section I.3 explains how we parallelize the neighborhood creation. However, the way that we implemented this in Hadoop MapReduce is not the normal usage of MapReduce, which generally reads one object (typically a line) at a time and produces zero, one, or more outputs to be shuffled and reduced. This algorithm requires that a song is compared to all other songs, so for this to be possible, at least one of the chunks must be in memory; this is the inner loop in our comparison. We found that the outer loop cannot be read in one line (one user or one song) at a time because of the innermost loop in our algorithm, which iterates across all users. All the user information is known about the chunk in memory, but the user information about the outer loop chunk has not all been read yet, so one does not know if the next user has also rated the current song. Thus, we found that the chunking approach allows us to divide the problem into pieces without creating an inverted index, which is very expensive, but it requires two chunks are in entirely in memory.

The query is a separate MapReduce job. For this, all of the chunks need to be read into memory because one needs to know all of the "active" user's ratings; one could load only those user's information from the unchunked dataset, but for simplicity we chose to use the same parser we used for the kNN chunk parsing. The neighborhood file (which we expect to be larger than the original dataset) is read line by line, in the traditional MapReduce style. The neighborhood file is what is split based on song and its neighborhood across chunks.

### C. Algorithm Accuracy

There are two parameters that affect the accuracy in our implementation: the neighborhood size and the rating count threshold. The neighborhood size is important to how much data we want to store, both in memory, and in our end result. Limiting the size of the neighborhood allows us to only keep track of items that are similar and thus likely to be rated well. The neighborhood and the set of rated songs for a user need to be large enough to produce several recommendation results (for each song no more than the

size of the neighborhood recommendations can be made and items already rated in this set cannot be recommended).

The rating count threshold is the minimum number of ratings a song must have in order for it to be added to another song's neighborhood; to our knowledge this piece was not implemented by previous work. This was a parameter we added on a very small data set because we found that if a song has only a few ratings the predicted similarity value was not accurate; the values were 1 or -1 (or very close to these values) in the similarity range of [1,-1] (most to least similar). Values that are at 1 are an issue because they will fill the neighborhood (which is made up of the songs that have a the highest similarity values) even though there is little data backing that these songs are actually similar. We found that requiring a certain number of ratings allowed the similarity values to be more realistic, thus allowing the neighborhoods to contain more similar songs, and thus we expect will produce better recommendations.

### D. Obstacles and Lessons Learned

Throughout the implementation process of the parallel portion, we found that working with MapReduce is much more complicated than we expected. We found that Hadoop was very sensitive to versions and that the packages used (for the same actions) changes completely depending on the version. Additionally, the documentation (other than the javadocs) is not very thorough. One piece that we tried to implement in MapReduce was making our objects "Writable", which meant they could be written out in our defined format and read in, and thus the parameters we received from Hadoop would be our objects instead of Text objects. However, both defining the writing and reading classes (multiple are required for each) and handling multi-line objects proved to be challenging especially since the few examples we found online were version dependent (but the version was rarely specified). In order to "make it work" we abandoned this customization and parsed the Text object in the map function in order to work around the Hadoop "magic." For similar reasons, we abandoned our MapReduce implementation of the recommendation query in favor of the sequential implementation due to lack of remaining time.

## III. Evaluation

There are two aspects to consider for our implementations of KNN: how accurately it predicts and what the speedup benefits of MapReduce are.

### A. Accuracy

We ran our implementations on the KDD Data Set 1, which consists of 1,000,990 users, 624,961 songs, 252,800,275 training ratings (5.6 GB), and 6,005,940 test ratings. There is also a set of 4,003,960 validation ratings that we are not utilizing. The test ratings set is a set of six songs and their correct rating values. This is not an ideal comparison set because the outputs of our program are the songs with the highest predicted ratings for a user.

In order to evaluate the accuracy of our implementation, we created a very small dataset to run the query sequentially. The predicted ratings were reasonable, and this gave us the confidence in our sequential implementation necessary to proceed and begin the MapReduce implementation.

Since we chose to focus on getting a working kNN MapReduce implementation and did not get the query MapReduce implementation to a runnable status, we cannot report on the accuracy given a large dataset.

### B. Performance

#### i. *Setup*

We used the Trition computing cluster to test our MapReduce algorithm. The nodes we used were the gB222X Appro blade nodes, which have two quad-core 2.4GHz processors. Although each node has

eight cores, we only used one core per node. This allowed us to reduce the artificial speedups caused by same node communication. A good MapReduce algorithm cannot rely on fast internode communication.

The software we used included; PBS, myHadoop, and Hadoop. PBS was used to request the amount of nodes we wanted. MyHadoop is used to manage Hadoop clusters on PBS controlled clusters like Trition.

### ii. *Method*

We ran one test for each node configuration. To start a test we first requested nodes from PBS. Once we got our nodes we ran a myHadoop script to format a HDFS, load our data, and start Hadoop on each node. The script then ran our Hadoop job and saved the logs locally. After our job was complete the HDFS was erased and Hadoop was shut down on each node.

A job is the calculation of one chunk, which contains 20,000 songs. Each of the songs in the chunk is compared to the ~200,000,000 songs in the database of Track 1 to find its similarity. During each test we only recorded the time of the job and not the setup or clean up time.

For each test we chose parameters to mimic real world usage. For the neighborhood size we used 100. This will generate 100 similarities for each song (the k parameter), which for one chunk is about 2,000,000 pairings. Next we filtered the results with a user rating count of 50 (parameter r). This discards any similarity that was calculated using less than 50 users.

### iii. *Results*

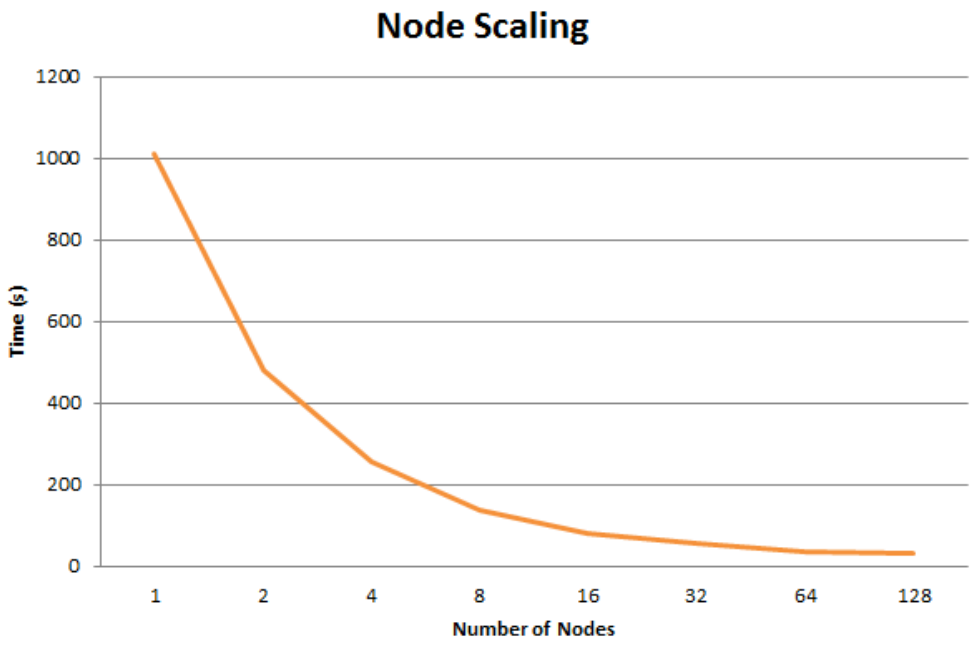


*Figure 4. Node Scaling (Time vs. Number of Nodes)*

Our results are promising. Figure 4 shows how our algorithm's time scales with the number of nodes. We can see dramatic increases in performance after only a couple nodes are added. However, we can see the speedup of our algorithm seems to decrease as we add more than eight nodes. To get a better look we graphed the efficiency of our approach.

# What's Next? Music Recommendation System

Stephanie Smith, Sarah Jones, Karl Bo Lopker

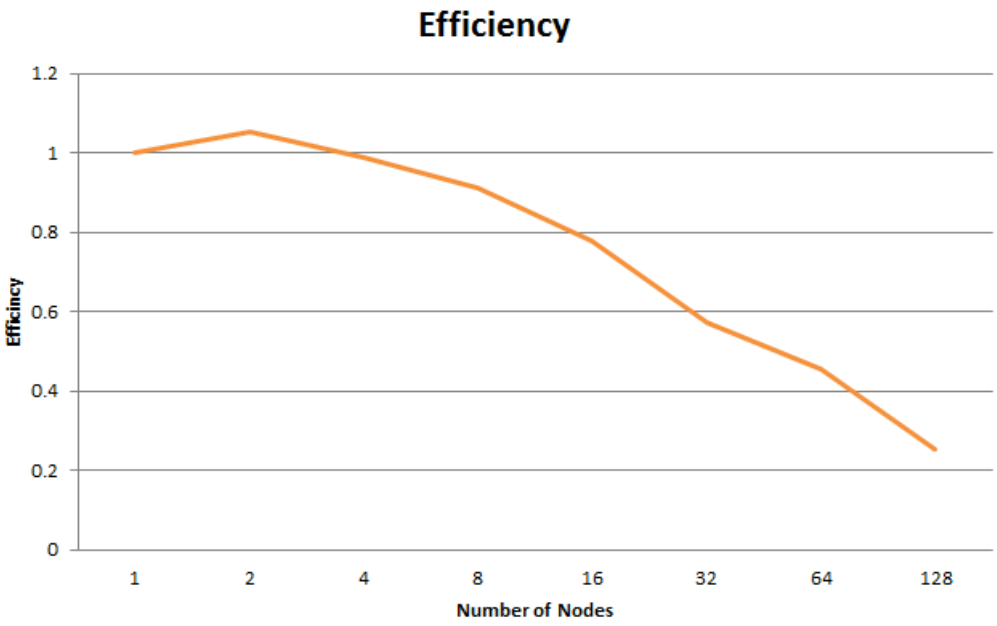


*Figure 5. Efficiency vs. Number of Nodes*

Figure 5 shows the parallel efficiency. If an algorithm were perfectly scalable this graph would show a straight horizontal line at 1. This is not the case with our algorithm for this testing environment. As we added more than eight nodes the efficiency decreases quickly. This is likely caused by two factors.

As we add more nodes each node has less local data. This will cause an increase of node communication, which may starve bandwidth. This is to be expected since bandwidth is the main scarce resource in any MapReduce implementation. However our algorithm relies heavily on every node having access to all the data so it will be penalized more by slow communication.

The second factor is that each node has less work to do as node count increases. Loading each node with enough work is important because of the MapReduce startup overhead. Best practices recommend around 100 tasks for each node, but our test job only had 200 tasks to distribute. This is a perfect amount for eight nodes, but way too little for 128. Increasing the amount of tasks by increasing the database size should increase efficiency.

We were unable to compare the time performance to any of the four KDD papers that run Track 1. None of these papers provided full execution times; we cannot compare speedup because this is algorithm and sequential implementation dependent.

## IV. Conclusion/Discussion

From this process we developed a runnable Java MapReduce implementation that can analyze more than 250 million ratings in half an hour. Although we are unable to compare our times with the Track1 papers because they do not provide full run times, we feel that this run time is fast enough to be considered a viable offline job.

We will not be continuing the work on this project, but future work would include running and analyzing the query for predicted ratings on MapReduce. Additionally, reducing the overhead of the memory required without significantly increasing IO time would be something to consider further; for example one could split the map processes based on both the chunks, like we do in our kNN MapReduce, and also on the neighborhood input.

# What's Next? Music Recommendation System

Stephanie Smith, Sarah Jones, Karl Bo Lopker

## V. Individual Contribution

**Karl Bo Lopker** - Researched how Hadoop and MapReduce worked, Configured Hadoop to run on test and Trition systems, Engineered harness for parallel and sequential parts in Java, Ran and analyzed Hadoop results, Results writing

**Sarah Jones** - Research on KNN algorithm, Sequential: object models, Neighborhood Parser, Neighborhood algorithm implementation, code refactorization, Predicted Rating algorithm implementation, testing, MapReduce: loading into memory, map and reduce methods, writing out and reading in neighborhood as customized multi-line (failed), MapReduce implementation of predicted rating (decided not to use because of debugging complexity), Sequential run, Writing

**Stephanie Smith** - Sequential: object models, Neighborhood Parser, Neighborhood algorithm implementation, code refactorization, Predicted Rating algorithm implementation, testing, MapReduce: loading into memory, map and reduce methods, reading in neighborhood as customized multi-line (failed), MapReduce implementation of predicted rating (decided not to use because of debugging complexity), debugging of MapReduce implementation, writing

## VI. Plan and Milestones

1) K-Nearest Neighbor - May 18th
Be able to calculate similar items and recommend items
2) Map-Reduce - ~~May 27th~~ June 6 (on test environment)
Translate step 1 into a Map-Reduce/Hadoop algorithm
3) ~~Improvements/~~Tuning - ~~June 5th~~ June 11
4) Paper - ~~June 10th~~ June 12
5) Presentation - ~~June 5th~~ June 13

## VII. References

[1] Chen, Tianqi, Zhao Zheng, Qiuxia Lu, Xiao Jiang, Yuqiang Chen, Weinang Zhan, Kailong Chen, Yong Yu, Nathan N. Liu, Bin Cao, Luheng He, and Qiang Yang. "Informative Ensemble of Multi-Resolution Dynamic Factorization Models."*KDD-Cup*. Yahoo! Labs, 2011. Web. 25 Apr. 2012. <http://kddcup.yahoo.com/pdf/Track1-LeBuShiShu-Paper.pdf>.

[2] Wu, Yao, Qiang Yan, Danny Bickson, Yucheng Low, and Qing Yang. "Efficient Multicore Collaborative Filtering." *KDD-Cup*. Yahoo! Labs, 2011. Web. 25 Apr. 2012. <http://kddcup.yahoo.com/pdf/Track1-LeBuShiShu-Paper.pdf>.

[3] Liang, Huizhi, Hogan, Jim, & Xu, Yue (2010) Parallel user profiling based on folksonomy for Large Scaled Recommender Systems : an implimentation of Cascading MapReduce. In Proceedings of 10th Industrial Conference on Data Mining, IEEE, Berlin. <http://eprints.qut.edu.au/41889/1/icdm_cloudcomputing_cameraReady.pdf>.

[4] Alabduljalil, Maha, Xun Tang, Tao Yang, and Alexandra Potapova. "Optimizing Parallel Algorithms for Similarity Search." (Not yet published) Email. 5 May 2012.

[5] Yahoo! "Datasets." *KDD Cup from Yahoo! Labs*. Yahoo!, 2011. Web. 17 May 2012. <http://kddcup.yahoo.com/datasets.php>.