International Journal of Engineering & Technology, 7 (3.8) (2018) 16-21

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Simplified Mapreduce Mechanism for Large Scale Data Processing

Md Tahsir Ahmed Munna[1]*, Shaikh Muhammad Allayear[1], Mirza Mohtashim Alam[1], Sheikh Shah Mohammad Motiur Rahman[2], Md Samadur Rahman[3], M. Mesbahuddin Sarker[3]

[1]*Dept. Of Multimedia And Creative Technology, Daffodil International University, Bangladesh*
[2]*Dept. Of Software Engineering, Daffodil International University, Bangladesh*
[3] *Institute Of Information Technology, Jahangirnagar University, Bangladesh*
**Corresponding Author E-Mail: Tahsir.Se@Gmail.Com*

Abstract

MapReduce has become a popular programming model for processing and running large-scale data sets with a parallel, distributed paradigm on a cluster. Hadoop MapReduce is needed especially for large scale data like big data processing. In this paper, we work to modify the Hadoop MapReduce Algorithm and implement it to reduce processing time.

Keywords: *MapReduce; Large Scale Data; Hadoop; Simplified Algorithm; Performance Analysis*

1. Introduction

The amount of data is increasing day by day. To process huge amounts of data, Google introduced MapReduce mechanism [1]. The MapReduce model has become popular because a programmer can harness the processing power of large data centers for very large parallel tasks in a simple way. The MapReduce programming model has been divided into two parts- Map function and Reduce function. The only thing the programmer needs to do is to write the logic of a Map function and reduce function. It is a big challenge how efficiently we can use MapReduce mechanism. In the MapReduce framework there are two executing parts: Map Task Execution and Reduce Task Execution. In MapReduce, for Map Task Execution, contents from a file are divided into several parts/splits depending on the number of Mappers present (Fig. 2). Generally, the size of each split is 64MB to 128MB. Each Split is assigned to each Mapper. Finally, mapping related all tasks start where each mapper generates an intermediate key/value pair for all the mappers particular contents. In practice all the mappers are separate machines and works in parallel process. After that, each Mapper is either individually reduced in advance or partially reduced; this is called the Combiner function. After completion of Map task, each of the contents from all the mappers are gathered parallel in one place. Subsequently, they are shuffled and sorted in the same place. After that Reducer performs the reducing task on all values of a particular type of keys. Sorting is required to shuffle the contents, so that same keys with values can be brought together while each Reducer can work on a particular type of key or value pairs only. Number of Reducers depends on number of different types of keys present. Again, all the Reducers work in parallel with each other. To process data faster and efficiently we modified the algorithmic logic that Google proposed for MapReduce framework [1]. Our modified MapReduce programming logic gives better performance against the original MapReduce algorithm.

The rest of this paper is organized as follows. In Section II we describe the background, research motivations and related works of MapReduce. The modified proposed model has discussed in detail in Section III. Comparison between modified model with original model has briefly discussed and debated in Section IV. Result and Performance analysis has described in Section V. Finally in Section VI conclude the study.

2. Motivation and Related Works

From the previous section we can see that Mappers have to wait to start their mapping tasks after all the splitting and assigning to Mappers are over, and Reducers have to wait to start their reducing tasks after all the Shuffling/Sorting is over. It would have taken much less time if Mappers and Reducers did not have to wait for their tasks to start. And so, in this paper our target is to somehow cut the waiting times for Mappers and Reducers to start their tasks and to provide a better algorithmic framework for MRA (MapReduce Agent). Considerable work has been devoted to processing joins on relational data. Blanas [3] compared the implementations of traditional join algorithms in MapReduce for Map function and Reduce function. Afrati and Ullman [4] provided specialized algorithms for multiway equijoins. Lin [5] tackled the same problem utilizing column-based storage. Okcan and Riedewald [6] devised algorithms for reporting the cartesian product of two tables. Zhang [7] discussed efficient processing of multiway theta-joins. Regarding joins on non-relational data, Vernica [8], Metwally and Faloutsos [9] studied set-similarity join. Afrati [10] re-visited this problem and its variants under the constraint that an algorithm must terminate in a single round. Lu [11], on the other hand, investigated k nearest neighbor join in Euclidean space. MapReduce has been proven useful for processing massive graphs. Suri, Vassilvitskii [12], and Tsourakakis [13] considered triangle counting, Morales [14] dealt with b-matching, Bahmani [15] focused on the discovery of densest subgraphs, Karloff [16] analyzed computing connected components and spanning trees, while Lattanzi [17] studied maximal matching, vertex/edge cover, and minimum cut. Data mining and2.1. The paper should have the following structure statistical analysis are also popular topics on MapReduce. Clustering was investigated by Das[19], Cordeiro [19], and Ene [120]. Classification and regression were studied by Panda [21]. Ghoting developed an integrated toolkit to facilitate machine learning tasks. Pansare [23] and Laptev explained how to compute aggregates over a gigantic file. Grover and Carey [25] focused on extracting a set of samples satisfying a given predicate. Chen [26] described techniques for supporting operations of data warehouses. Among the other algorithmic studies on MapReduce, Chierichetti [27] attacked approximation versions of the set cover problem. Wang [28] described algorithms for the simulation of real-world events. Bahmani [29] proposed methods for calculating personalized page ranks. Jestes [30] investigated the construction of wavelet histograms. There is only one paper, as far as we know, that they eliminated the Shuffle-Sort altogether regarding it as a barrier. It takes far lesser time than ours but is not convenient in situations where data is lost from the Reducer; we will see the reason later. The paper is by Abhishek, Nicolas, Brian, Indranil and Roy [31].

3. Proposed Simplified MapReduce

To eliminate the waiting time for the Mappers and Reducers to start their tasks and to make the algorithm work much faster than the original one, we approached a different method where we have proposed a total of three parts in the original MapReduce algorithm to create a new Simplified MapReduce algorithm (Fig. 1).

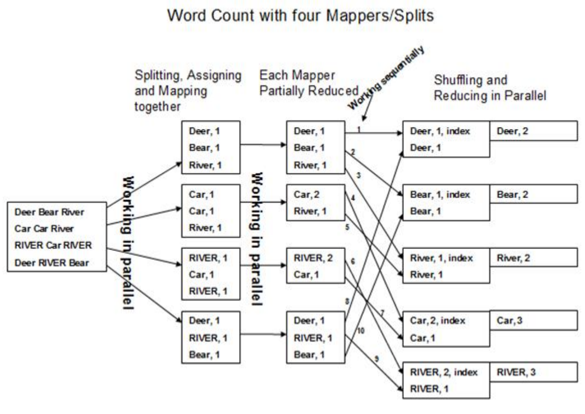

Fig. 1: Our Proposed Simplified Map Reduce Algorithm

3.1. Simplified Map Task Execution

In the Map Task Execution we have modified *first part* from the original MapReduce algorithm: **Parallel Splitting-Assigning with Mapping (first part):** Splitting, assigning to Mappers and Mapping are all done together in parallel, that is, while one splitting, as signing and Mapping are being started together for one Mapper, another Mapper also becomes responsible for splitting, assigning and mapping at the same time.

3.2. Simplified Reduce Task Execution

In Reduce Task Execution we have done our modified second and third parts from the original MapReduce algorithm:

Parallel Shuffling with Reducing (second part): After the Combiner function (from Map Task Execution), same keys are brought together (i.e. Shuffled) and Reduced at the same time, i.e. both Shuffling and Reducing are done in parallel without having to sort. Here, when shuffling for one type of key starts, reducing for that type of key starts at the same time. Although all the Reducers cannot work in parallel with each other like the original MapReduce algorithm but still this procedure takes far lesser time than the original one.
Minimizing Shuffle Time without Sorting (third part): We cannot eliminate the shuffle altogether because if any Reducer loses any information then Reducer does not have to recollect information all the way from Mappers, it can recollect it from the Shuffle part. But, we showed a way where shuffling can be done by cutting out the sorting part and minimizing the searching part, hence making the parallel Shuffling with Reducing work much faster than in Original MapReduce Algorithm.

3.3. Creating Word Count

In this section we show the detailed implementations with difference between an application coded for the Original MapReduce algorithm and the Refined MapReduce algorithm. For this, we use the Word Count application provided with the Hadoop distribution as MapReduce algorithms are usually tested with Word Count. In Word Count all types of words occurring once or more than once are gathered together in one place in a cluster and then the number of occurrences of each type of word is counted. Word Count is like the "Hello World" program in MapReduce, but it is not useless. Because counting words in particular strings is a critical part in lots of real world programs.

4. Comparative Description Between Simplified and Original MapReduce Mechanism

4.1. Word Count with Original MapReduce

4.1.1. Map Task Execution

The number of words presented in the text file is counted and taken into variable *i*. Then an input is taken from the user as variable split to know how many Splits/Mappers there should be, that is, how many parts the text file will be divided into. If split is more than *i* then an error message will be shown and will be asked to re-enter, also if split is equal to *0*, then error message will be shown saying that splits cannot be *0* and again asked to re-enter. If split is less than or equal to *i* then *i* is divided by split to calculate variable *n*, which is number of words there will be in one split.

After is calculated, the words in the text file are divided into the number of splits taken from the user by split and at the same time each Split is assigned to each Mapper. If more words remain in the text file, even after all Mappers are done taking □n□ number of words each, the remaining words are added to the last Mapper with the □n□ number of words. We created Mappers by creating a class named map where each object *ob* represents each Mapper and each word1 in that object represents each word in the object or Mapper. After all the splitting and assigning to Mappers are done each Mapper generates an intermediate *key/value* pair for every word. Here the key is the word itself and value is "1" for each word. For the Combiner function, in each Mapper, every word is compared with every other word in that particular Mapper to see if that word exists more than once in that Mapper. If the word exists more than once then the value part for that *key/value* pair is replaced with the exact amount.

4.1.2. Reduce Task Execution

To process Shuffling and Sorting, we gathered all the words/keys with values from all the Mappers in word4 array. For the Sorting part we used Quicksort function (which is the fastest sorting algorithm) to sort all the words in word4, hence bringing the same keys with values together, hence getting Shuffled. Finally, for the Reducer part all the values of the same.

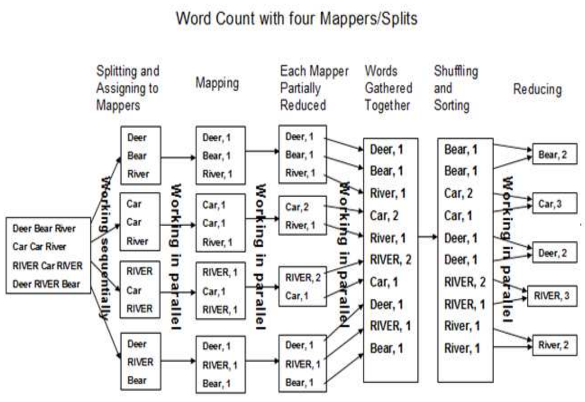

Fig.2. Word Count with Original MapReduce Algorithm

Keys/words are added together and it is done by calling separate Reducers for different keys, that is, each Reducer is different for each particular type of keys. For this we created a reduce class, where each word3 of each object *ob2* represents a Reducer containing a key and the reduced value of that key, where each Reducer adds all the values of a particular type of keys/words directly from word4 where Shuffling and Sorting was done. In Fig. 1, we demonstrated word count with original MapReduce algorithm.

4.2. Word Count with Our Simplified MapReduce

Word count with our Simplified MapReduce Algorithm has been stated in Algorithm 1.

```
Algorithm 1: Word Count with Our Simplified MapReduce Algorithm
1  i:=no. of words in txt file;
2  split:=input no. of mappers from user;
3  if split > i then
4      error;
5  else if split <= 0 then
6      error ;
7  else
8      n:=i/split;
9      splitting, assigning, mapping() function starts;
10 splitting, assigning, mapping() for mapper(1),for ob [1]
11     q:=1;
12     do
13         word:=word from file;
14         if split or mapper = last AND words remaining > n then
15             n:=n+1;
16         if word = no more words then
17             break;
18         ob.word1.key:=word;
19         ob.word1.value:=1;
20         q:=q+1;
21     while q <= n;
22 At the same time as mapper(1), start from line 10. For mapper(2), for ob[2], from q:= n+1
     word to next n no. of words in file;
23 Each mapper or ob partially reduced in parallel with another mapper then Shuffling with
     Reducing() Function is called;
24 Shuffling with Reducing()
25     j := total no. of reduced mappers;
26     j1:=1;
27     do
28         for every word OR key in mapper (j1) OR ob[j1] do
29             result:=ob.word1.value;
30             if no ob1 for this key type then
31                 ob1(new).word2.key:=ob.word1.key;
32                 ob1(new).word2.value:=ob.word1.value;
33                 index:=next empty position in this ob1;
34                 ob1(new).word2[1].index:=index;
35                 ob2(new).word3.key:=ob1(new).word2.key;
36                 ob2(new).word3.value:=ob1(new).word2.value;
37             else if ob1 exist for this key type then
38                 ob1.word2[index].key := ob.word1.key;
39                 ob1.word2[index].value := ob.word1.value;
40                 index := next empty position in this ob1;
41                 ob1.word2[1].index := index;
42                 result := result + ob2(for this key type).word3.value;
43                 ob2(for this key type).word3.value := result;
44         j1:=j1+1;                                  1
45     while j1 <= j;
```

4.2.1. Map Task Execution

Like the original MapReduce the number of words presented in the text file is counted and taken into variable *i*. Then an input is taken from the user as variable *split* to know how many Splits/Mappers there should be, that is, how many parts the text file will be divided into. If *split* is more than *i* then an error message will be shown and will be asked to re-enter, also if *split* is equal to *0*, then error message will be shown saying that splits cannot be *0* and again asked to re-enter. If *split* is less than or equal to *i* then *i* is divided by *split* to calculate variable *n*, which is number of words there will be in one Split. After □n□ is calculated, we created Mappers by creating a *class* named *map* where each object *ob* represents each Mapper and each *word1* in that object represents each word in the object/Mapper. Next, for our **first part**, Splitting (where the words in the text file are divided into the number of splits taken from the user by *split)*, assigning to Mappers and Mapping (where intermediate *key/value* pair is generated) are all done in parallel, that is, while one splitting, assigning and Mapping are all starting together for one □n□ number of words, another splitting, assigning and Mapping together, for next □n□ number of words, are also starting at the same time. As before, for the Mapping part, the *key* is the word itself and *value* is "1" for each word. If more words remain in the text file even after all Mappers are done taking □n□ number of words each, then the remaining words are added to the last Mapper with the □n□ number of words. For the Combiner function, our work is same as the original MapReduce work. In each Mapper, every word is compared with every other word in that particular Mapper to see if that word exists more than once in that Mapper. If the word exists more than once then the *value* part for that *key/value* pair is replaced with the exact amount.

4.2.2. Reduce Task Execution

Our **second part** starts from here. After the Combiner function, for parallel Shuffling with Reducing, we created a *shuffle* class where each object *ob1* and each *word2* in every *ob1* represents a *key/value* pair for that particular type of *keys*. We created Reducers as before where we created a *reduce* class, where each *word3* of each object *ob2* represents a Reducer containing a *key* and the reduced *value* of that *key*. Shuffling with reducing together is processed like this:

- If the word/*key* in *ob.word1* from Mapper does not exist as an object *ob1* in *shuffle* class then an *ob1* object for that *key* is created and entered in *ob1.word2* along with the *value*. To minimize the searching part (**third part**), the *next empty position* in that *ob1* after the *current* position (where the *key/value* pair is entered) is also stored in that *ob1.word2* as variable *index*, hence searching for an empty position in that *ob1* is not required. At the same time, *ob2.word3* in *reduce* class is created for that *key* where the *key* and the corresponding *value* from *ob1.word2* is entered.
- If the word/*key* in *ob.word1* from Mapper does exist as an object *ob1* in *shuffle* class then using the *index*, the *next empty position* in that *ob1* is located and there the new *key* of same type with *value* is entered from Mapper along with *index* for the *next empty position* in that *ob1*. The *index* variable should always be with the first *word2* in every *ob1* so that procedure does not have to waste time finding the *index* variable. At the same time, the *value* of the new-but-same-type *key* of *ob1.word2* is added with the existing *value* (for the previous same-type key) of *ob2.word3*, and then the existing *value* is replaced with the added *value* in *ob2.word3* in *reduce* class.

This goes on for every word from all Mappers, one word after another and one Mapper after another. Fig. 2, represents word count with our Simplified Algorithm.

5. Result and Performance Analysis

To measure the time for parallel Splitting with Mapping we took the time taken to complete one Splitting, Assigning and Mapper’s task together, and to measure the time for separate Split-ting-Assigning and Mapper’s task, we took the time taken for the all the Splitting and Assigning to complete and added it with the time taken for one Mapper’s task to complete. To measure the time taken to Partially Reduce each Mapper we took the time taken for one Mapper to complete the reducing task. To measure the time for parallel Shuffling with Reducing we took the time taken to complete the whole parallel Shuffling with Reducing, and to measure the time for separate Shuffling/Sorting and Reducing we took the time taken for the whole Shuffling/Sorting to complete and added it with the time taken for one Reducer’s task to complete. Comparing the times for separate Splitting-Assigning and Mapping and separate Shuffling/Sorting and Reducing with parallel Splitting-Assigning with Mapping and parallel Shuffling with Reducing, we see that parallel Splitting-Assigning with Mapping and parallel Shuffling with reducing show less times than separate Splitting-Assigning and Mapping and separate Shuffling/Sorting and Reducing. Table 1 illustrated Comparison Time between Original Algorithm and the Our Simplified Algorithm. From Table 1, Total time to complete a job, T=T1+T2+T3. In Fig. 3, we have shown the total job completion times for 1200 word count by using 1, 600 and 1100 mappers for original algorithm and refined algorithm. Subsequently in Fig. 4, the total job completion times for 2400 word count by using 1, 600 and 1100 mappers for original algorithm and refined algorithm have been visualized. The illustration of the total job completion times for 3600 word count by using 1, 600 and 1100 mappers for original algorithm and refined algorithm has been given in Fig. 5.

Table I.: Performance Comparison between Original Algorithm and Proposed Simplified Algorithm

Original Algorithm

	Time for Separate Splitting Assigning and Mapping in micro-second, T_1			Time for Reducing each Mapper Partially in micro-second, T_2			Time for Separate Shuffling /Sorting and Reducing in micro-second, T_3		
Mapper	1	600	1100	1	600	1100	1	600	1100
Words									
1200	15000	17000	18000	143000	0	1000	0	7000	6000
2400	30000	31000	33000	570000	0	2000	0	19000	21000
3600	43000	42000	44000	1285000	0	5000	0	26000	28000

Our Simplified Proposed Model

	Time for Separate Splitting Assigning and Mapping in micro-second, T_1			Time for Reducing each Mapper Partially in micro-second, T_2			Time for Separate Shuffling /Sorting and Reducing in micro-second, T_3		
Mapper	1	600	1100	1	600	1100	1	600	1100
Words									
1200	15000	1000	1000	146000	0	1000	0	2000	2000
2400	31000	1000	3000	572000	0	2000	0	3000	3000
3600	44000	1000	3000	1289000	0	5000	0	4000	4000

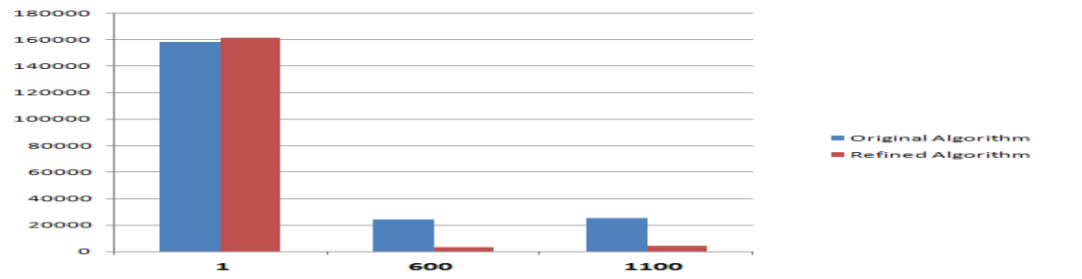

Fig. 3. The total job completion times for 1200 word count by using 1,600 and 1100 mappers for original and refined algorithm

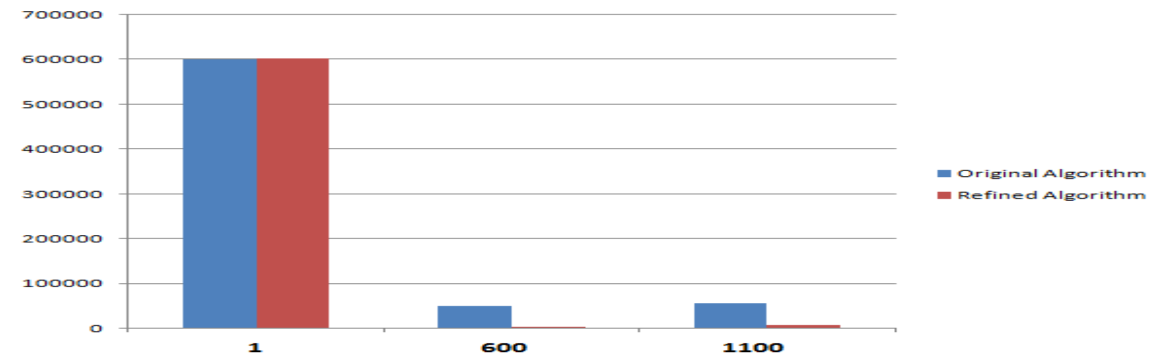

Fig. 4. The total job completion times for 2400 word count by using 1,600 and 1100 mappers for original and refined algorithm

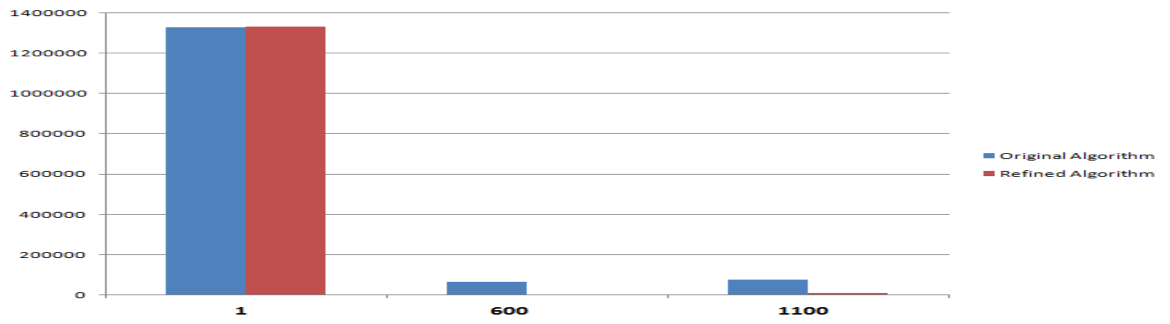

Fig. 5. The total job completion times for 3600 word count by using 1,600 and 1100 mappers for original and refined algorithm

Our time taken to complete the job for only one Mapper is more than in the original algorithm because in our algorithm splitting, assigning, and mapping are all working together for all the words in the file for one Mapper. Whereas in the original algorithm splitting and assigning to Mapper is working separately from mapping for all the words in the file for one Mapper and the time has been taken to just do the mapping job separately is zero for any number of Mappers. But when number of Mappers is two or more than two then our time is less because we are counting the time taken for only one Mapper (the last Mapper) to finish splitting, assigning and Mapping together. Whereas in the original one, time is taken for the whole splitting and assigning to complete for all the words in the file for all the Mappers and then added with the time taken to complete mapping for one Mapper (the last Mapper). We are choosing the last Mapper because the last Mapper can have more words than preceding Mappers as described in Section 5.2. Number of Mappers, in practice, should always be at the very least two or more than two**,** otherwise MapReduce will not live up to its name of parallel paradigm.

6. Conclusion

MapReduce has added new dimension for large scale parallel programming. Our paper demonstrated that our modified MapReduce algorithm gives better performance to process huge data with lesser time than traditional MapReduce algorithm. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance and locality optimization. Second, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems. We have used publicly available source code from Google for our research work, which can be downloaded from [32].

References

[1] Welcome to Apache™ Hadoop®! (n.d.). Retrieved August 28, 2017, from http://hadoop.apache.org/
[2] Shaikh Muhammad Allayer, Md. Salahuddin, Faishal Ahmed and Sung Soon Park: Introducing iSCSI Protocol on Online Based MapReduce Mechanism. ICCSA 2014: Computational Science and Its Applications – ICCSA 2014 pp 691-706.
[3] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.
[4] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *TKDE*, 23(9):1282–1298, 2011.
[5] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD*, pages 961–972, 2011.
[6] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960, 2011.
[7] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-joinprocessing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.
[8] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495– 506, 2010.
[9] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.
[10] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509,2012.
[11] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.
[12] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.
[13] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *SIGKDD*, pages 837–846, 2009.
[14] G. D. F. Morales, A. Gionis, and M. Sozio. Social content matching in mapreduce. *PVLDB*, 4(7):460–469, 2011.
[15] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454– 465, 2012.

[16] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938– 948, 2010.S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

[17] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*,pages 271–280, 2007.

[18] R. L. F. Cordeiro, C. T. Jr., A. J. M. Traina, J. Lopez, U. Kang, and C. Faloutsos. Clustering very large multi-dimensional datasets with mapreduce. In *SIGKDD*, pages 690–698, 2011.

[19] A. Ene, S. Im, and B. Moseley. Fast clustering using mapreduce. In *SIGKDD*, pages 681–689, 2011.

[20] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426– 1437, 2009.

[21] A. Ghoting, P. Kambadur, E. P. D. Pednault, and R. Kannan. Nimble: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In *SIGKDD*, pages 334–342, 2011.

[22] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4(11):1135–1145, 2011.

[23] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *PVLDB*, 5(10):1028– 1039, 2012.

[24] R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In *ICDE*, pages 486– 497, 2012.

[25] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.

[26] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in mapreduce. In *WWW*, pages 231–240, 2010.

[27] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[28] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized page rank on mapreduce. In *SIGMOD*, pages 973–984, 2011.

[29] J. Jestes, F. Li, and K. Yi. Building wavelet histograms on large data in mapreduce. In *PVLDB*, pages 617–620, 2012.

[30] Abhishek Verma, Nicolas Zea, Brian Cho, Indranil Gupta, Roy H. Campbell: Breaking the MapReduce Stage Barrier.

[31] Google Code Archive - Long-term storage for Google Code Project Hosting. (n.d.). Retrieved August 28, 2017, from http://code.google.com/p/hop