LETTER

Skew-Tolerant Key Distribution for Load Balancing in MapReduce

Jihoon SON[†a)], **Hyunsik CHOI**[††b)], ***Nonmembers***, ***and*** **Yon Dohn CHUNG**[††c)], ***Member***

SUMMARY MapReduce is a parallel processing framework for large scale data. In the reduce phase, MapReduce employs the hash scheme in order to distribute data sharing the same key across cluster nodes. However, this approach is not robust for the skewed data distribution. In this paper, we propose a skew-tolerant key distribution method for MapReduce. The proposed method assigns keys to cluster nodes balancing their workloads. We implemented our proposed method on Hadoop. Through experiments, we evaluate the performance of the proposed method in comparison with the conventional method.

key words: *skew-tolerance, MapReduce, load balance, key distribution*

1. Introduction

The MapReduce framework [1] is proposed for parallel processing of a huge volume of data. MapReduce consists of two steps, *map* and *reduce*, and adopts the *key-value* data model. In the map step, each mapper scans and processes data sequentially. After the map step, the intermediate results of the map step are grouped by their keys, and then the grouped results are transmitted to reducers as their inputs. In this process, data sharing the same key are assigned to the same reducer.

However, this approach is not robust for the skewed data. After the map step, keys of intermediate results are assigned to reducers evenly, but data are assigned according to their keys. If the data distribution for keys is skewed, the amount of assigned data for each reducer is also skewed. This may cause significant performance degradation, and thus uniform data assigning is important.

This is a well-known problem of the *load balancing* in parallel/distributed systems. The load balancing problem is known as NP-hard [2]. There have been some research works for load balancing in MapReduce, but they still have some scalability problems [3]–[5].

In this paper, we propose a *skew-tolerant key distribution method* for the MapReduce framework. In the proposed method, workloads are *evenly* distributed to reducers by uniform assignment of data to each of them. Due to the evenly distributed workloads, the total run time is reduced compared to the conventional method. Our contributions are summarized as follows.

- We model the load balancing problem in MapReduce.
- We propose a *skew-tolerant key distribution method* which reduces the overall run time by distributing workloads to reducers evenly.
- We propose an implementation of the proposed method without modifying the MapReduce framework.
- We show the performance evaluation results comparing our proposed method and the conventional one.

2. Related Work

There have been many studies to address the load balancing problem. [2] presented a greedy solution for the general load balancing problem. However, this solution requires a large memory space, which may cause the scalability problem.

In parallel database management systems (PDBMSs), there have been research works for the load balancing. DeWitt et al. proposed a load balancing method which uses multiple algorithms specialized for different data skewness [6]. In [7], the authors proposed hierarchical PDBMSs which are shared-nothing systems whose nodes are shred-memory multiprocessors. Rahm et al. proposed a multi-resource load balancing method [8], [9]. In these methods, the combination of inter-query/inter-transaction is considered for complex queries in multi-resource systems. However, the above methods focused on PDBMSs, and thus are hard to be applied to MapReduce directly.

Recently, some studies have been presented for solving the load balancing problem in MapReduce [3]–[5]. In [3], authors proposed a system, called *Scarlett*, which adopts a replication policy based on access patterns of files. Scarlett decides the number of replicas of each file based on its access frequency, and places them on as many distinct machines as possible. In this paper, we focus on the key assignment method for load balancing in MapReduce.

Shadi Ibrahim et al. [5] proposed *LEEN* which is a key partitioning method considering the data locality and the workload distribution. During the map phase, each mapper collects the frequencies of keys. Before the shuffle phase, LEEN sorts keys by their localities and then assigns them to reducers in the sorted order while balancing the workload distribution. This assignment information is used by the partitioner in the shuffle phase. This method requires to maintain the mapping table of keys and reducers, which may cause the scalability problem similar to [2].

Manuscript received August 31, 2011.

[†]The author is with the Center for Advanced Mobile Solutions, Korea University, Seoul, Korea.

[††]The authors are with the Department of Computer Science and Engineering, Korea University, Seoul, Korea.

a) E-mail: jihoonson@korea.ac.kr

b) E-mail: hyunsikchoi@korea.ac.kr

c) E-mail: ydchung@korea.ac.kr

DOI: 10.1587/transinf.E95.D.677

In [4], the authors proposed an approximation algorithm based on the clustering to handle the scalability problem. In this method, keys are clustered into a fixed number of clusters. After that, clusters are assigned to reducers in the same manner of the greedy solution in [2]. Here, some clusters which have too heavy workloads are splitted into several sub-clusters and each sub-clusters are assigned to reducers. The performance of algorithm is highly affected by the number of clusters because workloads are approximated within each cluster. However, details of the cluster construction algorithm and the optimal number of cluster is not presented in [2].

3. Problem Definition

In this paper, we focus on the load balancing problem in MapReduce. We are given a MapReduce cluster consisting of m identical nodes, $N_1, N_2, N_3, \ldots, N_m$ and n intermediate results, where each result has a key k_j and d_j data ($1 \leq j \leq n$). For each intermediate result, an *assignment* specifies the node to which the key is assigned and the amount of data assigned to it. Any key can be assigned to any node. Here, let K_i denote the set of keys assigned to the node N_i. D_i, the total amount of data assigned to the node N_i, is,

$$D_i = \sum_{k_j \in K_i} d_j. \quad (1)$$

Since the workloads of a key grows with the increase of data amount related to the key, the total workloads of N_i also grows with the increasing D_i. The problem we solve in the paper is to find the assignment whose D_{max} is minimum, where $D_{max} = \max_{1 \leq i \leq m} D_i$.

Since the load balancing problem is NP-hard, we adopt the greedy approach for the heuristic solution.

4. The Proposed Method

4.1 Skew-Tolerant Key Distribution Method

In this paper, we propose a novel key distribution method in MapReduce, called *Skew-tolerant Key Distribution* (SKD). In SKD, workloads are evenly distributed to nodes by the uniform distribution of data. In this paper, we assume that we already know the data distribution for keys. It can be easily obtained by scanning the whole data once. We also assume that the cluster consists of identical nodes.

In the load balancing problem of MapReduce, the assignment is ideal when every node is assigned $D_{mean} = \frac{\sum_{i=1}^{n} d_i}{m}$ data, because $D_{max} = D_{mean}$ is minimized. Thus, each node has D_{mean} capacity for assigned data in SKD. However, the ideal assignment is not realistic because there are different amounts of data sharing the same key. If nodes are restricted to be assigned to up to D_{mean} data, in most of cases, some nodes have assigned data less than D_{mean}, but others have more data. To handle this problem, SKD uses a small value D_θ as a threshold for allowing to assign a little more amount of data to each node. That is, each node are allowed to be assigned maximally $D_{mean} + D_\theta$ data. The threshold D_θ must be a value between 0 and $d_{max} = \max_{1 \leq j \leq n} d_j$. We'll discuss about the reasonable value for D_θ in Sect. 4.2.

Algorithm 1 Balanced key assignment algorithm

```
1:  procedure BalancedKeyAssignment(a set of keys K, a set of data D, D_mean, D_θ)
2:      Sort K by IDs.
3:      A ← φ                                          ▷ A is an assignment.
4:      j ← 1
5:      while j ≤ n do
6:          k_j ∈ K
7:          for each node N_i ∈ N do
8:              K_i ← φ
9:              while D_i ≤ D_mean do
10:                 K_i ← K_i ∪ k_j
11:                 D_i ← D_i + d_j
12:                 j ← j + 1
13:             end while
14:             if D_i + d_j ≤ D_mean + D_θ then
15:                 K_i ← K_i ∪ k_j
16:                 D_i ← D_i + d_j
17:                 j ← j + 1
18:             end if
19:             A ← A ∪ K_i
20:         end for
21:     end while
22:     return A
23: end procedure
```

SKD consists of two algorithms, the *balanced key assignment (BKA) algorithm* and the *partition algorithm*. BKA is a preprocessing step for SKD. Before the MapReduce process, keys are assigned to cluster nodes considering their workloads. When BKA begins, it sorts keys by their IDs and chooses a node N_i. While $D_i \leq D_{mean}$ is satisfied, keys are assigned to N_i in the sorted order. When a key k_j is selected for assigning, if $(D_i + d_j) > D_{mean}$, the key k_j is assigned to N_i if the condition $(D_i + d_j) \leq (D_{mean} + D_\theta)$ is satisfied. After that, BKA selects a next node N_h where $i \neq h$ and $1 \leq h \leq m$, and continues assigning keys. This is repeated until all keys are assigned.

Algorithm 1 shows our key assignment algorithm. The input parameters of the algorithm are a set of keys K, a set of data D, D_{mean}, and D_θ. At Line 2, keys are sorted by their IDs. From Line 7 to Line 13, a node N_i is selected and keys are assigned to N_i while $D_i \leq D_{mean}$ is satisfied. From Line 14 to Line 18, a key k_j is assigned to N_i if $(D_i + d_j) \leq (D_{mean} + D_\theta)$. At Line 22, BKA returns the assignment A.

After the map phase, the partition algorithm is called for distribution of intermdediate results. For each result, this algorithm finds the node which has a key range including the key of the result, and returns the ID of the found node.

4.2 Discussion of the Threshold D_θ

In this section, we'll discuss about the value of D_θ. For each node, D_θ is used for checking the last key whether it is assigned to the node or not. If D_θ is too small, most of nodes would be assigned less than D_{mean} data, but the last assigned node would be assigned more than D_{mean} data. On the contrary, if D_θ is too big, most of nodes would be assigned more than D_{mean} data, but the last assigned node would be assigned less than D_{mean} data.

Let suppose the worst case. The worst case is when every last key which has d_{max} data failed to be assigned to every node except the last assigned node.

Lemma 4.1: In the worst case, $D_\theta > \frac{m-1}{m} \times d_{max}$.

Proof Let the last assigned node as N_m. In the worst case, every node has data of the amount of $(D_{mean} + D_\theta - d_{max})$ except N_m. In this case, D_m, the amount of assigned data of N_m, is $((d_{max} - D_\theta) \times (m-1) + D_{mean})$. Here, the condition $D_m \leq (D_{mean} + D_\theta)$ should be satisfied.

$$\begin{aligned} &(d_{max} - D_\theta) \times (m-1) + D_{mean} < D_{mean} + D_\theta \\ &(d_{max} - D_\theta) \times (m-1) < D_\theta \\ &\frac{m-1}{m} \times d_{max} < D_\theta \end{aligned} \tag{2}$$

Therefore, Lemma 4.1 holds. □

Lemma 4.1 is for the worst case, and thus not suitable to be used in general. In SKD, we define an average threshold for the general case as follows.

Definition 1: Given a set of unordered keys, the *expected workload* is defined as the expectation of the amount of data for a randomly chosen key k_i. The expectation is defined as follows: $\mathbb{E}[d] = \frac{\sum_{i=1}^{n} d_i}{n}$

Definition 2: The *average threshold* is defined as follows: $D_\theta = \frac{m-1}{m} \times \mathbb{E}[d]$

When the average threshold is used, SKD assigns all remaining keys to the last node D_m to ensure that every key is assigned. This may cause the skewed assignment. However, the workload distribution becomes similar to that of the optimal as the number of keys increases. In Sect. 5, we show that the workload distribution of the proposed method is similar with that of the greedy algorithm.

4.3 Implementation Details

SKD can be easily implemented as the *partitioner class* on MapReduce. We implemented *skew-tolerant partitioner* (STP) class on the Hadoop framework. STP maintains the key assignment which is the result of BKA, and informs the proper reducer ID for the query key. Here, note that the assigned keys are sorted by their IDs. STP doesn't require a data structure which maintains the assignment of keys to nodes such as a mapping table between keys and nodes. Instead, STP maintains only the range of assigned keys for each node. Therefore, it requires very low memory space and key search overhead. Also, STP can be easily used in existing MapReduce programs by choosing STP instead of the *hash partitioner*.

5. Performance Evaluation

In this section, we present the experiment results of SKD for load balancing in MapReduce. We have implemented SKD on the Hadoop (version 0.21) [11].

5.1 Experiment Setup

Our hadoop cluster consists of 8 identical nodes. Each cluster node has the i5 760 CPU and the 1 GB RAM, and is connected to other nodes with 100 Mbps LAN.

We implemented the *equi-join* on the Hadoop framework for the performance evaluation. The equi-join is one of the most representative applications of MapReduce. This can be implemented in various ways. We used the repartition join method [12] which is the most commonly used join algorithm in MapReduce. In this algorithm, two relations (L and R) are dynamically partitioned on the join key and the corresponding pairs of partitions are joined.

We experimented three methods: the conventional method [1], the greedy method [2] and the proposed method. The conventional method is the default key distribution method of Hadoop. It uses the hash key distribution and is denoted by HKD in our evaluation results. GRD denotes the greedy algorithm. Note that all the previous load balancing methods [2], [4], [5] basically follow the *greedy* heuristics, and thus are not scalable at all in the MapReduce environment. This is because each node is required to maintain a (possibly unlimited) set of keys.

For the equi-join experiment, we used the modified TPC-H [13] to generate skewed data. We first experimented three methods with varying key cardinalities to compare the scalability. Table 1 shows the comparison of memory usage of GRD and SKD. As shown in the table, the memory usage of GRD is increased linearly with the increasing key cardinality, and finally returns the *out of memory* error for the data with 150 million keys due to the lack of memory space. However, SKD uses only 1 KB memory for storing the key ranges assigned to each reducer. HKD doesn't require any memory space because it uses the hash scheme instead of any data structures to distribute keys.

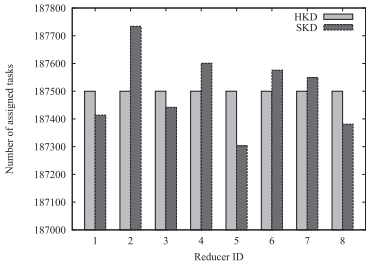
5.2 Analysis of Results

Figure 1 shows the key and data assignment for each reducer. In Fig. 1 (a), keys are assigned to reducers evenly in HKD. However, in SKD, the key assignment is skewed for the uniform distribution of workloads. Note that the data distribution is skewed for keys. As it can be seen in Fig. 1 (b), the amount of assigned data is uniform in SKD whereas it is very skewed in HKD. Furthermore, the data assignment of SKD is very similar with that of GRD.

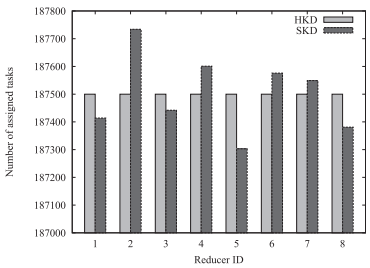
Figure 2 shows the task timeline of HKD and SKD. The task timeline of SKD begins after some small amount of time taken by BKA. As it can be seen in the graph, BKA

Table 1 Comparison of the memory usage.

Data size	1 GB	10 GB	100 GB
Key cardinality	1.5 million	15 million	150 million
Memory usage of GRD	22.89 MB	228.89 MB	n/a
Memory usage of SKD	1 KB	1 KB	1 KB
Memory usage of HKD	0	0	0

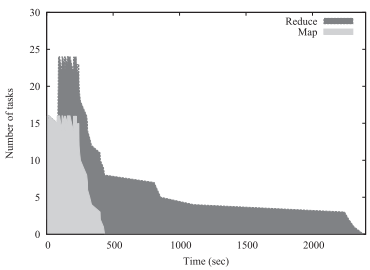


(a) Key assignment

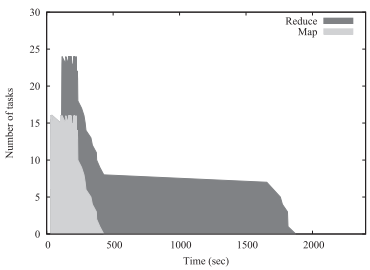


(b) Data assignment

Fig. 1 Key and data assignment.



(a) Task timeline of HKD



(b) Task timeline of SKD

Fig. 2 Task timeline.

takes very small amount of time (about 28 seconds). On the other hand, the whole run time of SKD is shorter than that of HKD because more keys are processed simultaneously in SKD than in HKD.

6. Conclusion

In this paper, we proposed a skew-tolerant key distribution method, SKD, for the load balancing in MapReduce. SKD consists of balanced key assignment algorithm (BKA) and the partition algorithm. In BKA, keys are assigned to each node until the assigned workload is larger than the average workload. Here, one more key can be assigned to the node if the amount of assigned data of the node is less than the threshold. The mapping information is maintained as the range of assigned keys for each node. When the reduce phase starts, the partition algorithm notifies the node ID which processes each intermediate result. The very small amount of memory space is required to maintain the assignment and it doesn't grow with the increasing amount of data. This accomplishes the scalability. Furthermore, the total run time is reduced due to uniform distribution of workloads in SKD. Our experiment results indicated that SKD is more efficient and scalable than the conventional methods.

Acknowledgments

This work (Grants No. 00043827-1) was supported by Business for Cooperative R&D between Industry, Academy, and Research Institute funded Korea Small and Medium Business Administration in 2010.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," Commun. ACM, vol.51, pp.107–113, Jan. 2008.
[2] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson, Introduction to Algorithms, 2nd ed., McGraw-Hill Higher Education, 2001.
[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A.G. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in MapReduce clusters," EuroSys, pp.287–300, 2011.
[4] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handling data skew in MapReduce," Proc. 2nd International Conference on Cloud Computing and Services Science, pp.100–109, Noordwijkerhout, The Netherlands, 2011.
[5] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for MapReduce in the cloud," CloudCom, pp.17–24, 2010.
[6] D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," Proc. 18th International Conference on Very Large Data Bases, pp.27–40, San Francisco, CA, USA, 1992.
[7] L. Bouganim, D. Florescu, and P. Valduriez, "Dynamic load balancing in hierarchical parallel database systems," Proc. 22th International Conference on Very Large Data Bases, pp.436–447, San Francisco, CA, USA, 1996.
[8] E. Rahm and R. Marek, "Dynamic multi-resource load balancing in parallel database systems," Proc. 21th International Conference on Very Large Data Bases, pp.395–406, San Francisco, CA, USA, 1995.
[9] E. Rahm, "Dynamic load balancing in parallel database systems," Proc. Second International Euro-Par Conference on Parallel Processing, vol.I, Euro-Par '96, pp.37–52, London, UK, 1996.
[10] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel DBMS," Proc. VLDB Endow., vol.2, pp.1390–1396, Aug. 2009.
[11] Apache Software Foundation, "Hadoop," 2006. http://hadoop.apache.org/core
[12] 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," Proc. 2010 International Conference on Management of Data, pp.975–986, New York, NY, USA, 2010.
[13] TPC-H Benchmark. http://www.tpc.org/tpch/