J Supercomput (2013) 66:539–555
DOI 10.1007/s11227-013-0924-9

An improved partitioning mechanism for optimizing massive data analysis using MapReduce

Kenn Slagter · Ching-Hsien Hsu · Yeh-Ching Chung · Daqiang Zhang

Published online: 11 April 2013

Abstract In the era of Big Data, huge amounts of structured and unstructured data are being produced daily by a myriad of ubiquitous sources. Big Data is difficult to work with and requires massively parallel software running on a large number of computers. MapReduce is a recent programming model that simplifies writing distributed applications that handle Big Data. In order for MapReduce to work, it has to divide the workload among computers in a network. Consequently, the performance of MapReduce strongly depends on how evenly it distributes this workload. This can be a challenge, especially in the advent of data skew. In MapReduce, workload distribution depends on the algorithm that partitions the data. One way to avoid problems inherent from data skew is to use data sampling. How evenly the partitioner distributes the data depends on how large and representative the sample is and on how well the samples are analyzed by the partitioning mechanism. This paper proposes an improved partitioning algorithm that improves load balancing and memory consumption. This is done via an improved sampling algorithm and partitioner. To evaluate the proposed algorithm, its performance was compared against a state of the art partition-

K. Slagter · Y.-C. Chung
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

K. Slagter
e-mail: kennslagter@sslab.cs.nthu.edu.tw

Y.-C. Chung
e-mail: ychung@cs.nthu.edu.tw

C.-H. Hsu (✉)
Department of Computer Science, Chung Hua University, Hsinchu, Taiwan
e-mail: chh@chu.edu.tw

D. Zhang
School of Software Engineering, Tongji University, Shanghai, China
e-mail: dqzhang@ieee.org

ing mechanism employed by TeraSort. Experiments show that the proposed algorithm is faster, more memory efficient, and more accurate than the current implementation.

Keywords TeraSort · MapReduce · Load balance · Partitioning · Sampling · Cloud computing · Hadoop

1 Introduction

Over the past decades, computer technology has become increasingly ubiquitous. Personal computers, smart phones, tablets, and an ever-growing number of embedded devices can now all connect and communicate with each other via the internet. Computing devices have numerous uses and are essential for businesses, scientists, governments, engineers, and the everyday consumer. What all these devices have in common is the potential to generate data. Essentially, data can come from anywhere. Sensors gathering climate data, a person posting to a social media site, or a cell phone GPS signal are example sources of data.

The popularity of the Internet alongside a sharp increase in the network bandwidth available to users has resulted in the generation of huge amounts of data. Furthermore, the types of data created are as broad and diverse as the reasons for generating it. Consequently, most types of data tend to have their own unique set of characteristics as well as how that data is distributed.

Data that is not read or used has little worth, and can be a waste of space and resources. Conversely, data that is operated on or analyzed can be of inestimable value. For instance, data mining in business can help companies increase profits by predicting consumer behavior, or discover hitherto unknown facts in science or engineering data. Unfortunately, the amount of data generated can often be too large for a single computer to process in a reasonable amount of time. Furthermore, the data itself may be too large to store on a single machine. Therefore, in order to reduce the time it takes to process the data, and to have the storage space to store the data, software engineers have to write programs that can execute on two or more computers and distribute the workload among them. While conceptually the computation to perform maybe simple, historically the implementation has been difficult.

In response to these very same issues, engineers at Google developed the Google File System (GFS) [4], a distributed file system architecture model for large-scale data processing and created the MapReduce [3] programming model. The MapReduce programming model is a programming abstraction that hides the underlying complexity of distributed data processing. Consequently, the myriad minutiae on how to parallelize computation, distribute data, and handle faults no longer become an issue. This is because the MapReduce framework handles all these details internally, and removes the onus of having to deal with these complexities from the programmer.

Hadoop [25] is an open source software implementation of MapReduce, written in Java, originally developed by Yahoo!. Hadoop is used by various universities and companies including EBay, FaceBook, IBM, LinkedIn, and Twitter. Hadoop was created in response to the need for a MapReduce framework that was unencumbered by proprietal licenses, as well as the growing need for the technology in Cloud computing [18].

Since its conception, Hadoop has continued to grow in popularity amongst businesses and researchers. As researchers and software engineers use Hadoop they have at the same time attempted to improve upon it by enhancing features it already has, by adding additional features to it, or by using it as a basis for higher-level applications and software libraries. Pig, HBase, Hive, and ZooKeeper are all examples of commonly used extensions to the Hadoop framework [25].

Similarly, this paper also focuses on Hadoop and investigates the load balancing mechanism in Hadoop's MapReduce framework for small- to medium-sized clusters. This is an important area of research for several reasons. First, many clusters that use Hadoop are of modest size. Often small companies, researchers and software engineers do not have the resources to develop large cluster environments themselves, and often clusters of a modest size are all that is required for certain computations. Furthermore, it is common for developers creating Hadoop applications to use a single computer running a set of virtual machines as their environment. Limited processing power and memory necessitates a limited number of nodes in these environments.

In summary, this paper presents the following contributions:

- A method for improving the work load distribution amongst nodes in the MapReduce framework.
- A method to reduce the required memory footprint.
- Improved computation time for MapReduce when these methods are executed on small or medium sized cluster of computers.

The rest of this paper is organized as follows. Section 2 presents some background information on MapReduce and its inner workings. Section 3 introduces an improved load balancing methodology and a way to better utilize memory. Section 4 contains experimental results and a discussion of this paper's findings. Section 5 presents related work. Section 6 concludes this paper and briefly discusses future work.

2 Background and preliminaries

2.1 MapReduce

MapReduce [3] is a programming model developed as a way for programs to cope with large amounts of data. It achieves this goal by distributing the workload among multiple computers and then working on the data in parallel. From the programmers perspective MapReduce is a relatively easy way to create distributed applications compared to traditional methods. It is for this reason MapReduce has become popular and is now a key technology in cloud computing.

Programs that execute on a MapReduce framework need to divide the work into two phases known as Map and Reduce. Each phase has key-value pairs for both input and output [32]. To implement these phases, a programmer needs to specify two functions: a map function called a Mapper and its corresponding reduce function called a Reducer.

When a MapReduce program is executed on Hadoop, it is expected to be run on multiple computers or nodes. Therefore, a master node is required to run all the

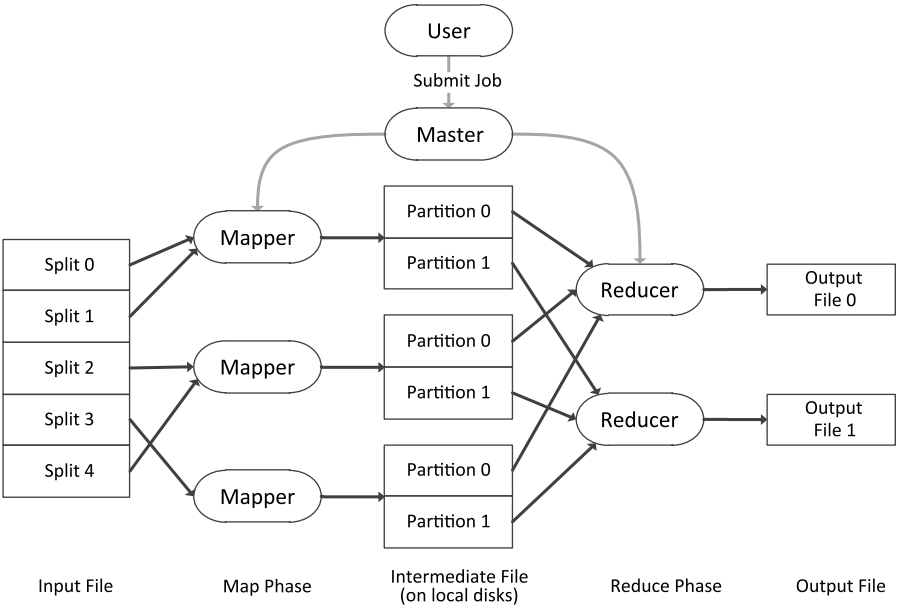

Fig. 1 MapReduce Dataflow

required services needed to coordinate the communication between Mappers and Reducers. An input file (or files) is then split up into fixed sized pieces called input splits. These splits are then passed to the Mappers who then work in parallel to process the data contained within each split. As the Mappers process the data, they partition the output. Each Reducer then gathers the data partition designated for them by each Mapper, merges them, processes them, and produces the output file. An example of this dataflow is shown in Fig. 1.

It is the partitioning of the data that determines the workload for each reducer. In the MapReduce framework, the workload must be balanced in order for resources to be used efficiently [7]. An imbalanced workload means that some reducers have more work to do than others. This means that there can be reducers standing idle while other reducers are still processing the workload designated to them. This increases the time for completion since the MapReduce job is not complete until all reducers finish their workload.

2.2 HashCode

Hadoop uses a hash code as its default method to partition key-value pairs. The hash code itself can be expressed mathematically and is presented in this paper as the following equation:

$$\begin{aligned} HashCode &= W_n \times 31^{n-1} + W_{n-1} \times 31^{n-2} + \cdots + W_1 \times 31^0 \\ &= \sum_{n=1}^{TotalWord} W_n \times 31^{n-1} \end{aligned} \tag{1}$$

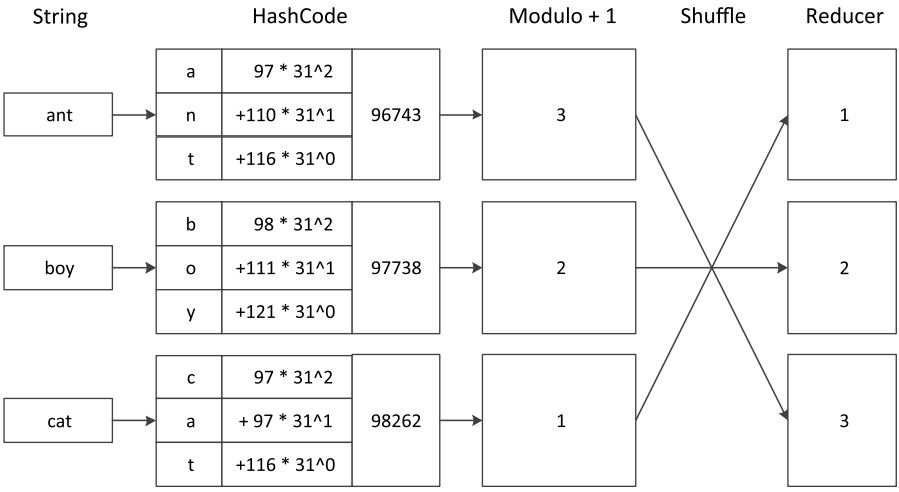

Fig. 2 HashCode partitioner

The hash code presented in Eq. (1) is the default hash code used by a string object in Java, the programming language on which Hadoop is based. In this equation, W_n represents the nth element in a string. The reason integer 31 is used in this equation is because it is a prime number. Hash codes traditionally use prime numbers because they have a better chance of generating a unique value when multiplied with another number.

A partition function typically uses the hash code of the key and the modulo of reducers to determine which reducer to send the key-value pair to. It is important then that the partition function evenly distributes key-value pairs among reducers for proper workload distribution.

Figure 2 shows how the hash code works for a typical partitioner. In this example, there are three reducers, and three strings. Each string comes from a key in a key-value pair. The first string is "ant." The string "ant" consists of three characters. The characters "a," "n," and "t" have the corresponding decimal ASCII values of 97, 110, and 116. These values are then used with Eq. (1) to get the hash code value of 96743. Since there are three reducers, a modulo of 3 is used which gives a value of 2. The value is then incremented by one in the example as there is no reducer 0, which changes the value to 3. This means the key-value pair will be sent to reducer 3. Following the same methodology, the strings "boy" and "cat" are assigned to reducers 2 and 1, respectively.

2.3 TeraSort

In April 2008, Hadoop broke the world record in sorting a Terabyte of data by using its TeraSort [24] method. Winning first place it managed to sort 1 TB of data in 209 seconds (3.48 minutes). This was the first time either a Java program or an open source program had won the competition. TeraSort was able to accelerate the sorting process by distributing the workload evenly within the MapReduce framework. This

was done via data sampling and the use of a trie [13]. Although the original goal of TeraSort was to sort 1 TB of data as quickly as possible, it has since been integrated into Hadoop as a benchmark.

Overall, the TeraSort algorithm is very similar to the standard MapReduce sort. Its efficiencies rely on how it distributes its data between the Mappers and Reducers. To achieve a good load balance, TeraSort uses a custom partitioner. The custom partitioner uses a sorted list of $N - 1$ sampled keys to define a range of keys for each reducer. In particular, a key is sent to a reducer i if it resides within a range such that $sample[i-1] <= key < sample[i]$. This ensures that the output of reducer i is always less than the output for reducer $i + 1$.

Before the partitioning process for TeraSort begins, it samples the data and extracts keys from the input splits. The keys are then saved to a file in the distributed cache [16]. A partitioning algorithm then processes the keys in the file. Since the original goal of TeraSort was to sort data as quickly as possible, its implementation adopted a space for time approach. For this purpose, TeraSort uses a two-level trie to partition the data.

A trie, or prefix tree, is an ordered tree used to store strings. Throughout this paper, a trie that limits strings stored in it to two characters is called a two-level trie. Correspondingly, a three-level trie stores strings of up to three characters in length, a four-level trie stores strings of up to four characters in length and an n level trie stores strings of up to n characters in length.

This two-level trie is built using cut points derived from the sampled data. Cut points are obtained by dividing a sorted list of strings by the total number of partitions and then selecting a string from each dividing point. The partitioner then builds a two-level trie based on these cut points. Once the trie is built using these cutpoints, the partitioner can begin its job of partition strings based on where in the trie that string would go if it were to be inserted in the trie.

3 The proposed techniques and trie optimizations

This section introduces the concept of the Xtrie and describes how TeraSort can be modified using an Xtrie and how Xtrie can benefit TeraSort. Furthermore, this section introduces the Etrie and shows how it can save memory using a ReMap method.

3.1 Xtrie

The Xtrie algorithm presented here provides a way to improve the cut point algorithm inherited from TeraSort. One of the problems with the TeraSort algorithm is that it uses the quicksort algorithm to handle cut points. By using quicksort, TeraSort needs to store all the keys it samples in memory and that reduces the possible sample size, which reduces the accuracy of the selected cut points and this affects load balancing [24]. Another problem TeraSort has is that it only considers the first two characters of a string during partitioning. This also reduces the effectiveness of the TeraSort load balancing algorithm.

Algorithm 1 Trie Cut Point Algorithm

Input:

IS: set of input strings
i: index in the trie array
trieSize: size of the trie array
partitionCount: total number of partitions
prefixCount: number of prefixes in the trie array
k: partition number

Output:

OS: a set of partitioned strings

Create IS by extracting n samples from source data.

```
1.  counter = 0
2.  for each string S in IS
3.          tc = TrieCode(S)
4.          if(trie[tc] == FALSE)
5.                  prefixCount = prefixCount + 1
6.          end if
7.          trie[tc] = TRUE
8.  end for
9.  splitSize = prefixCount / partitionCount
10. for i = 0 to trieSize
11.         if(trie[i] == TRUE)
12.             split = split + 1
13.             if (split >= splitSize)
14.                     k = k + 1
15.             end if
16.             OSk.add(trie[i])
17.         endif
18. end for
```

A trie has two advantages over the quicksort algorithm. Firstly, the time complexity for insertion and search of the trie algorithm is O(k) where k is the length of the key. Meanwhile, the quicksort algorithm best and average case is O($n \log n$) and in the worst case O(n^2) where n is the number of keys in its sample. Secondly, a trie has a fixed memory footprint. This means the number of samples entered into the trie can be extremely large if so desired. Algorithm 1 shows how this trie is used.

In Algorithm 1, the trie is an array accessed via a trie code. A trie code is similar to a hashcode, but the codes it produces occur in sequential ASCIIbetical order. The equation for the trie code is as follows:

$$\begin{aligned} TrieCode &= W_n \times 256^{n-1} + W_{n-1} \times 256^{n-2} + \cdots + W_1 \times 256^0 \\ &= \sum_{n=1}^{TotalWord} W_n \times 256^{n-1} \end{aligned} \tag{2}$$

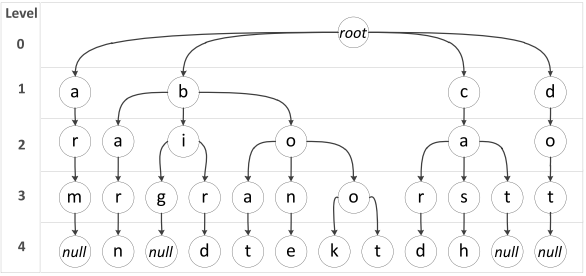

Fig. 3 Strings stored in a trie

Table 1 XTrie partitioner using a Two-level Trie

Prefix	Keys	Trie code	Count	Partition
ar	arm	0x6172	1	1
ba	barn	0x6261	1	
bi	big bird	0x6269	2	
bo	boat bone book boot	0x626f	4	2
ca	card cash cat	0x6361	3	3
do	dot	0x646f	1	

The problem with using a conventional trie is that it fails to reflect strings that share the same prefix. This can result in an uneven distribution of keys and an imbalanced workload.

As an example, consider the set of keys {“arm,” “barn,” “big,” “bird,” “boat,” “bone,” “book,” “boot,” “card,” “cash,” “cat,” and “dot”} as stored in the trie in Fig. 3. Since a two-level trie can only contain the first two characters of a string, the strings are truncated and the set is reduced to {“ar,” “ba,” “bi,” “bo,” “ca,” “do”}. If there are three reducers, the workload would be divided into three partitions. Each partition is then sent to a different reducer. This results in reducer-1 processing keys starting with {“ar,” “ba”}, reducer-2 processing keys starting with {“bi,” “bo”}, and reducer-3 processing keys starting with {“ca,” “do”}. Consequently, reducer-1 processes two keys, reducer-2 processes 6 keys and reducer-3 processes 4 keys. This workload is imbalanced, as ideally all three reducers should process the same number of keys.

In order to ameliorate this problem Xtrie uses a counter for each node in the trie. By using a counter, keys have proportional representation and the partitioner can distribute the total number of keys among reducers more evenly. Table 1 assumes a two-level trie is used and that three partitions are required. In Table 1, each prefix keeps count of how many keys are associated with it. Once all the keys have been inserted in the trie, the keys are divided up amongst the partitions based on the prefix count. In this example, the total number of keys is $1 + 1 + 2 + 4 + 3 + 1 = 12$.

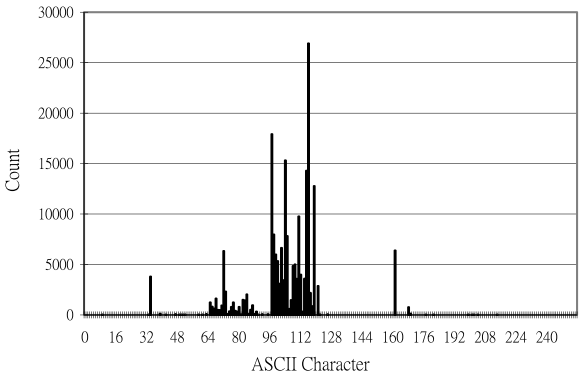

Fig. 4 Data distribution of English text

Once the partitions are made and the keys are divided up among reducers, this results in reducer-1 processing keys starting with {"ar," "ba," "bi"}, reducer-2 processing keys starting with {"bo"}, and reducer-3 processing keys starting with {"ca," "do"}. Consequently, reducer-1, reducer-2, and reducer-3 process 4 keys each.

Using this methodology, a two level Xtrie can now produce the same results as the TeraSort quicksort method but as expected, it does so using a smaller memory footprint and requiring less processing time. An insight analysis and performance evaluation will be discussed in the next section.

3.2 Etrie

One of the problems inherited by TeraSort and Xtrie is that they use an array to represent the trie. The main problem with this technique is that it tends to contain a lot of wasted space. For example, plain text files and log files often contain only rudimentary alphanumerical characters, whitespace, line breaks, and punctuation marks. Furthermore, when processing text, the whitespace, line breaks, and punctuation marks are often filtered out by the Mapper. Moreover, many kinds of applications only use a small range of keys, resulting in a lot of wasted space by the trie. For example, if plain English text is being sorted the data distribution may be skewed in a way similar to Fig. 4, whereby characters and numbers constitute nearly all of the characters to be stored by the trie.

This paper therefore presents the ReMap algorithm, which reduces the memory requirements of the original trie by reducing the number of elements it considers. The algorithm does this by placing each ASCII character into one of the three categories: *null*, *alphanumeric*, or *other*. Alphanumeric characters are upper and lower case letters or numbers. Other is used for all characters that are not alphanumeric except for null. Null is used to represent strings that have fewer characters than the number of levels in the trie. We present an example of the ReMap layout in Table 2.

The ReMap chart maps each of the 256 characters on an ASCII chart to the reduced set of elements expected by the Etrie. Since the purpose of Etrie is to reflect words found in English text ReMap reassigns the ASCII characters to the 64 elements shown in Table 3.

Table 2 Etrie ReMap chart

Hex	ASCII	Etrie Code	Hex	ASCII	Etrie Code	Hex	ASCII	Etrie Code
0x00	*null*	0x00	0x36	6	0x07	0x5b	[	0x3f
0x01	SOH	0x3f	0x37	7	0x08	...	...	...
0x02	STX	0x3f	0x38	8	0x09	0x60	`	0x3f
0x03	ETX	0x3f	0x39	9	0x0a	0x61	a	0x25
...	...	...	0x3a	:	0x3f	0x62	b	0x26
0x2f	/	0x3f	0x41	A	0x0b	0x63	c	0x27
0x30	0	0x01	0x42	B	0x0c	...	...	...
0x31	1	0x02	0x43	C	0x0d	0x78	x	0x3c
0x32	2	0x03	...	...	...	0x79	y	0x3d
0x33	3	0x04	0x58	X	0x22	0x7a	z	0x3e
0x34	4	0x05	0x59	Y	0x23	0x7b	{	0x3f
0x35	5	0x06	0x5a	Z	0x24		*other*	0x3f

Table 3 Etrie element chart

ASCII char	Etrie Code	ASCII char	Etrie Code
null	0x00	A	0x0b
0	0x01	...	
1	0x02	Z	0x24
2	0x03	a	0x25
...		...	
8	0x09	z	0x3e
9	0x0a	*other*	0x3f

By reducing the number of elements to consider from 256 to 64 elements per level, the total memory required is reduced to 1/16th of its original footprint for a two-level trie. In order to use the Etrie, the TrieCode presented in Eq. (2) has to be modified. The EtrieCode shown in Eq. (3) is similar to the TrieCode in Eq. (2), but has been altered to reflect the smaller memory footprint. The EtrieCode equation is as follows:

$$\begin{aligned} EtrieCode &= W_n \times 64^{n-1} + W_{n-1} \times 64^{n-2} + \cdots + W_1 \times 64^0 \\ &= \sum_{n=1}^{TotalWord} W_n \times 64^{n-1} \end{aligned} \tag{3}$$

Using less memory per level allows deeper tries (tries with more levels) to be built. Deeper tries provides opportunities for distributing keys evenly amongst reducers. Consider the set of keys {"baby," "back," "bald," "ball," "bare," "bark," "barn," "bath," "bone"} as stored in the trie in Fig. 5. Attempting to partition keys evenly using a two-level trie is not feasible in this case. Using a two-level trie would create the set of prefixes {"ba," "bo"}. Although one can identify that the prefix "ba" represents 8 keys and the prefix "bo" represents 1 key, the prefix themselves are indivisible, which creates a distribution imbalance. This problem can be resolved by

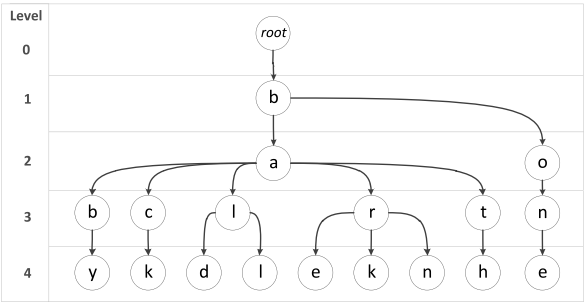

Fig. 5 A trie unbalanced by strings with the same prefix

using a deeper trie. By using a deeper trie, one increases the length of each prefix. In practice, this increases the number of prefixes that are considered, and reduces the number of keys each prefix represents. This provides a finer grain with which to create partitions, thereby creating a more even distribution of keys between partitions. For instance, when using the same set of keys, a three level trie would produce the prefixes {"bab," "bac," "bal," "bar,", "bat," "bon"}, thus dividing the 8 keys formerly represented by the prefix "ba" up into five smaller categories.

4 Performance evaluation and analysis

To evaluate the performance of the proposed algorithms, this study investigates how well the algorithms distribute the workload, and looks at how well the memory is utilized. Experiments conducted in this study were done using 100 eBooks, with each eBook containing a minimum of 100,000 words. Using the eBooks as our input, we simulated computer networks from 5 nodes in size to 100 nodes in size.

In order to compare the different methodologies presented in this paper and determine how balanced the workload distributions are, this study uses a metric called the uneven rate. The uneven rate α is calculated using the following equations:

Let V = optimal partition size = total keys / total partitions
Let S_n = number of keys in partition n.
Let $\Delta Sn = |Sn - V|$
Then

$$\alpha = \frac{\Delta S_n + \Delta S_{n-1} + \cdots + \Delta S_1}{TotalPartitions} \div V$$
$$\alpha = \frac{\sum_{n=1}^{TotalPartitions} \Delta S_n}{TotalPartitions} \div V \tag{4}$$

In order to evaluate the memory consumption of each method, we defined a metric called the space utilization rate, which we present in the following equation:

β: space utilization rate
A: occupied elements in the trie array
B: total elements in the trie array

$$\beta = \frac{A}{B} \tag{5}$$

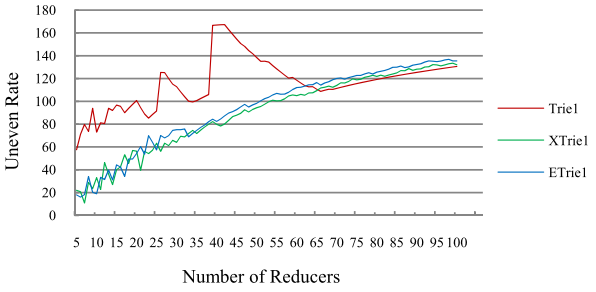

Fig. 6 Uneven rate for one level trie

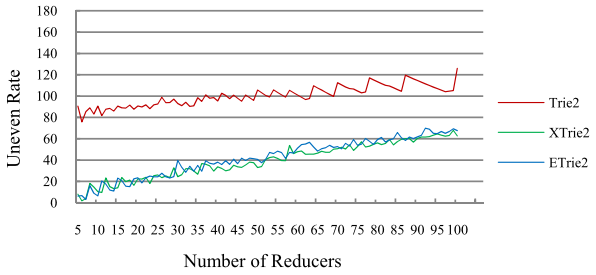

Fig. 7 Uneven rate for two level trie

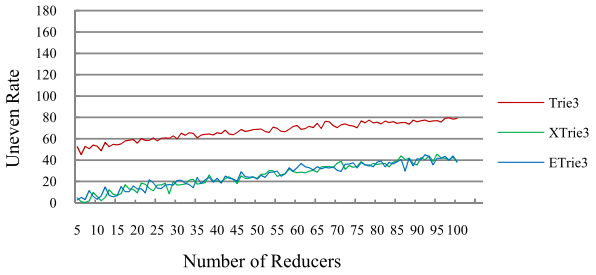

Fig. 8 Uneven rate for three level trie

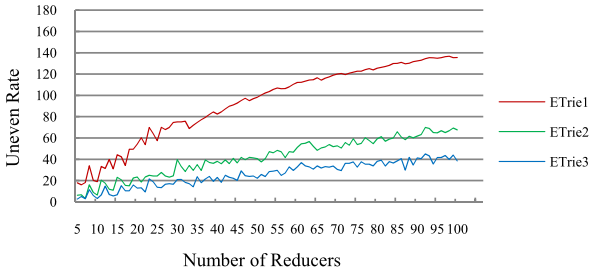

Fig. 9 Uneven rate between different level Etrie

Figures 6 to 9 show that when the number of reducers increase the rate of unevenness increases. In other words, as more reducers become available, the more difficult it is to divide the workload evenly between them.

Figures 6 to 8 show that both Xtrie an Etrie exhibit similar behaviors and that the unevenness rate decreases the more levels they have, especially when the number of nodes is low. In Fig. 6, the uneven rate of a conventional trie shows comparable

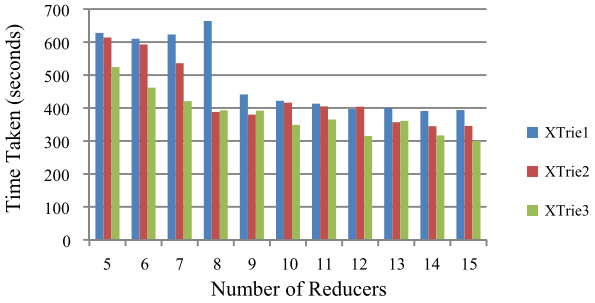

Fig. 10 Sort Time Taken between different level XTrie

performance to Xtrie and Etrie methods after the reducers exceeded 60 in number. The convergence of the methodologies occurs due to the limited number of elements used to represent the trie. As the number of nodes increases, the ability to distribute the keys evenly among the nodes reduces.

Figures 7 and 8 shows that for two and three level tries, Xtries and Etries exhibit similar workload behaviors, both having nearly identical rates of unevenness. Furthermore, they both show a much smaller rate of unevenness (smaller being better) than the conventional trie, especially when the number of computers used are small.

In Fig. 9, Etries are compared to each other but using different number of levels. Because the key prefixes in a one level trie is so short the keys are coarsely grouped together within the trie and a poor uneven rate. Etries using two or three levels have a lot lower uneven rate due to their being more finer differentiation between keys. In English, the average word length is 4.5 letters long [33], and longer words tend to be used less often than smaller ones. Therefore, it is expected that there is a limit on how much increasing the number of levels trie will improve the uneven rate.

According to experimental results, the uneven rate is lower (lower being better) when a trie has more levels. This is because the deeper a trie is the longer the prefix each key represents. This results in a finer differentiation between individual keys. This makes it easier for the partitioner to distribute keys more evenly among reducers thereby lowering the uneven rate.

In a second experiment, we measured the time taken for different types of XTrie, the results are shown in Fig. 10. The physical machine used in this experiment had 2 Intel Xeon CPUs running at 2.93 GHz, with 6 cores per CPU. An 11 GB file was used as input. Results show that as the number of levels in the XTrie increased the unevenness typically decreased, and consequently the time taken decreased.

In a third experiment, measurements were taken to compare how time efficient the XTrie algorithm is compared to quicksort algorithm used by the TeraSort. As shown in Fig. 11, as more and more samples were included the time taken by quicksort method grows rapidly as expected by its time complexity.

In a fourth experiment, measurements were taken to compare how well the Xtrie and Etrie utilized space, where T1, T2, and T3 represent XTrie using one-level, two-level, and three-level tries, respectively. Similarly, ET1, ET2, and ET3 represent one-level, two-level, and three-level Etries. The space utilization efficiency is presented as follows in Table 4.

As can be seen in Table 4, Etrie space utilization is better than the Xtrie space utilization no matter how many levels are used to represent the trie. However, no

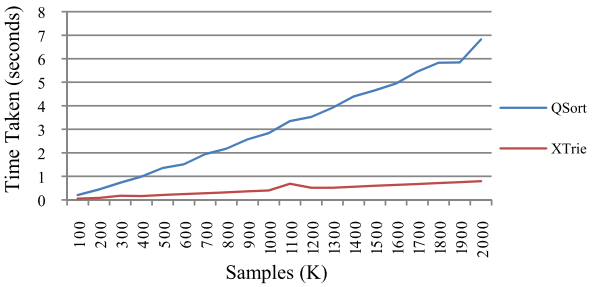

Fig. 11 Time complexity comparison of QuickSort and XTrie

Table 4 Space utility

	T1	T2	T3	ET1	ET2	ET3
Number of nodes	77	693	3080	56	608	2903
Space required	256	65,536	16,777,216	64	4,096	262,144
Space utilization (%)	30.08	1.06	0.02	87.5	14.84	1.11

matter which method was used, space utilization suffered as the number of levels in a trie increased. This is because the array representing the trie becomes increasingly sparse as its size increases.

Overall, according to the experimental results, Xtrie and ETrie outperform TeraSort's partitioning method in time complexity. Furthermore, Etrie has much better space utilization compared to other tries, and the two level ETrie is 14 times more space efficient than the two level XTrie.

5 Related work

Sorting is a fundamental concept and is required step in countless algorithms. Various sorting algorithms have been created over the years including bubble sort, quick sort, merge sort, and so on. Different sorting algorithms are better equipped for sorting different problems. Burst Sort [31] is a sorting algorithm designed for sorting strings in large data collections. The implementation involves building a burst trie, which is an efficient data structure for storing strings, and requires no more memory than a binary tree. The burst trie is fast as a trie, but was not as fast as a hash table. The TeraSort algorithm also uses these trie methods as a way to sort data but does so under the context of the Hadoop architecture and the MapReduce framework.

An important issue for the MapReduce framework is the concept of load balancing. Over the years, a lot of research has been done on the topic of load balancing. This is because how data is distributed and work shared amongst the components of a physical machine or nodes in network can be a key contributing factor to the efficiency of a system. Where data is located [10], how it is communicated [15], what environment it resides on [19, 26, 28] and the statistical distribution of the data can all have an affect on a systems efficiency. Many of these algorithms can be found worldwide in various papers and have been used by frameworks and systems prior to

the existence of the MapReduce framework [8, 17]. Some of the less technical load balancing techniques are round robin, random, or shortest remaining processing time. While these techniques are well known, they have been found either inappropriate or inadequate for the task of sorting data on the MapReduce framework.

In the MapReduce framework, the workload must be balanced in order for resources to be used efficiently. An imbalanced workload means that some reducers have more work to do than others do. This means that there can be reducers standing idle while other reducers are still processing the workload designated to them. This increases the time for completion since the MapReduce job cannot complete until all reducers finish their workload.

By default, Hadoop's workload is balanced with a hash function. However, this methodology is generic and not optimal for many applications. For instance, RanKloud [1] uses its own *uSplit* mechanism for partitioning large media data sets. The *uSplit* mechanism is needed to reduce data replication costs and wasted resources that are specific to its media based algorithms.

In order to work around perceived limitations of the MapReduce model, various extend or change the MapReduce models have been presented. BigTable [2] was introduced by Google to manage structured data. BigTable resembles a database, but does not support a full relational database model. It uses Rows with consecutive keys grouped into tablets, which form the unit of distribution and load balancing. And suffers from the same load and memory balancing problems faced by shared-nothing databases. The open source version of BigTable is Hadoop's HBase [27], which mimics the same functionality of BigTable.

Due to its simplicity of use, the MapReduce model is quite popular and has several implementations [9] [11, 12]. Therefore, there has been a variety of research on MapReduce in order to improve the performance of the framework or the performance of specific applications like datamining [14, 22], graph mining [5], text analysis [20], or genetic algorithms [6, 21] that run on the framework.

Occasionally, researchers find the MapReduce framework to be too strict or inflexible in its current implementation. Therefore, researchers sometimes suggest new frameworks or suggest new implementations as a solution. One such framework is Dynamically ELastic MApReduce(DELMA) [23].

DELMA is a framework that follows the MapReduce paradigm, just like Hadoop MapReduce. However, it is capable of growing and shrinking its cluster size, as jobs are underway. This framework extends the MapReduce framework so that nodes can be added or removed while applications are running on the system. Such a system is likely to have interesting load balancing issues, which is beyond the scope of our paper.

Another alternative framework to MapReduce is Jumbo [30]. In [30], the authors state that some of the properties of MapReduce makes load balancing difficult. Furthermore, Hadoop does not provide many provisions for workload balancing. For these reasons, the authors created Jumbo a flexible framework that processes data in a different way from MapReduce. One of the drawbacks of MapReduce is that multiple jobs may be required for some complex algorithms, which limits load balancing efficiency. Due to the way it handles data, Jumbo is able to execute these complex algorithms in a single job. The Jumbo framework may be a useful tool with which to

research load balancing, but it is not compatible with current MapReduce technologies.

Finally, to work around load balancing issues derived from joining tables in Hadoop, [29] introduces an adaptive MapReduce algorithm for multiple joins using Hadoop that works without changing its environment. It does so by taking tuples from smaller tables and redistributing them among reducers via ZooKeeper, which is a centralized coordination service. Our paper also attempts to do workload balancing in Hadoop without modifying the underlying structure, but focuses on sorting text.

6 Conclusion and future work

This paper presented *XTrie* and *ETrie*, extended partitioning techniques, to improve load balancing for distributed applications. By improving load balancing, MapReduce programs can become more efficient at handling tasks by reducing the overall computation time spent processing data on each node. The TeraSort, developed by O'Malley, Yahoo, was designed based on randomly generated input data on a very large cluster of 910 nodes. In that particular computing environment and for that data configuration, each partition generated by MapReduce appeared on only one or two nodes. In contrast, our study looks at small- to medium-sized clusters. This study modifies their design and optimizes it for a smaller environment. A series of experiments have shown that given a skewed data sample, the ETrie architecture was able to conserve more memory, was able to allocate more computing resources on average, and do so with less time complexity. In future, further research can be done extending this framework so that it can use different micropartitioning methods for applications using different input samples.

Acknowledgements We would like to thank the various colleagues in the System Software Laboratory at National Tsing Hua University as well as my colleagues at the Department of Computer Science and Information Engineering in Chung Hua University for their support and for their help on earlier drafts of this paper.

References

1. Candan KS, Kim JW, Nagarkar P, Nagendra M, Yu R (2010) RanKloud: scalable multimedia data processing in server clusters. IEEE MultiMed 18(1):64–77
2. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrws M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: 7th UENIX symposium on operating systems design and implementation, pp 205–218
3. Dean J, GhemawatDean S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51:107–113
4. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. In: 19th ACM symposium on operating systems principles (SOSP)
5. Jiang W, Agrawal G (2011) Ex-MATE data intensive computing with large reduction objects and its application to graph mining. In: IEEE/ACM international symposium on cluster, cloud and grid computing, pp 475–484
6. Jin C, Vecchiola C, Buyya R (2008) MRPGA: an extension of MapReduce for parallelizing genetic algorithms. In: IEEE fourth international conference on escience, pp 214–220
7. Kavulya S, Tany J, Gandhi R, Narasimhan P (2010) An analysis of traces from a production MapReduce cluster. In: IEEE/ACM international conference on cluster, cloud and grid computing, pp 94–95

8. Krishnan A (2005) GridBLAST: a globus-based high-throughput implementation of BLAST in a grid computing framework. Concurr Comput 17(13):1607–1623
9. Liu H, Orban D (2011) Cloud MapReduce: a MapReduce implementation on top of a cloud operating system. In: IEEE/ACM international symposium on cluster, cloud and grid computing, pp 464–474
10. Hsu C-H, Chen S-C (2012) Efficient selection strategies towards processor reordering techniques for improving data locality in heterogeneous clusters. J Supercomput 60(3):284–300
11. Matsunaga A, Tsugawa M, Fortes J (2008) Programming abstractions for data intensive computing on clouds and grids. In: IEEE fourth international conference on escience, pp 489–493
12. Miceli C, Miceli M, Jha S, Kaiser H, Merzky A (2009) Programming abstractions for data intensive computing on clouds and grids. In: IEEE/ACM international symposium on cluster computing and the grid, pp 480–483
13. Panda B, Riedewald M, Fink D (2010) The model-summary problem and a solution for trees. In: International conference on data engineering, pp 452–455
14. Papadimitriou S, Sun J (2008) Distributed co-clustering with map-reduce. In: IEEE international conference on data mining, p 519
15. Hsu C-H, Chen SC (2010) A two-level scheduling strategy for optimizing communications of data parallel programs in clusters. Int J Ad Hoc Ubiq Comput 6(4):263–269
16. Shafer J, Rixner S, Cox AL (2010) The hadoop distributed filesystem: balancing portability and performance. In: IEEE international symposium on performance analysis of system and software (ISPASS), p 123
17. Stockinger H, Pagni M, Cerutti L, Falquet L (2006) Grid approach to embarrassingly parallel CPU-intensive bioinformatics problems. In: IEEE international conference on e-science and grid computing
18. Tan J, Pan X, Kavulya S, Gandhi R, Narasimhan P (2009) Mochi: visual log-analysis based tools for debugging hadoop. In: USENIX workshop on hot topics in cloud computing (HotCloud)
19. Hsu C-H, Tsai B-R (2009) Scheduling for atomic broadcast operation in heterogeneous networks with one port model. J Supercomput 50(3):269–288
20. Vashishtha H, Smit M, Stroulia E (2010) Moving text analysis tools to the cloud. In: IEEE world congress on services, pp 110–112
21. Verma A, Llor'a X, Goldberg DE, Campbell RH (2009) Scaling genetic algorithms using MapReduce. In: International conference on intelligent systems design and applications
22. Xu W, Huang L, Fox A, Patterson D, Jordan M (2009) Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles (SOSP)
23. Fadika Z, Govindaraju M (2011) DELMA: dynamic elastic MapReduce framework for CPU-intensive applications. In: IEEE/ACM international symposium on cluster, cloud and grid computing, pp 454–463
24. O'Malley O (2008) TeraByte sort on Apache hadoop
25. Apache software foundation (2007) Hadoop. http://hadoop.apache.org/core
26. Hsu C-H, Chen T-L, Park J-H (2008) On improving resource utilization and system throughput of master slave jobs scheduling in heterogeneous systems. J Supercomput 45(1):129–150
27. HBase. http://hadoop.apache.org/hbase/
28. Zaharia M, Konwinski A, Joseph AD, Katz R, Stoica I (2008) Improving MapReduce performance in heterogeneous environments. In: OSDI
29. Lynden S, Tanimura Y, Kojima I, Matono A (2011) Dynamic data redistribution for MapReduce joins. In: IEEE international conference on cloud computing technology and science, pp 713–717
30. Groot S, Kitsuregawa M (2010) Jumbo: beyond MapReduce for workload balancing. In: VLDB, PhD workshop
31. Heinz S, Zobel J, Williams H (2002) Burst tries: a fast, efficient data structure for string keys. ACM Trans Inf Syst 20(12):192–223
32. Hsu C-H, Chen S-C, Lan C-Y (2007) Scheduling contention-free irregular redistribution in parallelizing compilers. J Supercomput 40(3):229–247
33. Shannon CE (1951) Prediction and entropy of printed English. Bell Syst Tech J 30:50–64