Int J Parallel Prog (2015) 43:489–507
DOI 10.1007/s10766-013-0288-z

An Adaptive and Memory Efficient Sampling Mechanism for Partitioning in MapReduce

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

Received: 17 April 2013 / Accepted: 4 October 2013 / Published online: 8 November 2013

Abstract Big Data refers to the massive amounts of structured and unstructured data being produced every day from a wide range of sources. Big Data is difficult to work with and needs a large number of machines to process it, as well as software capable of running in a distributed environment. MapReduce is a recent programming model that simplifies writing distributed programs on distributed systems. For MapReduce to work it needs to divide work amongst computers in a network. Consequently, the performance of MapReduce is dependent on how evenly it distributes the workload. This paper proposes an adaptive sampling mechanism for total order partitioning that can reduce memory consumption whilst partitioning with a trie-based sampling mechanism (ATrie). The performance of the proposed algorithm is compared to a state of the art trie-based partitioning system (ETrie). Experiments show the proposed mechanism is more adaptive and more memory efficient than previous implementations. Since ATrie is adaptive, its performance depended on the type of data used. A performance evaluation of a 2-level ATrie shows it uses 2.43 times less memory for case insensitive email addresses, and uses 1,024 times less memory for birthdates compared to that of a 2-level ETrie. These results show the potential of the proposed method.

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

K. Slagter (✉) · Y.-C. Chung
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC
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, ROC
e-mail: chh@chu.edu.tw

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 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 amongst 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) [6], a distributed file system architecture model for large-scale data processing and created the MapReduce [4] 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 [1] 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 [24].

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 [1].

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. Firstly, 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.

Furthermore, this paper delves into the world of trie-based sampling and partitioning algorithms and presents a more intelligent adaptive method for handling memory consumption and partitioning. For this purpose, we compare our ATrie method with that of ETrie. A 2-level ETrie is a trie-based sampling algorithm with equivalent functionality to the sampling algorithm used by TeraSort [18]. To function, an ETrie requires enough space to tally 4096 character combinations. The proposed method, ATrie, is also a trie-based sampling algorithm. However, ATrie analyzes its samples before it determines its space requirements. Therefore, the memory consumed by ATrie is dependent on the type of data used. For instance, when analyzing case insensitive emails a 2-level ETrie required 2.43 times more memory than a 2-level ATrie. Alternatively, a 2-Level ETrie required 1,024 times more memory than ATrie when handling 1,000,000 birthdates. Furthermore, the difference in memory requirements grew exponentially as the number of levels of each trie increased.

In summary, this paper presents the following contributions:

- A method for improving the work load distribution amongst nodes in the MapReduce framework.
- An adaptive method for reducing the required memory footprint and more intelligent string handling.
- A way to leverage users domain knowledge by providing them a regular expression based parser with which to access the adaptive methods proposed.

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 adaptive load balancing methodology that can better utilize memory and improves load balancing. 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 [4] 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 amongst multiple computers and then working on the data in parallel. From the programmers perspective MapReduce is a relatively easy way to create distributed applications

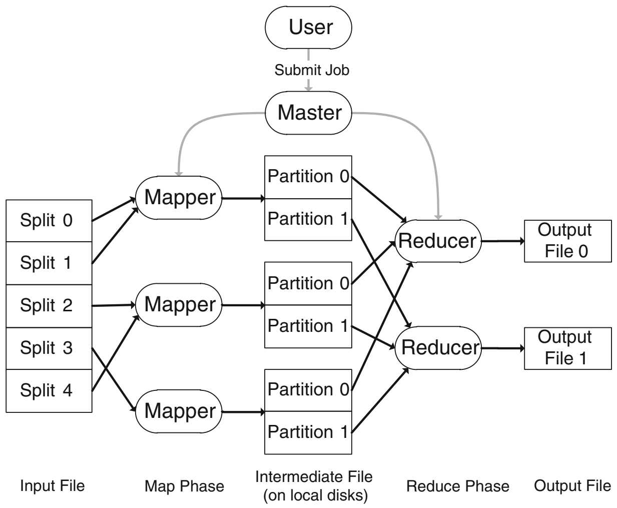

Fig. 1 MapReduce DataFlow

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 [27]. 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 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 [12]. 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 HashCodes

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.

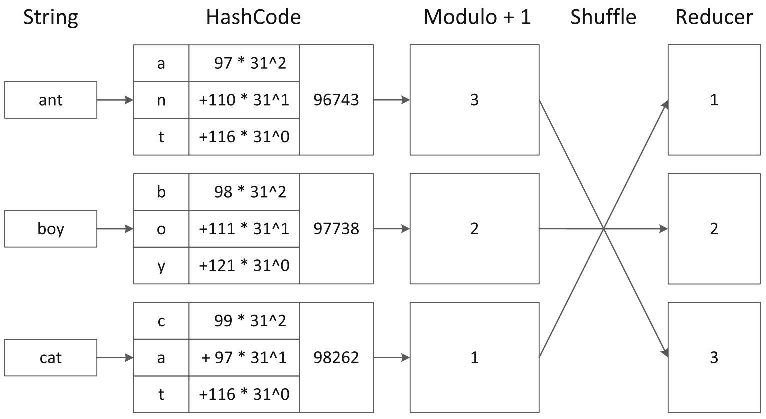

Fig. 2 HashCode Partitioner

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

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 n^{th} 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 amongst 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 [18] method. Winning first place it managed to sort 1TB of data in 209 s (3.48 min). 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 [19]. Although the original goal of TeraSort was to sort 1TB 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 [21]. 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.

2.4 XTrie

The XTrie algorithm provides a way to improve the cut point algorithm inherited from TeraSort [22]. One of the problems with the TeraSort algorithm is that it uses 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 [18]. 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.

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.

One way to represent a trie is as 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:

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

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. 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 amongst reducers more evenly. The algorithm for the XTrie is presented as follows:

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

2.5 ETrie

One of the problems that XTrie has is that the memory requirements of the trie increases rapidly with each character it analyzes. The reason for this is because that in order for

Table 1 Space requirements for an XTrie

n-Level	Space requirement
1	256
2	65,536
3	16,777,216
4	4,294,967,296

Table 2 Space requirements for an ETrie

n-Level	Space requirement
1	64
2	4,096
3	262,144
4	16,777,216

the XTrie to scan n characters of a string, it increases the array size quickly as it space requirement increases at a rate of 256^n. Consequently, XTries tend to be 2-Level tries. The accuracy of a 2-level XTrie is the same as that of TeraSort as its partitioner uses a 2-level trie (Table 1).

In order to improve memory consumption requirements an ETrie was proposed [22]. The ETrie used a ReMap method whereby a ReMap chart identified those ASCII characters that were commonly used in English text and essentially filtered out those characters that were not of interest. This reduced the number of considered characters to 64. Consequently, the space requirements needed to represent the ETrie was now 64^{n}. Although this caused the ETrie to lose a bit of accuracy compared to the same n-level XTrie, the improved the space requirements of the ETrie meant it could better partition data due to it being able to use a trie with more levels (Table 2).

The improved space requirements of an ETrie means that ETries are 3-level tries. Consequently, ETries provide a finer grain with which to create partitions, and thus more even workload distribution.

3 Background and Preliminaries

The method presented by ETrie is designed for English text. Although it substantially reduces memory consumption, there are ways to improve further the breadth and depth of Trie being used. One way to improve this is by adding a preliminary pass, whereby each string is analyzed prior to construction of the trie. Using an adaptive trie has several benefits. Firstly, it conserves memory by reducing the scope of characters it needs to consider when building the trie. Secondly, it can create deeper trie for a specified amount of memory. This means better decisions can be made on how to partition data since decisions can be made using a finer grain of granularity. Thirdly, an adaptive trie avoids problems arising from fixed prefixes or internal patterns that can cause poor partitioning (Tables 3, 4).

To create an adaptive trie one needs an adaptive triecode (ATC). The ATC is built by using multiple ReMap charts, each chart corresponding to the nth character of each sampled string including any null characters used to terminate the string. In

Table 3 Visibility charts

Character 1			Character 2			Character 3		
Hex	ASCII	Visible	Hex	ASCII	Visible	Hex	ASCII	Visible
0x61	a	True	0x61	a	True	0x61	a	False
0x62	b	True	0x62	b	False	0x62	b	False
0x63	c	True	0x63	c	False	0x63	c	False
0x64	d	True	0x64	d	False	0x64	d	False
0x65	e	True	0x65	e	False	0x65	e	False
0x66	f	True	0x66	f	False	0x66	f	False
0x67	g	False	0x67	g	True	0x67	g	True
0x6d	m	False	0x6d	m	False	0x6d	m	True
0x6e	n	False	0x6e	n	True	0x6e	n	False
0x6f	o	False	0x6f	o	True	0x6f	o	False
0x72	r	False	0x72	r	True	0x72	r	False
0x74	t	False	0x74	t	False	0x74	t	True
0x77	w	False	0x77	w	False	0x77	w	True
	Other	False		*Other*	False		*Other*	False

Table 4 ReMap charts

Character 1			Character 2			Character 3		
Hex	ASCII	Code	Hex	ASCII	Code	Hex	ASCII	Code
0x61	a	0	0x61	a	0	0x61	a	Undefined
0x62	b	1	0x62	b	Undefined	0x62	b	Undefined
0x63	c	2	0x63	c	Undefined	0x63	c	Undefined
0x64	d	3	0x64	d	Undefined	0x64	d	Undefined
0x65	e	4	0x65	e	Undefined	0x65	e	Undefined
0x66	f	5	0x66	f	Undefined	0x66	f	Undefined
0x67	g	Undefined	0x67	g	1	0x67	g	0
0x6d	m	Undefined	0x6d	m	undefined	0x6d	m	1
0x6e	n	Undefined	0x6e	n	2	0x6e	n	Undefined
0x6f	o	Undefined	0x6f	o	3	0x6f	o	Undefined
0x72	r	Undefined	0x72	r	4	0x72	r	Undefined
0x74	t	Undefined	0x74	t	undefined	0x74	t	2
0x77	w	Undefined	0x77	w	undefined	0x77	w	3
	Other	Undefined		*Other*	Undefined		*Other*	Undefined
	Total	6		*Total*	5		*Total*	4

order to create the ReMap charts a preliminary analysis pass over the strings marks the corresponding ReMap chart to identify which of the nth characters are visible amongst all the strings. Once these charts are marked for visibility the individual ReMap charts can built.

Consider the set of three letter keys { “ant”, “arm”, “art”, “bat”, “cat”, “cot”, “cow”, “dog”, “eat”, “egg”, “fat”, “fog” }. In order to build an adaptive trie code we first need to create a set of visibility charts. As we wish to consider all three characters in each string when building our adaptive trie code, we need to have three visibility charts. Once the three visibility charts have been generated each character is enumerated in each of the charts and the sum value of each chart is then determined.

Once the ReMap charts have been generated, the adaptive trie and the associated adaptive trie code has to be created. If the trie is stored as a fixed memory size array, then the space requirements for the trie is equal to the multiplied total of visible characters in each chart a shown in the following equation:

$$\begin{aligned}\text{Let } C_i &= \text{total number of visible characters in visibility chart } i\\ \text{Let T} &= \text{size of the array needed for the trie}\\ \text{Then } T &= \sum_{i=1}^{TotalCharts} C_i\end{aligned} \tag{3}$$

If a memory limit is to be imposed upon the adaptive trie space, then the number of ReMap charts can be reduced until the memory requirements are satisfied.

The adaptive trie code now needs an adaptive trie code to access it. The adaptive trie code is calculated using an irregular number system. Our decimal number system is is known as a base 10 number system. Any number in our counting system can be calculated by adding a series of 10n digits. For instance the number 214 can be represented as 2 × 102 + 1 × 101 + 4 × 100. The XTrieCode works in a similar way except it uses base 256 to index an array. In a similar fashion the ETrieCode uses base 64 to index an array. However the adaptive trie does not use a fixed base value, instead it uses the multiplied total number of visible characters in each chart. This can be expressed mathematically as the following equation:

$$\begin{aligned}\text{Let } C_i &= \text{total number of visible characters in visibility chart } i\\ ATC &= \begin{array}{l}(W_n \times (C_{i-1} \times C_{i-2} \times \cdots \times C_1))+\\ (W_{n-1} \times (C_{i-2} \times C_{i-3} \times \cdots \times C_1)) + \cdots + (W_2 \times C_1) + W_1\end{array}\end{aligned} \tag{4}$$

For example, the string “cat” can be represented as the set of ASCII characters { ‘c’, ‘a’, ‘t’ }. ReMapping each character using the tables above this converts the string to the set of numbers { 2, 0, 2 }. Then using Eq. 5 the subsequent adaptive trie code would be

$$\text{“cat”} = \{\text{‘c’} = 2, \text{‘a’} = 0, \text{‘t’} = 2\} = (2 \times (5 \times 4)) + (0 \times (4)) + 2 = 42$$

Once the values of C_0 to C_i are calculated they remain fixed throughout the lifetime of the adaptive trie. Therefore once they are calculated they can be stored and reused for the sake of efficiency. A listing of each string and subsequent adaptive trie code is presented in the following table.

In some circumstances, the user knows the type or form of the strings being analyzed ahead of time. In some situations these strings can be described in advance using a

Table 5 Adaptive Trie codes

String	ReMap	Adaptive TrieCode
ant	{ 0, 2, 2 }	10
arm	{ 0, 4, 1 }	17
art	{ 0, 4, 2 }	18
bat	{ 1, 0, 2 }	22
cat	{ 2, 0, 2 }	42
cot	{ 2, 3, 2 }	54
cow	{ 2, 3, 3 }	55
dog	{ 3, 3, 0 }	72
eat	{ 4, 0, 2 }	82
egg	{ 4, 1, 0 }	84
fat	{ 5, 0, 2 }	102
fog	{ 5, 3, 0 }	112

regular expression. Consider the situation whereby the user is sorting numbers (as strings), dates, internet ip addresses, ISBN or other pattern of ASCII characters. The user can then use regular expression to create an RE-Trie. Using regular expressions one can build a trie based on domain knowledge. For instance consider the scenario of series of dates (Table 5).

1977-05-31
1976-09-15
1982-02-23
1996-08-16
1996-04-30

These strings can be represented using the following regular expression:

$$\begin{aligned}\text{Regular Expression } &= [0-9]\{4\} - (((0[13578]|(10|12))\\ &-(0[1-9]|[1-2][0-9]|3[0-1]))\\ &|(02-(0[1-9]|[1-2][0-9]))|\\ &((0[469]|11)-(0[1-9]|[1-2][0-9]|30))) \end{aligned} \tag{5}$$

Which matches the date format YYYY-MM-DD (Year-Month-Date). This regular expression also validates month and number of days in a month (including leap year dates). Using this information a regular expression parser can be used to build an adaptive triecode. The advantage of having a regular expression used to build an adaptive trie is that one can avoid doing a preliminary pass, which means not having to read all strings twice.

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 three different test sets.

One test set used email addresses extracted from a 338MB server mail log. A second test used the same test set but with all the email addresses in lowercase. A third test set used 1,000,000 random birthdates.

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:

$$\begin{aligned}
&\text{Let } V = \text{optimal partition size} = \text{total keys / total partitions}\\
&\text{Let } S_n = \text{number of keys in partition } n.\\
&\text{Let } \Delta Sn = |Sn - V|\\
&\text{Then } \alpha = \frac{\Delta \mathrm{S_n} + \Delta \mathrm{S_{n-1}} + \cdots + \Delta \mathrm{S_1}}{TotalPartitions} \div V\\
&\alpha = \frac{\sum_{n=1}^{TotalPartitions} \Delta S_n}{TotalPartitions} \div V
\end{aligned} \tag{6}$$

Whether using an ATrie or an XTrie [22], the load balance of an n-level trie exhibits the same behavior. When using an ETrie [22], the load balance exhibits slightly worse performance for an n-level trie. TeraSort uses a 2-level trie, so its performance is similar to a 2-level XTrie and a 2-level ATrie when it comes to load balancing. There is a large difference however in memory consumption, speed of processing sampled data, and the amount of sampled data that can be handled by XTrie, ETrie and ATrie, which makes a difference.

In our first test case we examine what happens when we use a trie to partition a set of case sensitive email addresses. The results are shown in Figs. 3 and 4. As typical of trie based paritioning, as n increases for an $n-$level trie, the unevenness decreases. This is because as n increases the number of characters being considered in the string increases, making it easier to make a more even partitioning decision (Fig. 5).

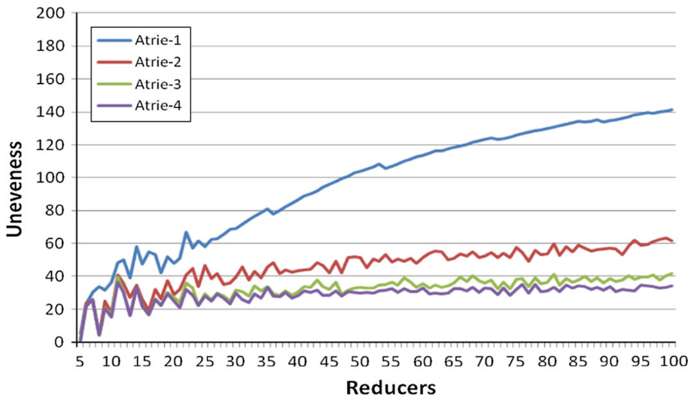

Fig. 3 Comparison of unevenness for an adaptive trie for case *sensitive* email addresses

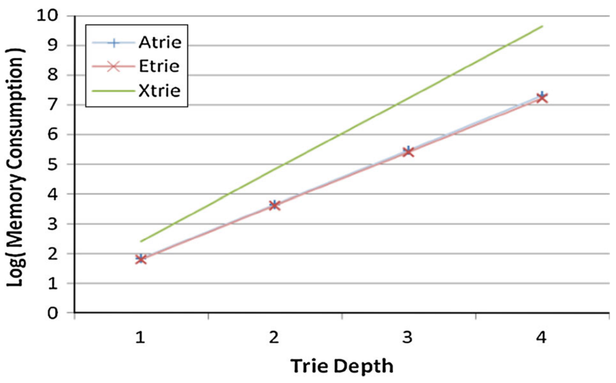

Fig. 4 Memory consumed by adaptive trie for case *sensitive* email addresses

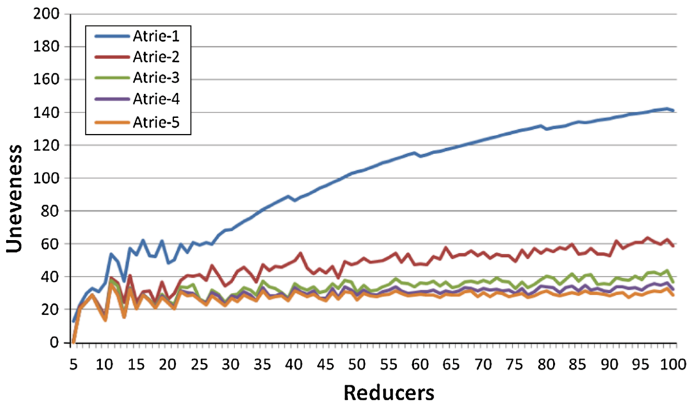

Fig. 5 Comparison of unevenness for an adaptive trie for case insensitive email addresses

Table 6 Space needed by Tries for case *sensitive* email addresses

Depth	Space needed by ATrie	Space needed by ETrie	Space needed by XTrie
1	67	64	256
2	4,489	4,096	65,536
3	300,763	262,144	16,777,216
4	20,451,884	16,777,216	4,294,967,296

Notice in Fig. 4, that both ATrie and ETrie exhibit similar performance regarding memory consumption. In fact the ATrie has slightly *worse* memory requirements than ETrie since email addresses contain characters other than letters and numbers (Table 6).

However, this is not the case if we assume Email addresses to be *case insensitive* and convert them all to lowercase. In such a situation, ATrie automatically adapts to use less memory as shown in Table 7 and shown in Fig. 6. The ATrie now needs about 7 times less space than it did previously.

Table 7 Space needed by Tries for case *insensitive* email addresses

Depth	Space needed by ATrie	Space needed by ETrie	Space needed by XTrie
1	41	64	256
2	1,681	4,096	65,536
3	68,921	262,144	16,777,216
4	2,894,682	16,777,216	4,294,967,296

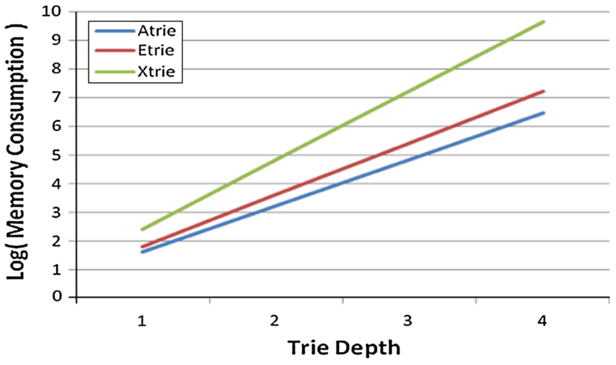

Fig. 6 Memory consumed by adaptive trie for case *insensitive* email addresses

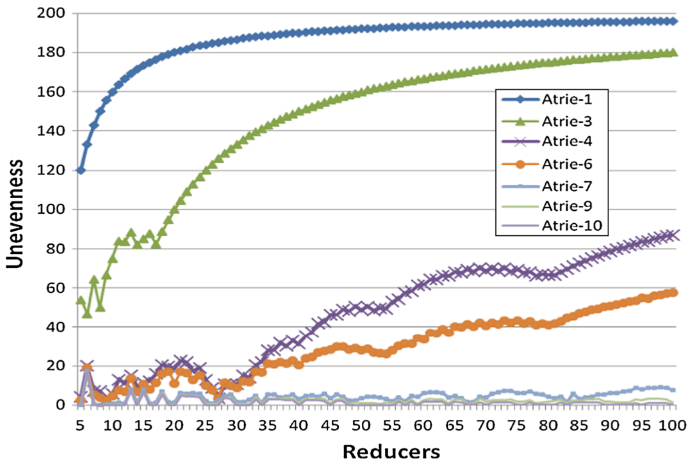

Fig. 7 Comparison of unevenness for an adaptive trie for 1,000,000 birthdates

Figure 7 is a comparison of the unevenness amongst reducers when partitioning a set of 1,000,000 birthdates. As expected as the number of reducers increase the unevenness increases. Furthermore, as n increases for an n-level ATrie, the unevenness decreases, however the unevenness is much more pronounced as compared to prior test cases. The reason for the unevenness is due to the related data format. Note, birthdates in the source file are all recorded using the same format YYYY-MM-DD. Furthermore, since

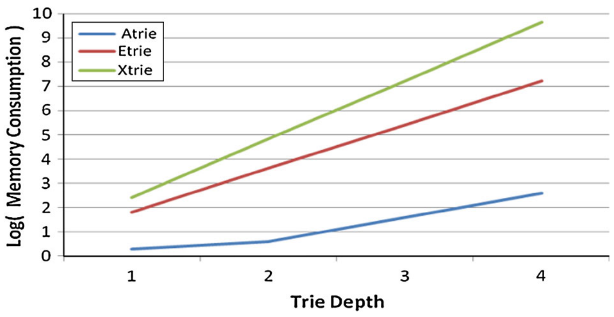

Fig. 8 Memory consumed by adaptive trie for 1,000,000 birthdates

Table 8 Space needed by Tries for Birthdates

Depth	Space needed by ATrie	Space needed by ETrie	Space needed by XTrie
1	2	64	256
2	4	64^2	256^2
3	40	64^3	256^3
4	400	64^4	256^4
5	400	64^5	256^5
6	800	64^6	256^6
7	8,000	64^7	256^7
8	8,000	64^8	256^8
9	32,000	64^9	256^9
10	320,000	64^{10}	256^{10}

these are birthdates all values in the source file span from 1920-01-01 to 2013-04-01. When using 1-level trie only the first character represented are “1” and “2”. Dates from 1920 to 1999 are therefore all sent to the same reducer, while dates from 2000 to 2013 are sent to another reducer. However, once enough characters are recognized by the trie the problem disappears. As shown in Fig. 7, unevenness decreases once it is parsed by a 7-level trie. However, as shown in the table below, this is impossible level for Etrie or Xtrie to handle as the space requirements become prohibitive, as shown in Fig. 8 and Table 8.

Figure 9, shows experimental results comparing the proposed trie based approach versus the approach used by TeraSort. The physical machine used in this experiment had 2 Intel Xeon CPUs running at 2.93Ghz, with 6 cores per CPU. An 11GB text file was used as input. Experimental results showed that as the number of levels in the trie increased the amount of unevenness decreased, and consequently the time taken decreased. The time taken for the 2-level trie approach and the traditional TeraSort approach were the same due to them both producing an identical set of cut points.

This paper presents an adaptive sampling mechanism for total order partitioning, which is used for sorting in this paper. Although, TeraSort is a standard benchmark for sorting, it only uses 100,000 samples, which is insufficient number of samples

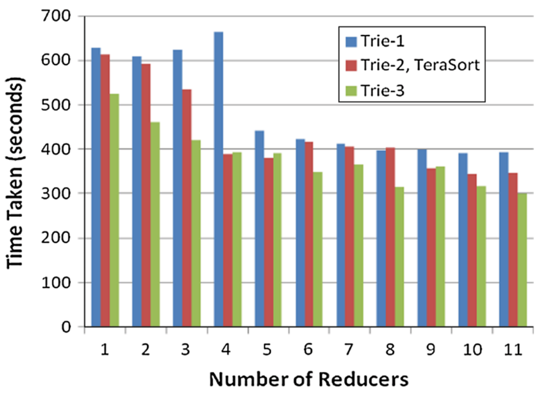

Fig. 9 Response time comparison between trie based methods versus traditional Terasort

for accurate partitioning [18]. Furthermore, comparing the time to partition randomly generated output, especially if the output is evenly distributed, is likely to be inconclusive. For these reasons, a response time comparison using TeraSort data has been omitted.

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 [9] 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. 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 [13,23]. 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

[2] 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 improve load balancing and provide better performance when sampling and partitioning XTrie and ETrie partitioning algorithms were introduced [22]. These algorithms used a fixed amount of memory, and were more efficient than the method employed by TeraSort. However, the method had some drawbacks as it needed a large memory footprint if it was to handle strings using a common prefix, a problem this paper attempts to address.

In order to work around perceived limitations of the MapReduce model various extend or change the MapReduce model have been presented. BigTable [3] 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 [8] which mimics the same functionality of BigTable.

Due to its simplicity of use, the MapReduce model is quite popular and has several implementations [14,16,17]. 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 [20,28], graph mining [10], text analysis [25] or genetic algorithms [11,26] 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) [5].

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 [7]. In [7] 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 [15] 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 amongst 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

In this paper, we have presented an Adaptive Trie that is capable of reducing memory consumption as well as automatically adapt for strings that have set formats and prefixes. Furthermore, this paper presented two ways to build and Adaptive Trie, either automatically by reading the source data with a preliminary pass, or manually by utilizing a simplified regular expression parser. The proposed Adaptive Trie was also compared against XTrie and ETrie, and it is shown how the proposed method can be used to build tries with greater number of levels given a fixed memory constraint. In future work we plan to evaluate and enhance our proposed methods for heterogeneous environments. Additionally, we would like to extend this work so that it can handle data types other than strings.

Acknowledgments 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. Apache Software Foundation, "Hadoop", http://hadoop.apache.org/core
2. Candan, K., Kim, J.W., Nagarkar, P., Nagendra, M., Yu, R.: RanKloud: scalable multimedia data processing in server clusters. IEEE MultiMed. **18**, 64–77 (2011)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: 7th USENIX Symposium on Operating Systems Design and Implementation, pp. 205–218 (2006)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
5. Fadika, Z., Govindaraju, M.: Delma: dynamically elastic mapreduce framework for cpu-intensive applications. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid, Computing, pp.454–463 (2011)
6. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: ACM SIGOPS Operating Systems Review, ACM, pp. 29–43 (2003)
7. Groot, S., Kitsuregawa, M.: Jumbo: Beyond MapReduce for Workload Balancing. VLDB, Phd Workshop (2010)
8. HBase, http://hadoop.apache.org/hbase/
9. Heinz, S., Zobel, J., Williams, H.: Burst tries: a fast, efficient data structure for string keys. Trans. Inf. Syst. (TOIS) **20**(12), 192–223 (2002)
10. Jiang, W., Agrawal, G.: Ex-mate: data intensive computing with large reduction objects and its application to graph mining. In: Cluster, Cloud and Grid Computing (CCGrid): 11th IEEE/ACM International Symposium on, IEEE 2011, pp. 475–484 (2011)
11. Jin, C., Vecchiola, C., Buyya, R.: MRPGA: an extension of MapReduce for parallelizing genetic algorithms. In: IEEE Fourth International Conference on eScience pp. 214–221 (2008)
12. Kavulya, S., Tan, J., Gandhi, R., Narasimhan, P.: An analysis of traces from a production mapreduce cluster. In: Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference, pp. 94–103 (2010)
13. Krishnan, A.: GridBLAST: a globus-based high-throughput implementation of BLAST in a Grid computing framework. Concurr. Comput. Pract. Exp. **17**(13), 1607–1623 (2005)
14. Liu, H., Orban, D.: Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. In: 11th IEEE/ACM International Symposium, pp. 464–474 (2011)
15. Lynden, S., Tanimura, Y., Kojima, I., Matono, A.: Dynamic data redistribution for MapReduce joins. In: IEEE International Conference on Coud Computing Technology and Science, pp. 713–717 (2011)

16. Matsunaga, A., Tsugawa, M., Fortes, J.: "Programming abstractions for data intensive computing on clouds and grids. In: IEEE Fourth International Conference on eScience, pp. 489–493 (2008)
17. Miceli, C., Miceli, M., Jha, S., Kaiser, H., Merzky, A.: Programming abstractions for data intensive computing on clouds and grids. In: IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 480–483 (2009)
18. O'Malley, O.: TeraByte Sort on Apache Hadoop (2008)
19. Panda, B., Riedewald, M., Fink, D.: The model-summary problem and a solution for trees. In: Data Engineering, International Conference on Data, Engineering, pp. 452–455 (2010)
20. Papadimitriou, S., Sun, J.: Disco: distributed co-clustering with map-reduce: a case study towards petabyte-scale end-to-end mining. In: IEEE International Conference on Data Mining, p. 519 (2008)
21. Shafer, J., Rixner, S., Cox, A.L.: The Hadoop distributed filesystem: balancing portability and performance. In: IEEE International Symposium on Performance Analysis of System and Software(ISPASS), p. 123 (2010)
22. Slagter, K., Hsu, C.-H., Chung, Y.-C., Zhang, D.: An improved partitioning mechanism for optimizing massive data analysis using MapReduce. J. Supercomput. **66**(1), 539–555 (2013)
23. Stockinger, H., Pagni, M., Cerutti, L., Falquet, L.: Grid approach to embarrassingly parallel CPU-intensive bioinformatics problems. In: IEEE International Conference on e-Science and Grid Computing (2006)
24. Tan, J., Pan, X., Kavulya, S., Gandhi, R., Narasimhan, P.: Mochi: visual log-analysis based tools for debugging Hadoop. In: USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (2009)
25. Vashishtha, H., Smit, M., Stroulia, E.: Moving text analysis tools to the cloud. In: IEEE World Congress on Services, pp. 110–112 (2010)
26. Verma, A., Llora, X., Goldberg, D.E., Campbell, R.H.: Scaling genetic algorithms using mapreduce. In: Intelligent Systems Design and Applications (2009)
27. White, T.: "Hadoop the definitive guide 2nd edition", Published Oreilly (2010)
28. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009)