International Journal of Biomedical Data Mining
ISSN: 2090-4924

Yang and Ngu, Int J Biomed Data Min 2016, 6:1
DOI: 10.4172/2090-4924.1000125

Research Article Open Access

Implementation of Decision Tree Using Hadoop Map Reduce

Tianyi Yang[1]* and Anne Hee Hiong Ngu[2]

[1]Texas Center for Integrative Environmental Medicine, Texas, USA
[2]Department of Computer Science, Texas State University

Abstract

Hadoop is one of the most popular general-purpose computing platforms for the distributed processing of big data. HDFS is an implementation of distributed file system by Hadoop to be able to store huge amount of data in a reliable way and process it in an efficient manner at the same time. MapReduce is the main processing engine of Hadoop. In this study, we have implemented HDFS and MapReduce for a well-known learning algorithm—decision tree in a scalable fashion to large input problem size. Computational performance with node count and problem size is evaluated.

Keywords: Computing; Processing; Data; Algorithm; Network

Introduction

MapReduce, as Hadoop project's principal processing engine, provides a framework for scalable distributive computing [1,2]. The MapReduce model is derived from the combinations of the map and reduce concepts in functional programming. A strong characteristic of this programming model is that it hides the complexity of dealing with a cluster of distributive computing nodes. Hence, the developers only need to focus on the implementation of map and reduce functions.

Decision tree is one of the most popular methods for classification [3]. A classical decision tree is a directed tree comprised of a root node, as well as decision nodes—all the other nodes each with exactly one incoming edge.

The procedure of building a decision tree is as follows. Given a set of training data, find the best splitting attribute from currently all available ones by applying a measure function on all attributes. Once the splitting attribute is determined, the instance is split into multiple partitions. The multiplicity depends on the number of values or ranges of values associated with the splitting attribute. Within each partition, if all samples belong to a single class, the algorithm terminates. Otherwise, the splitting process is recursively performed until each partition belongs to a single class, or no attribute is left. Once a decision tree is generated, classification rules are generated, which can be applied to classify new instances with class labels to be determined.

C4.5 is a one of the standard algorithm for decision tree, which uses information gain ratio as the splitting criterion. The algorithm is illustrated in Figure 1.

```
Input: training dataset $T$; attributes $S$.
Output: decision tree $Tree$.
 1: if $T$ is NULL then
 2:    return failure
 3: end if
 4: if $S$ is NULL then
 5:    return $Tree$ as a single node with most frequent class label in $T$
 6: end if
 7: if $|S| = 1$ then
 8:    return $Tree$ as a single node $S$
 9: end if
10: set $Tree = \{\}$
11: for $a \in S$ do
12:    set $Info(a,T) = 0$, and $SplitInfo(a,T) = 0$
13:    compute $Entropy(a)$
14:    for $v \in values(a,T)$ do
15:       set $T_{a,v}$ as the subset of $T$ with attribute $a = v$
16:       $Info(a,T) += \frac{|T_{a,v}|}{|T_a|} Entropy(a_v)$
17:       $SplitInfo(a,T) += -\frac{|T_{a,v}|}{|T_a|} log \frac{|T_{a,v}|}{|T_a|}$
18:    end for
19:    $Gain(a,T) = Entropy(a) - Info(a,T)$
20:    $GainRatio(a,T) = \frac{Gain(a,T)}{SplitInfo(a,T)}$
21: end for
22: set $a_{best} = \underset{a}{argmax}\{GainRatio(a,T)\}$
23: attach $a_{best}$ into $Tree$
```

Figure 1: C4.5 algorithm.

In the algorithm above,

$$Entropy(s) = -\sum_{j=1}^{C} p(S,j) \times (S,j)$$

is the ratio of instances in S which has the jth class label, and C denotes the total number of classes.

$$Info(S,T) = -\sum_{v \in vlaue(T_s)} \frac{|T_{S,v}|}{|T_S|} Entropy(S_v)$$

is the information needed after splitting by attribute S, in which TS is a subset of T induced by attributes S, and TS, v is a subset of TS of value v for attribute S, and Values(TS) is the set of values for attribute S for records in TS. Absolute value operator means cardinality of.

Information gain is defined as:

$$Gain(S,T) = Entropy(S) - Info(S,T),$$

***Corresponding author:** Tianyi Yang, Texas Center for Integrative Environmental Medicine, Texas, USA, Tel: +1 512-245-2111; E-mail: cosmosischaos@gmail.com

Received December 21, 2016; **Accepted** February 04, 2017; **Published** February 11, 2017

Citation: Yang T, Ngu AHH (2017) Implementation of Decision Tree Using Hadoop MapReduce. Int J Biomed Data Min 6: 125. doi: 10.4172/2090-4924.1000125

which measures the information gain after splitting by attribute S.

Split information

$$split\ Info(S,T) = -\sum_{v\in vlaue(T_s)} \frac{|T_{S,v}|}{|T_S|}\times\log\frac{|T_{S,v}|}{|T_S|}$$

Finally, the information gain ratio is:

$$Gain\ Ratio(S,T) = \frac{Gain(S,T)}{SplitInfo(S,T)}$$

which is the criterion to split the decision tree. The attribute which gives the maximum GainRatio is selected as the splitting attribute.

Three requirements with decreasing priority have been enforced in our implementation of decision tree using MapReduce and HDFS. 1) Scalability to input data. This voids any assumption that the memory of the master node can hold the data. Therefore, an iteration of launching of MapReduce jobs is required. 2) Minimize the use of mapper/reducer unless necessary. We have found that each launching of MapReudce job takes tens of seconds before it starts executing mapper method. So if the runtime of the section of the program outside MapReduce is less than this launching time, it is not worth implementing an additional set of Mapper and Reducer for any part in corresponding section. 3) Minimize HDFS file I/O in case computation can fulfil the same purpose. Local computation is much faster than typical hard drive I/O, and even faster than network storage I/O which is limited by network data transmission speed. So based on these three requirements, one set of mapper and reducer functions are implemented to read data from the source file and generate the tuple (attribute id, attribute value, class label, count).

Numerical Experiments and Results

The goal of this study is to evaluate the performance of Hadoop implementation of decision tree. In particular, the compute time vs. the count of processors that perform the mapper/reducer function, and the compute time vs. input data size are examined.

The infrastructure used in this study is AWS (Amazon Web Service), in which three specific services are utilized, i.e. 1) S3, i.e. Simple Storage Service, to which the compiled .jar file and input files are transferred from local computer. S3 further sends the above files to the virtual cluster that will be allocated by EC2; 2) EC2, i.e. Elastic Compute Cloud, which creates a virtual cluster to users' need; and 3) EMR, i.e. Elastic MapReduce, which runs MapReduce jobs in the virtual cluster.

To simulate typical working environment in which Hadoop's MapReduce projects are running, our rules for the specification of the virtual cluster are as follows: 1) One compute (performing mapper and reducer functions) processor per node. The philosophy behind this is the employment of homogeneous hardware layout for scaling behavior study. Recently, multi-processor compute nodes and multi-core processors have become main stream. However, data transfer in a single node is much faster than the cross node communication. To make a fair comparison for the scaling behavior of runtime vs. compute processors count, single compute core node is chosen; 2) Each chosen processor has the same medium computing power; 3) The network transfer capability among nodes is chosen to be medium. Requirements 2 and 3 are needed to mimic a typical Hadoop running environment. Based on the above rules, AWS m1.medium architecture is selected for both master node and core node. The master node assigns Hadoop tasks to core nodes and monitors their status, but does not participate in processing the Hadoop tasks. The core nodes run Hadoop tasks and store data in the HDFS system. In our experiment, the count of master node processor is always 1. And the count of core processors varies from 1 to 8. The Hadoop distribution version is Amazon 2.4.0. All the default configuration for this version of Hadoop setup is adopted, e.g. 3 copies of HDFS file duplication and 64 MB block size.

Table-1 shows the experimental results, in which, "ML" denotes million-lines and "MB" means megabytes. The 6-digit numbers show run time in unit millisecond. The run time is counted from the launching of main function to the completion of decision tree generation.

Figure 2 shows the runtime vs. number of processors. Figure 3 shows the speedup vs. number of processors. Speedup is defined as the serial runtime, i.e. runtime for number of processors equal to 1, divided by parallel runtime for corresponding number of processors. For relatively small problem sizes, i.e. 1 million lines and 2 million lines input data, Hadoop distributed computing does not improve performance at all. On the contrary, it hurts the performance. For 4 million lines input, the speedup goes up from 1 processor to 2 processors, continued with ignorable increase from 2 processors to 4, and then goes down from 4 to 8. The 8 million lines' curve shows consistent speed up from 1 to 8 processors, although the increase rate slows down from 4 to 8 processors. This behavior can be explained using Amdahl's law [4].

As is shown in Figure 4, in ideal situation (green curve), which can never be achieved in reality, speedup is linear to the number of processors. And in the idealist case, the slope for speedup vs. # of processors is 1. Amdahl's law states that for a parallel program, if the ratio of parallelizable section occupies a fraction f of total run time, the speedup is upper bounded by:

$$\frac{1}{1-f}$$

The proof is as follows: T(n) = Ts (n) + Tp (n)

In which T is runtime, n refers to the processor count, and subscript "S" or "P" denotes serial or parallel runtime, respectively. The special case where n=1 and

Tp(1) = f (1)

Serial code runtime has no difference in serial or parallel execution, i.e.,

Ts(n) = Ts (1) = (1- f) T (1)

Apply the ideal case to

we have

$$Tp(n) \geq \frac{T_p(1)}{n} = \frac{fT(1)}{n}$$

Substitute the expressions of Ts (n) and Tp (n) related to

T (1)

We have

$$T(n) \geq (1-f)T(1) + \frac{f}{n}T(1)$$

Finally

$$S(n) = \frac{T(1)}{T(n)} \leq \frac{1}{1-f+f/n}$$

So when # of processors n approaches infinity, S(n) increases and asymptotically approaches $\frac{1}{1-f}$ t

# of processors →	1	2	4	8
Input size (ML/MB) ↓				
1/25.4	518662	517247	555759	596748
2/50.9	532093	535297	538179	590010
4/101.9	601687	559425	556816	608845
8/203.8	736162	656622	600464	567313

Table 1: Computing times for different file sizes and numbers of processors.

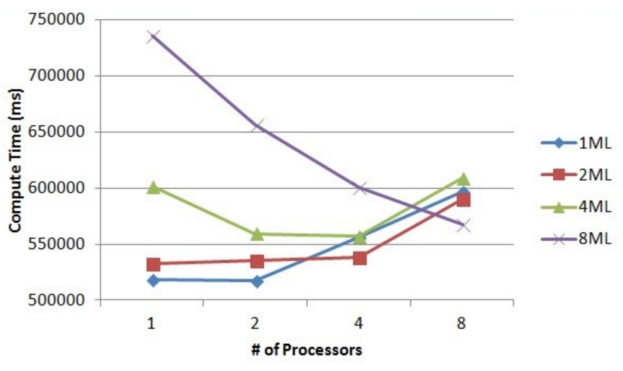

Figure 2: Runtime vs. processor count.

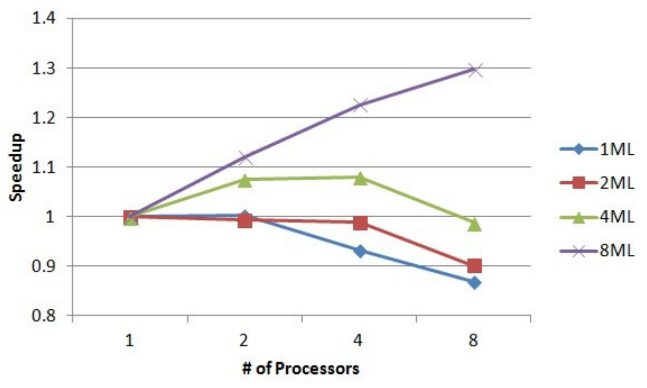

Figure 3: Speedup vs. processor count.

This is basically what the horizontal dashed curve shows in Figure 5. There exist multiple issues that can decrease the speedup for large number of processors. For example, f may not be a constant but decreases due to increased communications, imbalanced work load, limited bus speed, limited memory access rate, etc. In addition, the turning point (as shown in red curve) is not a constant. It typically moves to the right when the problem size goes up. This indicates that the smaller the input size, the earlier the saturated or even deteriorated speedup. This explains why our curves in Figure 3 with input size 1 and 2 million lines goes down earlier (from # of processors 1) than input size 4 million lines (from # of processors 4), and the latter turns earlier than input size 8 million which has not shown the peak for 8 processors, but already shows decreased slope. In our case, the main causes that make small inputs' speedup decrease and the largest input have very small speedup (no greater than 1.3) are probably due to the followings: 1) The overhead of launching MapReduce jobs. As is observed from the Hadoop's output, it takes tens of seconds each time a MapReduce job is launched before it starts executing the mapper function; and 2) The communication overhead. All nodes work simultaneously on the data to perform mapper and reducer functions. Due to the storage nature of HDFS, input data needs to be transferred among data nodes through Ethernet; 3) Writing reducer output. After performing the reduction, it takes time to write 3 copies of output to HDFS.

If the number of processes is increased and the speedup increases linearly, under the constraint that problem size remains the same, the problem is called strongly scalable. Very few parallel problems fall into this category. On the other hand, if speedup increases linearly with the increase of the problem size, the problem is called weakly scalable. Apparently, our results show weak scalability.

Figure 5 shows runtime vs. input size for different # of processors. With our choice of hardware layout, # of processors to perform mapper and reducer functions is the same as # of data nodes. The default block size for HDFS is 64MB. In addition, each block has 3 duplications in total. Therefore, it is normal that for processor count of 1 and 2, the runtime keeps increasing, since each data node contains full copy of all

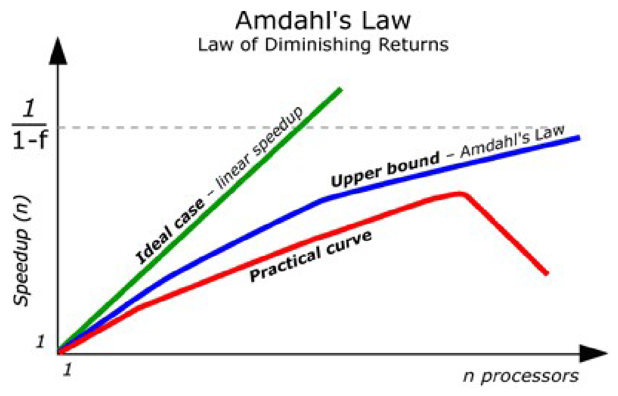
Amdahl's Law
Law of Diminishing Returns
1/(1-f)
Speedup (n)
1
1
n processors
Ideal case – linear speedup
Upper bound – Amdahl's Law
Practical curve

Figure 4: Amdal's law illustration.

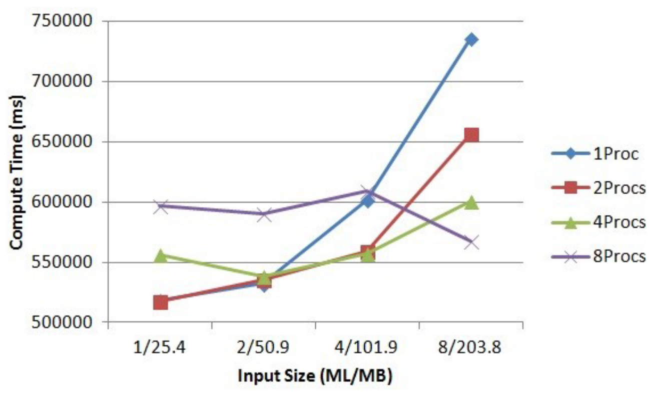

Figure 5: Runtime vs. input size.

data. However, with processor count of 3 and 4, for small data size, only part of the nodes have data. This means data needs to be transferred across network. For example, for input size of 1 million lines, only 1 block and 3 duplications exist. If # of nodes is 8, 3 nodes need to transfer data to the rest of the 5 nodes. While for data size of 8 million lines, i.e. 203.8 MB, 4 data blocks exist, and up to 12 nodes can have at least 1 block if distributed evenly. This indicates that 8 data nodes can all have 1 block of data, which makes data transfer across network unnecessary. So, this is the reason for a relative big speedup from 4 ML to 8 ML in the purple curve.

Conclusions and Future Work

Can we conclude that a decision tree algorithm benefits from the Map Reduce implementation from our experimental study? Probably we cannot. The major reasons are discussed in the following paragraphs.

Decision tree is an irregular program, which means the performance is dependent on multiple input factors. For example, a) the number of attributes. We have studied the input size's impact. However, size is only one of the controlling factors. Even if the lines of input are the same, each line can contain different number of attributes, which can give rise to different complexity in the generation of the decision tree; b) the number of class labels; c) the number of values for each attribute. A decision tree of averaged attribute value of 100 has a different complexity when compared with one of averaged attribute value of 2; d) the input size, which we have examined; e) the input content, which means the combination of attribute' values and class labels. Even if factors listed in a) through d) are identical, the different combination can lead to different tree complexity. In our study, in order to make a fair comparison, all the factors are kept identical on purpose except the input file size, which means the complexity of the constructed decision tree is all the same and only the variation of input size causes the difference of runtime in MapReduce program section.

In addition, we can state that the runtime difference in comparison is caused by MapReduce. The state of the program before each execution of Mapper is the same. The output of the Reducer differs in the values of the last column in the same iteration, which are proportional to file size. The complexity of computation performed on the output of the Reducer in the same iteration is identical. Hence, the runtime counted reflects only the difference in MapReduce execution.

There are other factors that can impact the results. For example, the selection of hardware layout can affect the performance. The shared-memory system of 8 processors on a single node is expected to perform better than 8 single-core nodes. Because the former uses all local data, while the latter involves network data transfer in general. On the other hand, Hadoop settings, e.g. data block size and duplication count, can also impact the performance.

In ideal situation, is it possible to achieve linear speedup in our implementation of decision tree? We can try to make the following efforts. First, we need to require the numbers of attributes and label count, the number of each attribute's values bounded. This indicates that the total number of unique instances is finite. Also, overheads like invocation of MapReduce jobs and hardware bottlenecks are also bounded. Second, network data transfer speed is sufficiently fast. "Sufficiently" here means in line with hard drive data transfer speed, for both downstream/read and upstream/write. This indicates all network data transfer works at the corresponding local hard drive speed. Therefore, there exists no additional network communication latency due to imbalanced work load. Third, the number of instances in the input file approaches infinity. This indicates that parallel execution fraction can approach 100%. However, for any finite problem size, linear speedup or quasi-linear speedup can continue to a certain number of node count. The reason is that MapReduce jobs are only a portion of the entire program. Even if the mapper and the reducer functions are perfectly parallelized, the rest of the system is still a serial program. The Amdahl's law tells us that speedup is upper-bounded by 1/(1-f), where f is the parallelizable fraction of the program. Even if 95% of a program can be perfectly parallelized, the speedup cannot surpass 20.

In conclusion, we have implemented a decision tree algorithm using Hadoop MapReduce computing engine and HDFS file system. We have observed that invocation of a MapReduce job can take substantiate amount of time. For our selection of hardware setup and input data, larger input data size shows some performance gain with increased number of processors/nodes. Our decision tree algorithm demonstrates weak scalability behavior. However, it is hard to draw a general conclusion on whether decision tree will benefit from MapReduce implementation due to the irregularity nature of decision tree problem.

Future study can focus on evaluating performance of more diversified inputs, e.g. different numbers of attributes, class labels, different number of attributes' values, and randomized instances, and generating performance statistical distributions for interested parameters. In addition, study of different hardware layout, e.g. multi-processor nodes, multi-core processors in shared memory system, systems including both shared memory and distributed memory layouts, may be of interest. Further, impact of change of default Hadoop settings, e.g. block size and duplication number in HDFS can be modified. Finally, effect of adding an additional mapper/reducer to compute gain ratio can be evaluated.

Acknowledgement

This work was performed while Tianyi Yang was a student at Texas State University

References

1. Dean J, Ghemawat S (2004) MapReduce: Simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, San Francisco, CA. pp: 137-149.
2. Shoberg J (2006) Building search applications with Lucene and Nutch (1st edn). Apress. p. 350.
3. Rokach L, Maimon O (2014) Data mining with decision trees: Theory and applications (2nd edn). World Scientific Publishing Company.
4. Amdahl, Gene M (1967) Validity of the Single processor approach to achieving large-scale computing capabilities. AFIPS Conference Proceedings 30: 483-485.