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
OMICS International

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.

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.

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

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.

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.

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

# 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.

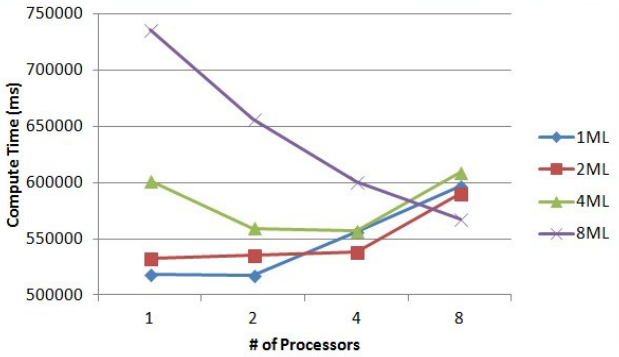

Figure 2: Runtime vs. processor count.

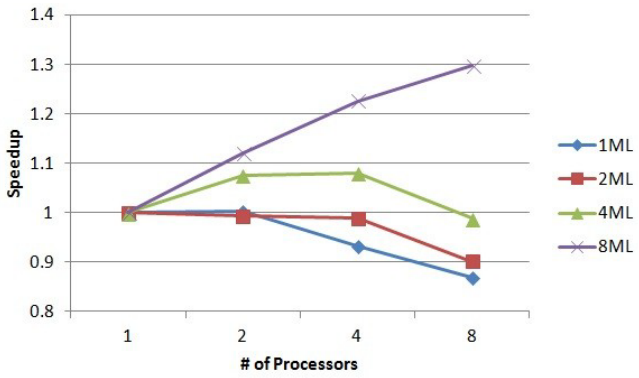

Figure 3: Speedup vs. processor count.

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

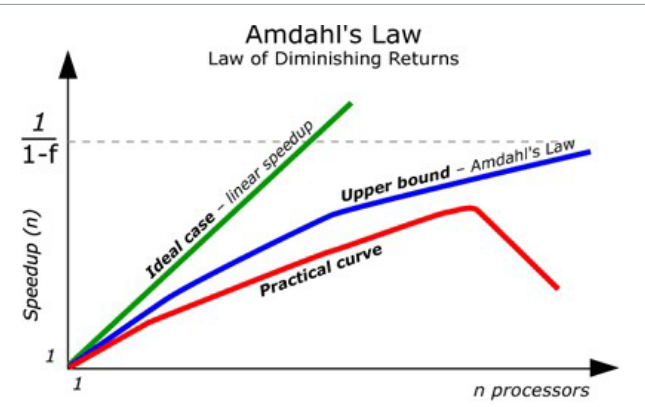

Figure 4: Amdal's law illustration.

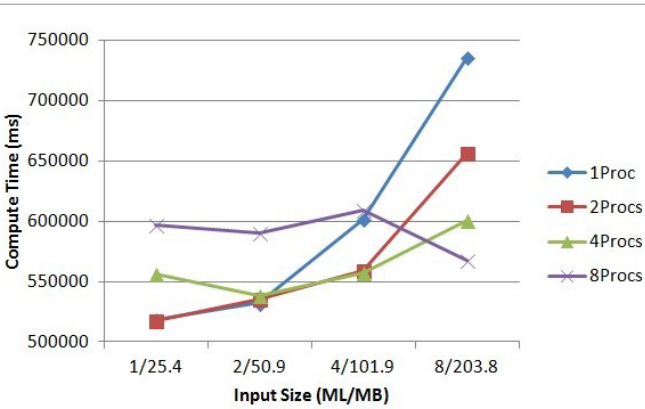

Figure 5: Runtime vs. input size.