Abstract

For so-called “big data”, the time taken to process a data mining algorithm is a critical issue. Many very good algorithms are unusable in the big data environment due to the fact that the processing takes an unacceptable amount of time. Therefore, speed of processing is very important. To address the speed issue, in this paper, we use horizontal processing of vertically structured data rather than the ubiquitous vertical (scan) processing of horizontal (record) data. We use pTrees, bit level, vertical data structures. pTrees technology represents and processes data differently from the ubiquitous horizontal data technologies. pTree technology, the data is structured column-wise (into bit slices) and the columns are processed horizontally (typically a few to a few hundred bit level columns), while in horizontal technologies, data is structured row-wise and these rows are processed vertically (often down millions, even billions of rows). P-hashes are lossless, compressed and data-mining ready data structures. pTrees are lossless because the vertical bit-wise partitioning that is used in the pTree technology guarantees that all information is retained completely. There is no loss of information in converting horizontal data to this vertical format. P-hashes are compressed because in this technology, segments of bit sequences which are either purely 1-bits or purely 0-bits, are represented by a single bit. This compression saves a considerable amount of space, but more importantly facilitates faster processing. pTrees are data-mining ready because the fast, horizontal data processing involves can be done without the need to decompress the structures first. pTree vertical data structures have been exploited in various domains and data mining algorithms, ranging from classification, clustering, association rule mining, as well as other data mining algorithms. Speed improvements are very important in data mining because many quite accurate algorithms require an unacceptable amount of processing time to complete, even with today’s powerful computing systems and efficient software platforms. In this paper, we evaluate and compare the speed of various data mining algorithms when using pTree technology.

Introduction

Predicate tree or pTree is the vertical data representation that represents the data column-by-column rather than row-by-row (which is relational data representation). It was initially developed for mining spatial data [2][4]. Since then it has been used for mining many other types of data [3][5]. The creation of pTree is typically started by converting a relational table of horizontal records to a set of vertical, compressed pTrees by decomposing each attribute in the table into separate bit vectors (e.g., one for each bit position of a numeric attribute or one bitmap for each category in a categorical attribute). Such vertical partitioning guarantees that the information is not lost.

For example, let R be a relational table consists of three numeric attributes R(A1, A2, A3). To convert it into pTree we have to convert the attribute values into binary and then take vertical bit-slices of every attribute and store them in separate files. Each bit slice is considered as a pTree, which indicates the predicate of a particular bit position is zero or one. This bit slice may be compressed dividing it into binary trees recursively. Figure 1 depicts the conversion of a numerical attribute, A1, into pTrees.

![Diagram of pTree construction](image)

**Figure 1.** Construction of pTrees from attribute A1. The pTrees are built from the top down, stopping any branch as soon as purity (either purely 1-bits or purely 0-bits) is reached.

Space advantage of pTree

I assume a dataset S consisting of N rows and m columns containing value of m bits as shown in figure 2a.

![Diagram of dataset S](image)

**Figure 2:** The data set S

So if we convert the dataset into pTrees we will get m pTrees where the length of each tree will be N bits.

In traditional approach, let the size of the dataset be S, which we can calculate as follows:

\[ \text{Size of each value} = \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \text{ bytes} \]

\[ \text{So size of each column} = \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \text{ bytes} \]

\[ \text{So the size of the whole dataset, } S_{\text{total}} = \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \times m \text{ bytes.} \]

In pTree approach, let the size of the size of the dataset be S_{pTree}:

\[ \text{Size of each pTree} = \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \text{ bytes.} \]

\[ \text{So the size of the whole dataset, } S_{\text{pTree}} = m \times \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \text{ bytes.} \]

In a large dataset where N >> 8, we can assume N = 8L

We get S_{\text{pTree}} = 8L \times \log_2 m \times \text{bytes and } S_{\text{total}} = \text{md} \times \text{bytes.}

Now if \( \text{m} < \text{L} \) use

\[ S_{\text{pTree}} = \frac{\text{m}}{8} \times \text{S}_{\text{total}} \]

When \( m = 8 \)

\[ S_{\text{pTree}} = S_{\text{total}} \]

Again,

Now if \( \text{m} < \text{L} \times \text{m} \times \text{S}_{\text{total}} \]

When \( m = 16 \)

\[ S_{\text{pTree}} = S_{\text{total}} \]

So we conclude that, \( S_{\text{pTree}} < S_{\text{total}} \) and \( S_{\text{pTree}} = S_{\text{total}} \) if \( m \) is a multiple of 8.

Speed advantage of pTree

Assume that a logical operation (AND, OR, NOT, XOR) between two machine words (of size W) of memory takes \( T_{\text{M}} \) units of time. When we do such a logical operations on two pTrees of length \( N \) we actually do it on \( L \) pairs of machine words where \( L = \left\lceil \frac{\log_2 N}{\log_2 m} \right\rceil \). So the logical operation on two pTrees takes \( LT_{\text{M}} \) units of time.

Assume an arithmetic operation (addition, subtraction, multiplication, division) between two bytes of memory takes \( T_{\text{M}} \) units of time.

Suppose we will do such an arithmetic operation on two columns of our previously described data set, S. For simplicity assume each value in the dataset takes 1 byte of memory.

So each column has \( N \) bytes memory and the arithmetic operation will take \( T_{\text{M}} = NT_{\text{M}} \) units of time.

Reference:


