Radix Sorting Algorithm in Big Data Environment

Shunzeng SHAO
Changzhou Institute of Engineering Technology, Changzhou Jiangsu 213164, China

Abstract: Sorting and searching become increasingly important with the development of information technology and, especially, the application of big data and cloud computation. Aiming at the low space efficiency of radix sorting, a no-bucket and grouped radix sorting algorithm is proposed to sort big data, which, from the aspects of time and space, has raised the efficiency of the classical algorithm. This article gives the principle, simple steps and analysis of the algorithm. As proved in our test, it is a stable and efficient sorting algorithm as it can greatly cut the storage space and improve the time efficiency as well, with the speed comparative that of the quick sorting algorithm.

Keywords: Sorting; Radix sorting; Space Complexity; Time Complexity

1. INTRODUCTION

The trend of big data and cloud computing is unstoppable. We are now facing a new order of magnitude arising along with big data: the size of business data increases from TB to PB, while for personal data, the size goes up from GB to TB. As a result, volume of the data processed in computer is growing larger and larger, and query sort of data is also becoming more important. People have been searching sorting algorithms of higher efficiency, and many efforts have been done around the radix sorting algorithm. For instance, the radix sorting algorithm was combined with other algorithms to raise both the time and the space efficiency by taking advantages of each algorithm. Or, the efficiency was improved by changing the radix or finding a more reasonable one. However, there are now reports on direct usage of the principle of radix sorting to raise the time and the space efficiency. By using the principle of radix sorting, this study has improved the classical algorithm, which increases greatly both the time efficiency and the space efficiency.

2. RADIX SORTING

Radix sorting, also called as “bucket sorting”, is a kind of distribution sort. As the name implies, it works by distributing the elements that need to be sorted into a number of “buckets” to make sort.

2.1. Principle

Shorter digits are padded on the left with zero to make the data (positive integers) that need to be ordered have the same length. Then, the data are ordered in a sequence starting from the least (most) significant digit and moves towards the most (least) significant digit.

2.2. Steps

Here is an example of the least significant digit (LSD) radix sort in which the radix is 10.

(1) Find out the maximum from all the data;
(2) Count the number of the digits of the maximum, which is also the times of the sort;
(3) Put the data into 10 buckets, starting from the least significant digit (1s place);
(4) Visit the buckets in order and put all the data back into the data store;
(5) Turn back to step (3) to make the sort by the next digit (10s place). The sort ends till all digits are processed.

2.3. Features of radix sorting algorithm

(1) Time complexity is represented as O (k*n), where, k is the digits of the data. In some cases, a radix sort is more efficient than comparative sorting algorithms.
(2) If the allocation is based on a static storage, the space complexity can be represented as O (n*r), where, r is the radix which, in general, is 10 (r=10). This requires a large space of secondary storage, which makes it difficult being applied in computer.
(3) Radix sort is a kind of stable sort.

3. NO-BUCKETED AND GROUPED RADIX SORT

The no-bucket and grouped radix sorting algorithm is an improved version of the classical radix sorting algorithm from two aspects: buckets that consume large assist storage space are deleted, which raises the space efficiency on the one hand; all data are grouped according to the radix, and each group is sorted separately,
which gets rid of unnecessary processes and, therefore, raises the time efficiency on the other hand.

3.1. No-bucket radix sort

The biggest problem of the classical radix sorting algorithm is that it takes up large auxiliary storage space. So, it is not that applicable when used in a computer for it consumes a great deal of resources. For the no-bucket radix sort, the requirement for space can be greatly decreased, which is conducive to the application of computer to process data.

**Principle**

First of all, find out the maximum from all the data. Count the number of the data in each bucket according to a certain value of the data, and turn the numbers into the initial position of the data in each bucket in the original storage space. According to the order of the buckets, redistribute the data into the auxiliary storage space. Process each data in increasing or decreasing order, and the sort ends.

**Steps**

For a LSD radix sort of the data in an array a[n], as shown below, in which the radix is 10, here are the steps.

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>......</th>
<th>Nn-1</th>
<th>Nn</th>
</tr>
</thead>
</table>

(1) Find out the maximum from all the data Max;
(2) Count the digits of the maximum (loop);
(3) Define a “bucket” of integers (buckets[10]) and initialize the elements to 0;
(4) According the digit value tempNum;
(5) Compute the digit code row_index;
(6) Add 1 to elements in buckets with the subscript of row_index;
(7) Go back to step (5) if not all the data are processed;
(8) For the data in each bucket, compute their initial position in the storage space but still put them back into the buckets;
(9) Compute the digit code row_index;
(10) Move the data to the positions in the auxiliary storage space, where, the subscript is buckets[row_index], i.e. the current positions of the buckets with the subscript of row_index in the auxiliary storage space;
(11) Add 1 to elements in buckets[row_index], which means that positions of the data in buckets[row_index] move backward in the auxiliary storage space by 1;
(12) Go back to step (9) if not all the data are moved to the auxiliary storage space, otherwise, the sort of one digit ends and continue;
(13) Go back to step (3) if not all the digits are sorted, otherwise the sort ends.

In the above algorithm, the process of “bucketing” that consumes the most space is deleted from the classical algorithm. For the data in each bucket, their initial positions in the auxiliary storage space are determined after the number of the data in each bucket is counted. The data are then put directly in the auxiliary storage space that was used to store the original data, as a result, the algorithm needs only the auxiliary storage space.

The main procedure is as follows:

```c
int * p1;
if(p==a)// If the pointer p points to the original storage space;
{
    p1=temp;// p1 points to the auxiliary storage space;
}
else
{
    p1=a;// Otherwise, p1 points to the original storage space
}
/*Set up an array of buckets, with the preset value of 10*/
int buckets[10]={0};
int i , j=0,k=0,k1 ;

// Count the number of the data in each bucket
int row_index;// digit code
for( i = 0; i < n ; i++ )
{
    row_index = (*(p+i)/tempNum) % 10;
    buckets[row_index]++;
}
```
//Initial digits of the data in each bucket
k=buckets[0];
buckets[0]=0;
for(i=1;i<10;i++)
{
    k1=buckets[i];
buckets[i]=buckets[i-1]+k;
k=k1;
}

//Perform the sort, i.e., move the data back into the auxiliary storage space
for(i=0;i<n;i++)
{
    row_index= (*p+i) / tempNum) % 10;
p1[buckets[row_index]]=*(p+i);
buckets[row_index]++;
}
return p1;//Return store addresses of the sorted data

3.2. Grouped radix sort
In the classical radix sort, all data are deemed to have the same length and all data are processed in accordance with the longest one, which wastes a plenty of time. The waste will be more especially when there are only few longer data. To solve this problem, a grouped radix sorting algorithm is designed in this paper.

Principle
First of all, data are grouped into ones, tens and hundreds. Then, the no-bucket radix sorting algorithm is used to individually sort each group of the data.

Steps
(1) Define an array of digit values for each data and assign values to the array;
(2) Determine the digits of the data;
(3) If the data is a j-digit number, add 1 to the array element bitNumber[j-1] that stores the data;
(4) If j is larger than the largest digit (loopTimes) among the already checked data, save j in the loopTimes;
(5) Go back to step (2) if not all the data are processed, otherwise continue;
(6) Compute the initial positions of each group of the data (ones, tens and hundreds) through the loop of “one digit by ten circles”, and save the data in bitNumber;
(7) According to the initial data of each group of the data, put all the data into the auxiliary storage space, and the grouping of the data ends;
(8) Use the no-bucket radix sorting algorithm to sort individually each group of the data, then all the data can be sorted after the groups are sorted.

The data are grouped in accordance with their lengths, and groups of the data are then sorted individually. For a one-digit number, only one times of no-bucket radix sort is required, which saves the loopTimes by 1 times. For a two-digit number, only two times of no-bucket radix sort are required, which saves the loopTimes by 2 times. And so forth, the time of sorting is shortened.

The procedure code for grouping the data:
int i,j,k,k1,*p1=temp; //Note that the pointer shall be assigned with a value, otherwise it points nowhere
int t[10]; //A temporary array for storing the initial positions of the grouped data
int tpow[10]={0,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000}; //An array of digit values
for(i = 0 ; i < n; i++) //Count the number of the digits
{
    for(j=1;;j++) //Judge the loop of a digit
    {
        if((p[i]/tpow)==0) // j-digit number
        {
            bitNumber[j-1]++; // Add 1
            break;
        }
    }
    if(j>loopTimes)//loopTimes is not the maximum
loopTimes=j;
}

//Determine the initial positions of the data in each bucket
k=bitNumber[0];
bitNumber[0]=0;
for(i=1;i<loopTimes;i++)
{
    k1=bitNumber[i];
    bitNumber[i]=bitNumber[i-1]+k;
k=k1;
}

//Put the data of the bitNumber array into the t array
for(j=0;j<loopTimes;j++)
{
    t=bitNumber;
}

//Data grouping
for(i=0;i<n;i++)
{
    for(j=1;;j++)
    {
        if(p[i]/tpow==0)
        {
            p1[t[j-1]]=p[i];
t[j-1]++;
        break;
        }
    }
}
return p1;//Return the pointer of the grouped data

4. EFFICIENCY OF THE NO-BUCKET AND GROUPED RADIX SORTING ALGORITHM

4.1. The space efficiency

Table 1. Comparison to the classical radix sorting algorithm in the space aspect

<table>
<thead>
<tr>
<th>Classical radix sort</th>
<th>No-bucket and grouped radix sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>The auxiliary storage space for the buckets: Radix *n</td>
<td>The auxiliary storage space for storing the data: n</td>
</tr>
<tr>
<td>Radix</td>
<td>The auxiliary storage space for storing both the number and initial positions of the data in each bucket: Radix</td>
</tr>
</tbody>
</table>

It is clear from the above comparison that the no-bucket and grouped radix sorting algorithm can save the storage by a half, which greatly reduces the space required for the sort and, therefore, can be expediently applied in a computer

4.2. The time efficiency

The entire sort of the no-bucket and grouped radix sorting algorithm contains two major phases: first, the data are grouped according to the lengths; second, using the no-bucket radix sorting algorithm to sort each group of the data.

Assume that there are n data to sort; the radix is 10; the digit of the longest data is p; the number of the data of the same length is ni (∑ni=n, where, i=1,2,3,…,p), and; the digit of each data is determined after k (1<=k<=p) times.

The grouping process: count the number of the data of the same length, then put the counted data into the auxiliary storage space.

The time complexity in each phase is the same: O(n*k). So the total time complexity is O(2*n*k).

If the numbers of the data of different length are the same, k=(p+1)/2

the time complexity is: O(2*n*k)

=O(2*n*(p+1)/2)

=O(n*(p+1))
The process of the no-bucket radix sort: make the sort from the shortest data to the longest data. For one-digit number, one times of the no-bucket radix sort is required. For two-digit number, two times of the no-bucket radix sort are required. For p-digit number, p times of the no-bucket radix sort are required. Similar to that in the grouping phase, the time complexity of each no-bucket radix sort is O(2*ni). The total time complexity of the no-bucket radix sort is:

\[ O(2*n_1) + O(2*n_2*2) + \ldots + O(2*n_p*p) \]

\[ = O(2*n_1+2*n_2*2+\ldots+2*n_p*p) \]

given that \(n_1=n_2=\ldots=n_p=n/p\)

\[ = O(2^n/p(1+2+\ldots+p)) \]

\[ = O(n^p/(1+p)) \]

The total time complexity of the no-bucket and grouped radix sort is:

\[ O(n^*(p+1)) + O(n) \]

\[ = O(n^*(p+2)) \]

Because \(p\) represents the digit of the longest data, which, in general, is no larger than several tens, the total time complexity of the algorithm is \(O(n)\).

5. TEST RESULTS OF THE TIME EFFICIENCY OF THE NO-BUCKET AND GROUPED SORTING ALGORITHM

The test of the time efficiency of the no-bucket and grouped sorting algorithm is performed under the Windows 8, using the Dev-cpp5.4.0 for programming. The no-bucket sorting algorithm is used replace the classical one because it is difficult to use a computer to realize the classical radix sort.

5.1. Comparisons to bubble sorting algorithm and other algorithms

Figure 1 shows the time to sort 50000 randomly generated data required by each algorithm.

![Figure 1](image.png)

As can be known from the above test, for the 50000 randomly generated data, the time efficiency of our algorithm is the same to that of the quick sort and is higher than that of the bubble sort by nearly three orders of magnitude.

5.2. Comparisons of the time efficiencies of the no-bucket and grouped radix sorting algorithm to sort the data of different lengths
(1) Fig. 2 shows the sorts of 500000 randomly generated data.

![Figure 2](image1)

(2) Fig. 3 shows the sorts of 500000 randomly generated data, which include 100000 one-digit numbers, 100000 two-digit numbers, 100000 three-digit numbers, 100000 four-digit numbers and 100000 five-digit numbers, i.e., the data are equally distributed.

![Figure 3](image2)

(3) Fig. 4 shows the sorts of 500000 randomly generated data, which include 25000 one-digit numbers, 25000 two-digit numbers, 50000 three-digit numbers, 100000 four-digit numbers and 300000 five-digit numbers.
Fig. 4

Figure 4 shows the sorts of 500000 randomly generated data, which include 300000 one-digit numbers, 100000 two-digit numbers, 50000 three-digit numbers, 25000 four-digit numbers and 25000 five-digit numbers.

Fig. 5

Figure 5

As can be seen from the above tests, for general data sorting, the time efficiency of the no-bucket and grouped radix sort equivalent to that of the quick sort that have the best time efficiency. For a set of unequally distributed data that contains more shorter data and less longer data, the time efficiency of the no-bucket and grouped radix sorting algorithm becomes higher. In a word, the total efficiency can be improved by using our algorithm.

6. CONCLUSIONS

The auxiliary space required by the classical radix sorting algorithm is: radix*n, while it is n if our algorithm is used. Compared with the classical algorithm, the space complexity is lower as the storage space is reduced by a half. The time efficiency of our algorithm is equivalent to those of heap sort, merge sort and quick sort. For a big set of unequally distributed data which contains more shorter data and less longer data, the time efficiency is even higher.

References

Chen Xuan, Ma Si, Gu Minhao, Zhu Kejun; Application of Radix Sorting in High Energy Physics Experiment,
Ge Hao, Yang Chuanjian; Research on Radix Sort Based on Distributing Counting Sort Method, Computer Technology and Development, 2008,18(2):122.
Yan Weimin, Wu Weimin; Data Structure (C Language), Beijing: Tsinghua University Press, 2010.
Yang Xiucheng, Li Tong, Zhao Na, Liang Ligang, Li Chao; Design and Analysis of Calculation Sorting Algorithm; Application Research Of Computers; 2014,31(3):658-662.
Yin Renkun; Data Structure (C Language Description), Beijing: Tsinghua University Press, 2012.