Unshuffle Sort and Ideal Merge

Art S. Kagel
ASK Database Management
222 Dunhams Corner Road
East Brunswick, NJ 08816
+1 732-213-5367
art@askdbmgt.com

ABSTRACT

In this paper the author describes a unique data sort algorithm, Unshuffle Sort, and a new algorithm for the merging of multiple sorted sinks, Ideal Merge.

Unshuffle is a distribution sort in two phases. The effort to optimize the second phase of the algorithm resulting in the development of an algorithm for the merge of sorted sinks of which the author has found no previous description and which can be shown to be the best possible.

Overall Unshuffle can be shown to a highly efficient sort when applied to real world data sets which are seldom truly random.

Unique features of Unshuffle include:

- Performs no exchanges
- Can be applied to unusual data set sources including arrays, linked lists, and streaming data
- Can begin to supply sorted output to consumers expecting streaming data immediately upon the arrival of the final input element

CCS Concepts

·Theory of computation~Sorting and searching - 500

Keywords

Sorting, Merging, Distribution Sort

1. INTRODUCTION

In 1984, as part of a software development project, I was tasked to find an efficient way to sort a large data set which was to be kept in memory in a linked list structure. As part of the requirements I was given, the data was not to be copied into an array for sorting as insufficient memory was available. The program would be running on the then new IBM PC/AT and was already pushing the limits of its memory capabilities.

Out of that project came the sorting algorithm I have named Unshuffle Sort (for reasons described below) and ultimately also the merge algorithm that I call the Ideal Merge. An early version of Unshuffle was published in Computing Language Magazine in 1986. [1] This paper describes the final version of the sort as I continue to use it today.

2. INVESTIGATIONS

Existing algorithms which were known to be applicable to sorting linked lists in place such as bubble sorts and Shell Sort were rejected as too slow for a large data set. Something new was needed. It occurred to me that the data sets we were processing were partially ordered. If I could devise a method for extracting that existing order, in effect if I could unshuffle the cards, so to speak, I might find a better way. There followed many

hours sitting with a deck of playing cards that resulted ultimately in the Unshuffle Sort.

Along the way I determined that Unshuffle is not restricted to sorting linked lists but can be applied to any data source including linked lists, arrays, and streams of data.

3. ALGORITHM

Unshuffle is a distribution sort at its most basic level and is performed in two phases. The first phase is the distribution of elements onto restricted ordered deques which results in a set of sorted lists. The second phase is the merging of these sorted lists into a single output target which is typically in the same format as the input data.

For simplicity, and to honor my hours spent sorting playing cards, I will refer to such an ordered deque as a "pile". Piles are kept in a doubly linked list in the order they were created from oldest or "first" to newest or "last". By convention older piles are said to be linked to the "left" of newer piles and newer piles to the "right" of older piles in the linked list.

The pile structure used for the distribution phase is restricted in the sense that in any single pile elements may only be added to the ends of the deque and cannot be inserted between existing elements. A further restriction requires that the value of the element at the top of any pile be greater than that of the top element of the pile to its left and that the value of the element at the bottom of any pile be less than the value of the bottom element of the pile to its left. If an element arrives which cannot be placed at either end of any existing deque a new deque is created. The resulting piles are therefore ordered on both their top and bottom elements at any given moment during the distribution phase.

Section 3.1 describes the distribution phase of the sort and section 3.2 describes the merge phase.

3.1 DISTRIBUTION

Terms used in this description:

- Each pile has a least valued end, referred to as the "top" and a most valued end referred to as the "bottom".
- The phrase "current side" refers to the end (top or bottom) of the pile to be used for comparison. The side onto which the previous element was appended is the current side initially for the next element.
- References to "beyond" indicate a value which is less than the element at the top of some pile or greater than the element at the bottom of the pile according to the ordering criteria in use.
- References to "contained" indicate an element that has a value which is less than the element at the bottom of some pile or greater than the element at the top of the pile according to the ordering criteria in use and the side being compared.
- The phrase "current pile" refers to whichever pile is being used for comparison.
- The phrase "previous pile" refers to the pile to the right of the current pile.
- The phrase "left pile" refers to the pile linked immediately left of the current pile.
- The phrase "first pile" refers to the leftmost pile, ie the first pile created.
- The phrase "current of current" refers to the element on current side of the current pile to which a new element is being compared.

The distribution phase algorithm is as follows:

1. Take the first input item and create an initial or first pile with this item as both top and bottom elements. Note which side is the current side. Continue.
2. If no more items goto Phase II: Merging, else continue.
3. Take next input item for comparison.
4. Select the last pile for comparison as the current pile. Maintain the same side as the last comparison to be the current side. Continue.
5. If the value of the new item is equal to current of current append to the current

side of current pile and goto 2, else continue.

6. If the new value is beyond the current of current and the current pile is not the first pile then select the left pile as current, go to 5, else continue.
7. If the new value is beyond the current of current and the current pile is the first pile, append to the current side of the current pile and goto 2, else continue.
8. If the new value is contained within the current pile and this is not the last pile append to the current side of the previous pile and goto 2, else continue.
9. If the current pile is the last pile and only one side has been compared, switch sides and goto 4, else continue.
10. (At this point both top and bottom have been searched without a match) create a new pile containing the new value and append it to the right of the last pile as the new last pile and go to 2.

3.2 MERGING

Once all elements have been distributed onto piles, the resulting set of piles will a) each be a sorted list and b) the list of piles will be sorted according to its top-most element. Indeed the piles will also be sorted according to their bottom-most elements in the reverse order. This fact can be used to perform a reverse sort in the merge phase without inverting the comparison criteria during the distribution. To create the output we now have to apply a merge of sorted sinks.

The Ideal Merge begins with a set of sorted sinks which, like the piles that result from the distribution, are sorted by their top element. Beginning from this point, the algorithm is (this assumes an ascending sort):

1 Output the first element of the first pile (this could be the top or bottom, for simplicity I will refer to the active element of each pile as the "top" element) and goto 2.

2 If the first pile is now empty,

2.1 discard that pile (the second pile is now first).

2.1.1 If no more piles exit

2.1.2 Else go to 1.

2.2 Else go to 3.

3 Compare the next element (now its first remaining or top element) on the first pile to the top element on the second pile.

3.1 If the first pile contains the lesser value then go to 1.

3.2 Else go to 4.

4 Remove the first pile from the list of piles. Use a binary search of the top elements of the remaining piles to find its proper position in the sorted list of piles and insert it there. Go to 1.

4. NOTES

4.1 Stability

Unshuffle is not naturally a stable sort. Stable output can be simulated by appending the input arrival order to the key and sorting this extended key.

4.2 Unique sort results

Uniqueness filtering during the sorting process can be applied both during the distribution phase and the merge phase. Filtering for uniqueness solely during the distribution is insufficient, though it is sufficient to filter only during the merge phase. Duplicate key filtering during both the distribution phase and merge phase will improve the performance of the sort overall as fewer elements will need to be merged and filtered during the merge.

4.3 Low Cost

Unlike many other sort algorithms, Unshuffle does not swap data elements, does not insert elements between other elements, and only links and unlinks deque members. Therefore performance is not dependent on the size of the data elements themselves but only on the number of elements, the entropy inherent in the data (higher entropy results in more piles and so more comparisons), and the size and complexity of the comparison key.

5. ORDER OF THE PHASES

5.1 Merge Phase – Ideal Merge

The order of the Ideal Merge used during the merge phase is O(N*log((M+1)/2)) where M is the number of piles and N is the total number of elements in all piles.

5.2 Distribution Phase

During the distribution phase each element is compared to at most M+1 other elements where M represents the number of piles at the time. The number of piles is dependent on the entropy or randomness of the data. The upper bound of M is one half of the number of unique key values. If all elements are unique and the data is fully random, then M approaches N/2. However, in practice, even given fully random input, the actual number of piles is typically far smaller than N/2 and the closer to N/2 that M approaches, the fewer comparisions are required to settle each new element onto the correct pile. Indeed the only way for M to equal N/2 is if the largest and smallest elements arrive first followed by the second largest and smallest, etc. resulting in N comparisons on average and 2N comparisons at worst.

The general order of the distribution phase is O(kN) where k is a measure of the entropy in the data and is proportional to the number of piles. That means that for a given level of entropy the behavior of Unshuffle is linear. Best case behavior is when Unshuffle is presented with a data set that is already ordered or ordered but reversed. In this case k=1, the distribution produces a single pile that need not be merged, and the performance of Unshuffle is comparable to an order validation run with O(N) with the slight additional overhead of N pointer assignments.

6. Applications

Real world data is not random. In business applications we may sort, for example, on an order number. Order numbers are typically monotonically increasing when assigned but some may arrive in the system out of order. In other cases the sort key may be preallocated to different offices (examples include license plate numbers and preprinted numbered paper order sheets) and assigned to entities as required. Here there will be runs of sequential values interleaved with other ordered runs out of order with each other. Similarly for ordering on transaction date/time,

All of these data sets are typical of real world data and are sorted more efficiently by Unshuffle than by algorithms like quicksort and heapsort that are optimized for random data and may even display worst case behavior when confronted with certain low entropy data sets.

7. ACKNOWLEDGMENTS

The author offers acknowledgment of the detailed and exhaustive work in the field of sorting by Donald Knuth and its influence on his continuing interest in the subject.

8. REFERENCES

[1] Art S. Kagel. Unshuffle, not quite a sort. Computer Language*, vol 3,* num 11 (Nov. 1986).