Egyptian Informatics Journal (2010) **11**, 19–26

Cairo University

Egyptian Informatics Journal

www.elsevier.com/locate/eij
www.sciencedirect.com

ORIGINAL ARTICLE

An empirical study on SAJQ (Sorting Algorithm for Join Queries)

Hassan I. Mathkour

Department of Computer Science, King Saud University, Riyadh, Saudi Arabia

Received April 2009; accepted 16 May 2010
Available online 23 July 2010

KEYWORDS

Sorting;
Stable–Sorting;
Sort-merge algorithms;
Auxiliary storage sorting;
Merging;
Database queries

Abstract Most queries that applied on database management systems (DBMS) depend heavily on the performance of the used sorting algorithm. In addition to have an efficient sorting algorithm, as a primary feature, stability of such algorithms is a major feature that is needed in performing DBMS queries. In this paper, we study a new Sorting Algorithm for Join Queries (SAJQ) that has both advantages of being efficient and stable. The proposed algorithm takes the advantage of using the m-way-merge algorithm in enhancing its time complexity. SAJQ performs the sorting operation in a time complexity of $O(n \log m)$, where n is the length of the input array and m is number of sub-arrays used in sorting. An unsorted input array of length n is arranged into m sorted sub-arrays. The m-way-merge algorithm merges the sorted m sub-arrays into the final output sorted array. The proposed algorithm keeps the stability of the keys intact. An analytical proof has been conducted to prove that, in the worst case, the proposed algorithm has a complexity of $O(n \log m)$. Also, a set of experiments has been performed to investigate the performance of the proposed algorithm. The experimental results have shown that the proposed algorithm outperforms *otherStable–Sorting algorithms that are designed for join-based queries*.

1. Introduction

In database management systems (DBMS), sorting operation is considered as an integral component in most performed queries [1–3]. Usually, sorting operations have a great impact on the system performance and space needed [4–6]. In general, the performance of DBMS queries depends on performance of the sorting algorithm. Also, most DBMS queries require sorting with stable results [4,7–9], where stable sorting preserves the order of those records with same key values. In other words, if the sorted file has two records with the same key value, the outcome of the sorting algorithm will have the two records in the same order, although their positions relative to other records may change [2,10,11]. Stability of a sorting algorithm is a property of the algorithm, not of the compari-

E-mail addresses: mathkour@ccis.ksu.edu.sa, binmathkour@yahoo.com

Peer review under responsibility of Faculty of Computers and Information, Cairo University.
doi:10.1016/j.eij.2010.06.003

Production and hosting by Elsevier

son mechanism [7,8,12,13]. For instance, Quick Sorting algorithm is not stable while Merge Sort algorithm is stable.

Internal memory sorting algorithms are only concerned with applying the sorting mechanism on chunks of data that can fit in main memory. Usually, external memory sorting algorithms use internal memory sorting to fulfill part of the sorting procedure. They typically perform two phases [7,14–16]. In the first phase, a set ordered sub-files is produced. The produced sub-files are processed to produce a totally ordered output data file. Merge-Based Sorting algorithms constitute an important external memory sorting algorithms class, where input data is partitioned into data chunks of approximately equal size, sorts these data chunks in main memory and writes the sorted chunks into disk. The sorted chunks are retrieved into main memory, merged then the sorted output is written into disk.

External memory sorting performance is often limited by I/O performance [9,17,18]. Disk I/O bandwidth is significantly lower than main memory bandwidth. It is important to minimize the amount of data written to or read from disks. Even with the recent advances in main memory technology, still very large files do not fit in RAM. In such case, data files must be sorted in at least two passes. In each pass, disk blocks are read and written to the disk. CPU-based sorting algorithms incur significant cache misses on data sets that do not fit in the data caches [18,19]. Therefore, it is not efficient to sort partitions comparable to the size of main memory [16,20,21]. These result in a tradeoff between disk I/O performance and CPU computation time spent in sorting the partitions. In merge-based external sorting algorithms, the time spent in Phase 1 can be reduced by choosing run sizes comparable to the CPU cache sizes. However, this choice increases the time spent in Phase 2 to merge a large number of small runs.

In this paper, a new Sorting Algorithm for Join Queries (SAJQ) [22] is studied. SAJQ is categorized as a Stable–Sorting algorithm that reduces the number of comparisons, element moves and the required auxiliary storage will be presented. Our proposed algorithm partitions the input data into data chunks that are already sorted, and then writes these runs to disk. The sorted runs are merged in main memory and written back into disk. In internal sorting, the proposed algorithm has three performance measures, element comparisons, element moves, and auxiliary storage used. SAJQ operates with an average $m\log m$ element comparisons, $m\log m$ element moves, and exactly n auxiliary storage, where n is the size of the input array to be sorted, and m is the average size of sub-arrays. No auxiliary arithmetic operations with indices will be needed. The proposed algorithm has been tested in extensive performance studies. Experimental results have shown that when the proposed algorithm is adapted to sort externally, it accelerates the computation of equi-join and non-equi-join queries in database systems. Also, it accelerates numerical statistic queries on data streams. The data sets used in the performance studies are databases consisting of up to one million values.

In the following sections, the SAJQ algorithm, and the computational requirement analysis will be presented in details. The SAJQ algorithm and the proof of its stability are presented in Section 2, and the m-way-merge algorithm is discussed in Section 3. The analysis of time complexity and the analysis of the external version of the SAJQ algorithm are discussed in Section 4. Experimental results are presented in Section 5. Section 6 concludes the paper.

2. The SAJQ Sorting Algorithm

Several issues should be considered to guarantee stability. One of those issues concerns with the mechanism of placing elements in corresponding segments, where the input array has to be scanned in sequential order from element a_1 to element a_n. In order to determine where to place a specific element in a specific segment, that element should be placed in the first segment that meets the condition of being greater than or equal to the last element in that sub-array. In the merging process, the m-way-merge [23] operation, that produces the output list, should select values from segments $\sigma_1, \sigma_2, \ldots, \sigma_m$ in preference according to the segment order. This will be discussed further in the following section.

The main idea of the proposed algorithm is to divide the elements a_i, $i = 1,\ldots,n$ of the input array A with length n into some disjoint segments $\sigma_0, \sigma_1, \ldots, \sigma_m$, of equal lengths l; $l = n/m$. All elements in segments σ_i's, $i = 1,\ldots,m$, are sorted according to key k. The algorithm starts with empty segments $\sigma_0, \sigma_1, \ldots, \sigma_m$, i.e., for each segment σ_i, $i = 1,\ldots,m$, the pointer to its last element $last_i = 0$. a_i is compared to the last element of the not full segments σ_j, $j = 1,\ldots,m$. a_i should fall in one of the following two cases:

- $a_i \geqslant \sigma_j(last_j)$, where $j = 1,\ldots,m$: a_i is appended to σ_j.
- $a_i < \sigma_j(last_j)$ for all j with $last_j \neq l$: insert a_i in σ_{temp}.

If the temporary segment σ_{temp} is full, then the m segments are m-way-merged, the 2-way-merged with the temporary segment σ_{temp}. Assume the number of sorted elements is L. The L elements are divided orderly on the first S segments; $S = \lceil \frac{L}{l} \rceil$; such that the first $S-1$ segments are full. The first $S-1$ segments will not be included in the subsequent inserts. The algorithm will consider only those buffers from S to m.

In the merging process, we should consider the segments order. When two segments σ_{k_1} and, σ_{k_2}, $k_1 < k_2$ are merged and if there exists two elements a_{i_1} in σ_{k_1} and a_{i_2} in σ_{k_2}, such that $a_{i_2} = a_{i_2}$, then in the merged segment, a_{i_2} is placed before a_{i_2}. In the following example, a sorting problem is performed on an input array. In the first phase, the input array is portioned into two sorted sub-arrays A and B. The second phase merges the elements of A and B into the output array.

In the following two examples, we depict the best case and worst cases of using the SAJQ technique.

Example 1 (best case): Assume the following input array X, $n = 14$,

Input array X
1
6
2
7
2
9
11
5
15
17
6
7
15
17

By using the SAJQ technique, with $m = 2$, the two sorted sub-arrays are built as follows:

- The first element "1" of X is placed in sub-array A.
- The second element "6" is compared with the last element in A; i.e., "1", "6" > ="1", then "6" is inserted after "1".
- The third element "2" is compared with the last element in A; i.e., "6", "2" < "6", then compared with the last element in B; i.e., "null". Element "2" is placed in sub-array B.
- The fourth element "7" is compared with the last element in A; i.e., "6", "7" > "6", then "7" is inserted after "6".
- The procedure is repeated for all elements in X. The sorted sub-arrays A and B, with elements position in X, are given below.

Position in X	*A*	*Position in X*	*B*	*Position in X*	*Temp*
1	1	3	2		–
2	6	6	2		
4	7	9	5		
7	9	12	6		
8	11	13	7		
10	15	14	15		
11	17	15	17		

The final sorted array is obtained by using an m-way merging algorithm, as given below.

	Output
1 from *A*	1
1 from *B*	2
2 from *B*	2
3 From *B*	5
2 from *A*	6
4 from *B*	6
3 from *A*	7
5 from *B*	7
4 from *A*	9
5 from *A*	11
6 from *A*	15
6 from *B*	15
7 from *A*	17
7 from *B*	17

Example 1 (worst case): Assume the following input array X, $n = 14$,

Input array X
17
16
15
14
11
9
8
7
6
5
4
3
2
1

By using the SAJQ technique, with $m = 2$, the two sorted sub-arrays are built as follows:

1. The first element "17" of X is placed in sub-array A.
2. The second element "16" is compared with the last element in A; i.e., "17", "16" < "17", then compared with the last element in B; i.e., "null". Element "16" is placed in sub-array B.
3. The third element "15" is compared with the last element in A; i.e., "17", "15" < "17", then compared with the last element in B; i.e., "16", "15" < "16", then compared with the last element in temp; i.e., "null". Element "15" is placed in sub-array temp.
4. The fourth element "14" is compared with the last element in A; i.e., "17", "14" < "17", then compared with the last element in B; i.e., "16", "14" < "16", then compared with the last element in temp; i.e., "15", "14" < "15". Now, we should merge the three arrays A, B and temp, and put the result in A.
5. The fourth element "14" is compared with the last element in A; i.e., "17", "16" < "17", then compared with the last element in B; i.e., "null". Element "14" is placed in sub-array B.
6. The fifth element "11" is compared with the last element in A; i.e., "17", "11" < "17", then "11" is compared with the last element in B; i.e., "14", "11" < "14", then compared with the last element in temp; i.e., "null". Element "11" is placed in sub-array temp.
7. The procedure is repeated for all elements in X until sub-array A is full. In this case m is decreased by 1, and sub-array B is used as the first sub-array.
8. The procedure continues until the remaining elements in X are placed in B.

Steps 1–3						*Step* 4				*Steps* 5–7					
A		*B*		*Temp*		*A*		*B*	*Temp*	*A*		*B*		*Temp*	
1	17	2	16	3	15	3	15	–	–	3	15	4	14	11	
						2	16			2	16				
						⇒ *1*	*17*			⇒ 1	17				

Position in X	*A*	*Position in X*	*B*
7	8	14	1
6	9	13	2
5	11	12	3
4	14	11	4
3	15	10	5
2	16	9	6
1	17	8	7

The final sorted array is obtained by using an m-way merging algorithm, as given below.

	Output
1 from *A*	1
1 from *B*	2
2 from *B*	3
3 From *B*	4
2 from *A*	5
4 from *B*	6
3 from *A*	7
5 from *B*	8
4 from *A*	9
5 from *A*	11
6 from *A*	14
6 from *B*	15
7 from *A*	16
7 from *B*	17

The SAJQ algorithm is given below.

Algorithm SAJQ

```
{
  // The first part of the proposed algorithm is the stable formation
of segments. An additional m sub-arrays σ_j, and last_j = 0,
j = 1,...,m are required.//
  i = 1; // i is set to the first element of the input array//
  b = 1; // b is the starting buffer to append items to//
  do while (i ⩽ n)
{
  STEP1: Read element a_i from the input array;
  STEP2:
    j = b; append = 0;
    do until (append = 1 or j > m)
      {if a_i ⩾ σ_j(last_j) and last_j ≠ l // subarray σ_j is not full, and
      a_i ⩾ last-element in σ_j;
        then {last_j + =1; σ_j(last_j) = a_i; //append a_i to σ_j;
              append = 1;}
        else j++;
      }
      if (append = 1)
        then {append = 0; i++; goto STEP1;};
          else {insert a_i in its right position in σ_temp, last_temp + 1;
          i++; // all sub-arrays σ_j
          are not suitable for insertion (either full or a_i < last-ele-
          ment in σ_j);
          if last_temp > 1 then
            {(m − b + 1)-way-merge segments σ_j's, j = b,...,m;
            2-way-merge the results with segment σ_temp;
        L is the total number of elements in the merged buffers;
          Do while L > l
            {Fill buffer b with l items;
            L = L − l
            b = b + 1;}
          Fill buffer b with L items;
          goto STEP1;}
          }
m-way-merge segments σ_j's, i = 1,..., m into final output array;
          }
```

3. The merging

In this section, we discuss the m-way-merge algorithm. 2-Way-merge is discussed first to show complexity measures and stability issues of the algorithm. The discussion is then generalized to the proposed more general case of m-way-merge.

Merging is the process whereby two sorted lists of elements are combined to create a single sorted list. Originally, most sorting was done on large database systems using a technique known as merging. Merging is used because it easily allows a compromise between expensive internal Random Access Memory (RAM) and external magnetic mediums where the cost of RAM is an important factor. Originally, merging is an external method for combining two or more sorted lists to create a single sorted list on an external medium.

The most natural way to merge two lists A with n_1 sorted elements and B with n_2 sorted elements is to compare the values of the two lists starting from the lowest (smallest) locations in both lists and then output the element with the smaller value. The next two smallest values are compared in a similar manner and the element with smaller value is placed in the output array. This process is repeated until one of the lists is exhausted. Such merging mechanism requires $n_1 + n_2 - 1$ comparisons and $n_1 + n_2$ data moves to create a sorted list of $n_1 + n_2$ elements. The order in which same values in lists A and B are placed in the output array favors those elements coming from A rather than from B. Merging in this manner is easily made stable. In addition, stability is guaranteed regardless of the nature of the input permutation.

A merge algorithm is stable if it always leaves sequences of equal data values in the same order at the end of merging.

If the merge algorithm is merging two lists such that it keeps the indices of a sequence of equal values from a given list in sorted order, i.e., they are kept in the same order as they were before the merge as illustrated in the example 1, the merge algorithm is said to be stable. Otherwise, the algorithm is said to be unstable.

Stability is defined such that values from A are always given preference whenever a tie occurs between values in A and B, and that the indices of equal set of values are kept in sorted order at the end of the merge operation. The following lemma states that the Stable–Sorting algorithm is stable.

Lemma 1. *Let A be an array with n elements. The Stable sort algorithm guarantees that the resulted sorted array is stable.*

Proof 1. The algorithm consists of a series of two phase episodes. The first phase of each episode produces a set of sorted sub-arrays, while the second phase merges the sorted sub-arrays. The elements of the input array are placed in sub-arrays in an ordered fashion. Two elements with same value are placed either in the same sub-array, where the element comes first in the input array is placed first in that sub-array. If the two elements are placed in two different sub-arrays, then the sub-array that contains the first element should have lower order than the other sub-array that contains the second element. The m-way-merge technique merges the sorted sub-arrays in order to produce the sorted array. Before each of the followed episodes, the sorted elements are placed in order in the m sub-arrays. In the followed episodes, the two phases are identical to those of the first episode. □

The proposed m-way-merge operation can be implemented as a series of 2-way-merge operations with preserving stability requirements, or can be merge m lists in the same time. The 2-way-merge operation can be generalized for m sorted lists, where m is some positive integer >2. For keeping track of the next smallest value in each list during the merge operation, m pointers will be needed. The number of comparisons needed for merging m sorted lists, each consisting of l data items is $O(n\log_2(m))$, where $n = ml$.

4. Performance analysis

4.1. Time complexity

Scanning the elements of the input array of n elements and divide it into m sorted sub-arrays is done in linear time, except when the temporary array reaches its maximum length. The best case, when the temporary array is not reaching its maximum length, needs $O(n)$ comparison and $O(n)$ moves. The maximum time required for doing the sorting process, and

keeping the stability required, needs $O(n\log m)$ comparison and $O(n)$ moves. In the worst case, each of the empty buffers will hold only one element, and the algorithm may have to insert elements to the temporary array until it gets full. The process is repeated until all elements are inserted. Below, we prove that the Stable–Sorting algorithm has a time complexity of $O(n\log m)$, where m could be much smaller than n.

Assume $n = m^r$, for some integer $r > 0$, if $r = 1$, which means number of buffers m equals to number of records n, the proposed approach could have, in the worst case, $O(n\log n)$ time complexity. If the number of buffers is decreased, by increasing r, we may get less cost but to a certain limit. If $r \to \infty$; $(m \to 1,\ l \to n)$ the proposed approach could have $O(n\log n)$ cost. From Figs. 2–4, we conclude that the best value for r is 4.

Best case:

$<= n + n\log(m) + l\log(l) \Rightarrow O(n\log(m))$

Worst case:
1 :m getting m elements into m buffers
:$l\log(l)$ getting l elements (sorted) into temp buffer
:$m\log(m)$ m-way-merge of m buffers
:$(m + l)$ merge with temp buffer
:$(m + l)$ placing l elements in buffer 1 and m elements in buffer 2
:Total $= m + l\log(l) + m\log(m) + 2(m + l)$

In general in iteration i

i :$m - i$ getting $m - i$ into $(m - i)$ buffers
:$l\log(l)$ getting l elements (sorted) into temp buffer
: $<(im - i)\log(m)$ $(m - i)$-way-merge of $m - i$ buffers $< im\log(m - i)$
: $(im - i + l)$ merge with temp buffer
$(im - i + l)$ placing l elements in buffer $m - i$ and $im - x(i)$ elements in buffer i
: Total $= (m - i) + l\log(l) + (im - i)/2\log(m) + 2(im - i + l)$
The process will be repeated m times
Total cost $\leqslant \sum_m^{i=1}(m - i) + l\log(l) + (im - i)\log(m) + 2(im - i + l)$ or
Total cost $\leqslant m(m + l + l\log l - 2) + \frac{m(m+1)}{2}(2(m - 1) + (m - 1\log m - 1))$
$\leqslant n + m^3 + \frac{m^2}{2} - \frac{3m}{2} + n\log l + \frac{m^3 - m}{2}\log m$
In case that $n = m^3$, *then* $l = m^2$
Total cost $\leqslant 2n + \frac{1}{2}n^{\frac{2}{3}} - \frac{3}{2}m + 2n\log m + \frac{1}{2}(n - m)\log m$
Total cost is $O(n\log m)$
if $n = m^4$, $l = m^3$
Total cost $\leqslant n + n^{\frac{3}{4}} + \frac{1}{2}n^{\frac{1}{2}} - \frac{3}{2}m + 3n\log m + \frac{1}{2}\left(n^{\frac{3}{4}} - m\right)\log m$
Total cost is $O(n\log m)$
In general, if $n = m^r$, $l = m^{r-1}$

$$\text{Total cost} \leqslant n + n^{\frac{3}{r}}\frac{1}{2}n^{\frac{3}{r}} - \frac{3}{2}m + (r - 1)n\log m + \frac{1}{2}\left(n^{\frac{3}{r}} - m\right)\log m \quad (1)$$

If $r = 1$; $m = n$, the proposed approach could have $O(n\log n)$ cost. If we decrease the number of buffers, by increasing r, we may get less cost but to a certain limit. Part of the total cost equation, $(r - 1)n\log(m)$, will increase the total cost. If $r \to \infty$; $(m \to 1, l \to n)$ the proposed approach could have $O(n\log n)$ cost.

In order to optimize the results of the inequality (1), with respect to m, we can either optimize the problem analytically by finding its global minimum, or run the inequality equation against some possible values of m. Because of the great complexity of the inequality equation, we have chosen the later solution. For three different values of n; n = 10,000, 100,000, and 1,000,000, Figs. 1–3, respectively, have shown that out technique is almost optimum when m chosen to be 4.

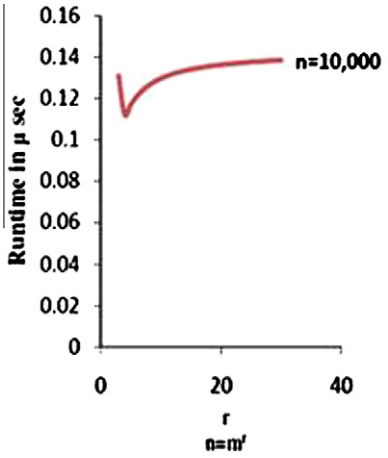

Figure 1 Run time with n = 10.000 tuples.

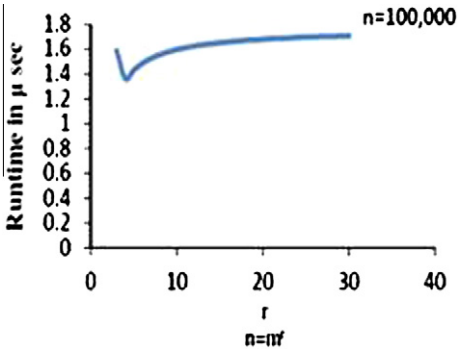

Figure 2 Run time with n = 100.000 tuples.

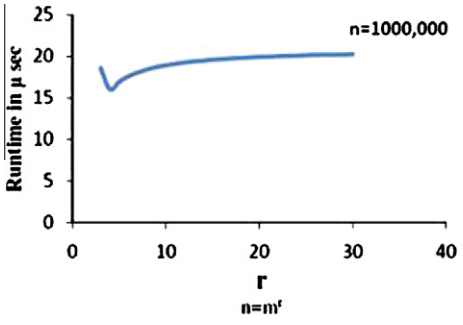

Figure 3 Run time with n = 1000.000 tuples.

4.2. Space complexity

There is extra auxiliary storage equal to an array of the same size as the input array. Clever implementation of the Stable–Sorting algorithm can decrease the required auxiliary storage. There are no indices or any other hidden data structures required for the execution of the algorithm. Auxiliary storage need for the proposed algorithm is $O(n)$.

4.3. External sorting version of the proposed algorithm

External memory sorting algorithms reorganize large datasets. They typically perform two phases. The first phase produces a set of files; the second phase processes these files to produce a totally ordered permutation of the input data file. An important external memory sorting algorithms is Merge-Based Sorting where input data is partitioned into data chunks of approximately equal size, sorts these data chunks in main memory and writes the "runs" back to disk. The second phase merges the runs in main memory and writes the sorted output to the disk. Our proposed algorithm partitions the input data into equal sized data chunks. Each chunk is completely sorted in memory, using the same technique described in Section 2. It starts by placing chunk A with length n elements into memory. For each chunk A, the algorithm starts with empty segments $\sigma_1, \sigma_2, \ldots, \sigma_m$, then in the same way described in the Stable–Sorting algorithm, elements a_i, $i = 1, \ldots, n$ of chunk A are divided into some disjoint sorted segments $\sigma_1, \sigma_2, \ldots, \sigma_m$, of equal lengths l; $l = n/m$, and merged back then stored in disk. The process is repeated for all chunks A's in file f. In, the second phase, the sorted chunks A's are retrieved and merged back to disk.

External memory sorting performance is often limited by I/O performance. Disk I/O bandwidth is significantly lower than main memory bandwidth. Therefore, it is important to minimize the amount of data written to and read from disks. Large files can not fit in RAM. In the proposed implementation, the input file is sorted in two passes. Each pass reads and writes to the disk.

In this paper, the SAJQ algorithm is adapted to perform external sorting using run sizes comparable to the available RAM size to minimize time required for the second part of the algorithm which is the final merging operation. This is done by assuming the input array to be sorted is stored in a file on the hard disk called f. f is much larger than available space in main memory. The main memory available is g bytes. We divide the space available in main memory into two partitions, g_1 and g_2; $g = g_1 + g_2$. The first g_1 bytes of file f are copied into the first partition. The other g_2 bytes of the main memory are divided into $m + 1$ blocks each has length l; $g_2 = (m + 1)l$ and $g_1 = ml$. We perform the proposed algorithm on the portion of f in main memory. The g_1 bytes of the sorted output will be stored back in file f in place of its first g_1 bytes. Then we repeat the same procedure on the next g_1 bytes from file f until the whole file is exhausted. The second part is to m-way-merge the sorted g_1 chunks of file f. The merge procedure is done externally.

5. Experimental results

Two classes of performance studies have been carried on our proposed algorithm. In the first class of experiments, we have used the SAJQ algorithm as an internal sorter, where its performance has been compared to the performance of other

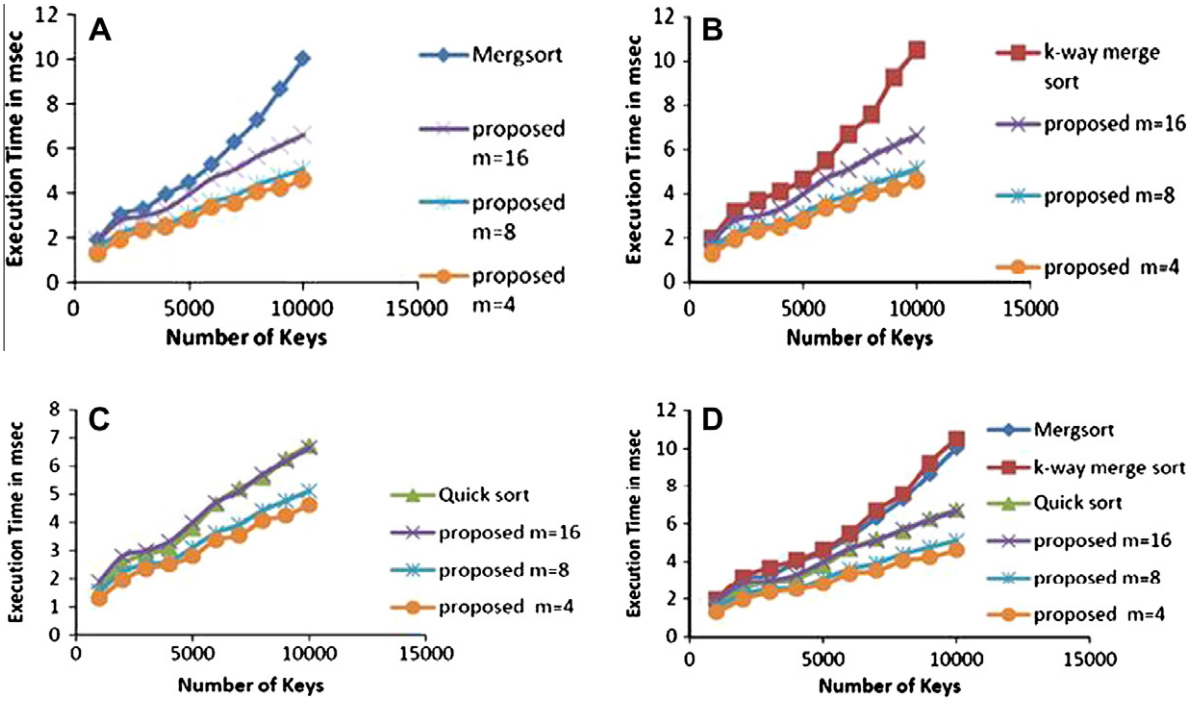

Figure 4 The proposed external SAJQ algorithm against mergesort, k-way mergesort, and quicksort algorithms, and a comparison of various sorting algorithms. (A) SAJQ algorithm against mergsort algorithm. (B) SAJQ algorithm against k-way mergesort algorithm. (C) SAJQ algorithm against Quick Sort algorithm. (D) SAJQ algorithm against all three Sorting algorithms.

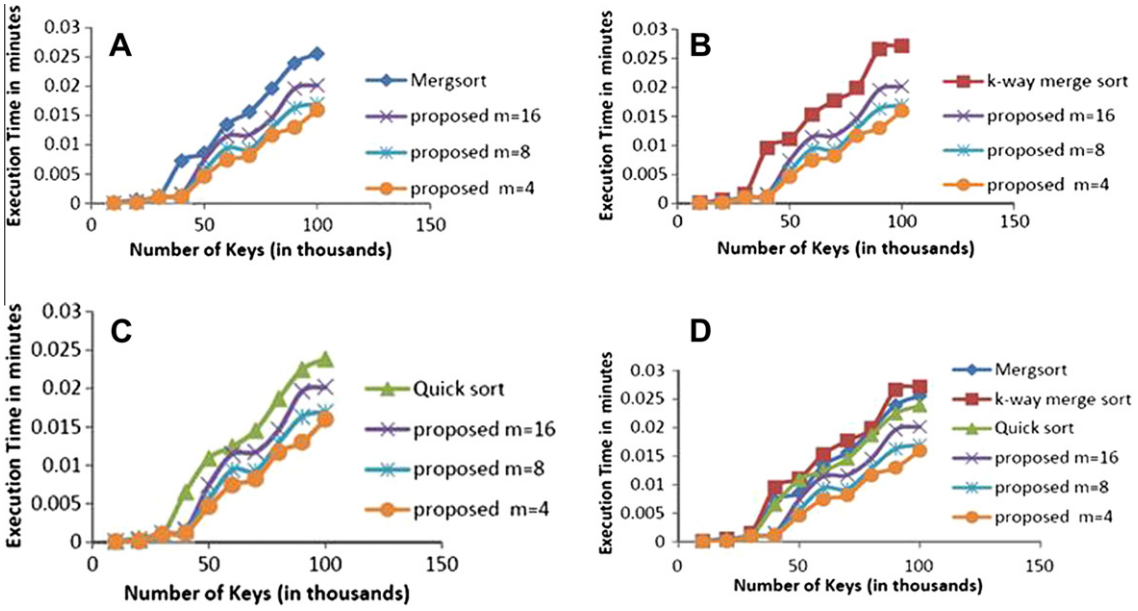

Figure 5 The proposed external SAJQ algorithm against mergesort, k-way mergesort, and quicksort algorithms, and a comparison of various sorting algorithms. (A) SAJQ algorithm against mergsort algorithm. (B) SAJQ algorithm against k-way sort algorithm. (C) SAJQ algorithm against Quick Sort algorithm. (D) SAJQ algorithm against all three Sorting algorithms.

well-known internal sorting algorithms. Those algorithms include the Merge-Sort, the k-way mergesort and the Quick Sort algorithms. The SAJQ algorithm was tested with three different values of m; 4, 8, and 16. Synthetic data has been used in the experimental study, where the number of generated keys was varied between 1000 and 10,000 keys, each key is 20 characters long. The performance improvement obtained by using our proposed algorithm with $m = 4$, against mergesort (Fig. 4a) is gaining on average a 45% speedup over mergesort optimized CPU implementation. Using our proposed algorithm with $m = 4$, against k-way mergesort (Fig. 4b) is gaining on average a 40% speedup over k-way mergesort optimized CPU implementation. The performance improvement obtained by using our proposed algorithm with $m = 4$, against Quick Sort (Fig. 4c) is gaining on average a 22% speedup over Quick Sort optimized CPU implementation, and finally, we depict in Fig. 4d the running times of sorting algorithms in milliseconds for a number of keys to be sorted ranging from 1000 to 10,000 keys each key is 20 characters long.

In the second class of experiments, we have used the SAJQ algorithm as an external sorter, where its performance has been compared to the performance of the external version of the Merge-Sort, the k-way mergesort and the Quick Sort algorithms. The SAJQ algorithm was tested with three different values of m; 4, 8, and 16. Synthetic data has been used in the experimental study, where the number of generated keys was varied between 10,000 and 100,000 keys, each key is 20 characters long. The performance improvement obtained by using our proposed algorithm with $m = 4$, against mergesort (Fig. 5a) is gaining on average a 33% speedup over mergesort optimized CPU implementation. Using our proposed algorithm with $m = 4$, against k-way mergesort (Fig. 5b) is gaining on average a 27% speedup over k-way mergesort optimized CPU implementation. The performance improvement obtained by using our proposed algorithm with $m = 4$, against Quick Sort (Fig. 5c) is gaining on average a 18% speedup over Quick Sort optimized CPU implementation, and finally, we depict in Fig. 5d the running times of sorting algorithms in milliseconds for a number of keys to be sorted ranging from 10,000 to 100,000 keys each key is 20 characters long.

6. Conclusions

In this paper, the SAJQ algorithm was presented. This algorithm is Stable–Sorting algorithm with $O(n \log m)$, comparisons where m is much smaller than n, and $O(n)$ moves and $O(n)$ auxiliary storage. Comparison of the new proposed algorithm and some well-known algorithms has proven that the new algorithm outperforms the other algorithms. Further, there is no auxiliary arithmetic operations with indices required. We have improved the performance of join operations by applying our fast sorting algorithm and compared its performance against other optimized nested join algorithms. Our algorithm also has low bandwidth requirements.

References

[1] LaMarca A, Ladner RE. The influence of caches on the performance of sorting. J Algorithm 1999;31(1):66–104. April.
[2] Anantha Krishna R, Das A, Gehrke J, Korn F, Muthukrishnan S, Shrivastava D. Efficient approximation of correlated sums on data streams. In: TKDE; 2003.
[3] Das Abhinandan, Gehrke Johannes, Riedewald Mirek. Approximate join processing over data streams. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data. ACM Press; 2003. p. 40–51.
[4] Van Lunteren Jan. Searching very large routing tables in wide embedded memory. In Proceedings of IEEE globecom; November 2001.
[5] Bentley JL, Sedgewick R. Fast algorithms for sorting and searching strings. In: ACM-SIAM SODA '97; 1997. p. 360–9.

[6] Franceschini G, Geffert V. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. In IEEE FOCS '03; 2003. p. 242–50.
[7] Knuth D. Sorting and searching, Volume 3 of the art of computer programming. Reading, MA: Addison-Wesley Publishing Company; 1973.
[8] Manegold Stefan, Boncz Peter A, Kersten Martin L. What happens during a join? Dissecting CPU and memory optimization effects. In: Proceedings of 26th international conference on very large data bases, vol. 26; 2000. p. 339–50.
[9] Arge L, Ferragina P, Grossi R, Vitter JS. On sorting strings in external memory. In: ACM STOC '97; 1997. p. 540–8.
[10] Y. Azar, A. Broder, A. Karlin, and E. Upfal, Balanced allocations. In: Proceedings of 26th ACM Symposium on the Theory of Computing, 593–602, 1994.
[11] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 2nd ed. Cambridge, MA: MIT Press; 2001.
[12] Broder Andrei, Mitzenmacher Michael. Using multiple hash functions to improve IP lookups. In: Proceedings of IEEE INFOCOM; 2001.
[13] Bandi N, Sun C, Agrawal D, El Abbadi A. Hardware acceleration in commercial databases: a case study of spatial operations. In: Proceedings of the 30th international conference on very large data bases, vol. 30; 2004. p. 1021–32.
[14] Agarwal Ramesh C. A super scalar sort algorithm for RISC processors. SIGMOD Record (ACM Special Interest Group on Management of Data) 1996;25(2):240–6.
[15] Arasu A, Manku GS. Approximate counts and quantiles over sliding windows. In: PODS; 2004.
[16] Bender MA, Demaine ED, Farach-Colton M. Cache-oblivious B-trees. In: IEEE FOCS '00; 2000. p. 399–409.
[17] Boncz Peter A, Manegold Stefan, Kersten Martin L. Database architecture optimized for the new bottleneck: memory access. In: Proceedings of the 25th international conference on very large databases, vol. 25; 1999. p. 54–65.
[18] Franceschini G. Proximity mergesort: optimal in-place sorting in the cacheoblivious model. In: ACM-SIAM SODA '04; 2004. p. p. 284–92.
[19] Andersson A, Hagerup T, H°astad J, Petersson O. Tight bounds for searching a sorted array of strings. SIAM J Comput 2001;30(5):1552–78.
[20] LaMarca A, Ladner R. The influence of caches on the performance of sorting. In: Proceedings of the ACM/SIAM SODA; 1997. p. 370–9.
[21] Franceschini G. Sorting stably, in-place, with $O(n \log n)$ comparisons and $O(n)$ moves. In: STACS '05; 2005.
[22] Mathkour H. A new sorting algorithm for accelerating join-based queries. In: Mathematical and computational applications; 2010.
[23] Aggarwal A, Vitter JS. The input/output complexity of sorting and related problems. Commun ACM 1988;31(9): 1116–27.