Mathematical and Computational Applications, Vol. 15, No. 2, pp. 208-217, 2010.
© Association for Scientific Research

A NEW SORTING ALGORITHM FOR ACCELERATING JOIN-BASED QUERIES

Hassan I. Mathkour
Department of Computer Science
King Saud University
Riyadh, 11543, Saudi Arabia
mathkour@ksu.edu.sa
binmathkour@yahoo.com

Abstract- The performance of several Database Management Systems (DBMSs) and Data Stream Management Systems (DSMSs) queries is dominated by the cost of the sorting algorithm. Sorting is an integral component of most database management systems. Stable sorting algorithms play an important role in DBMS queries since such operations requires stable sorting outputs. In this paper, we present a new stable sorting algorithm for internal sorting that scans an unsorted input array of length *n* and arranges it into *m* sorted sub-arrays. By using the *m*-way merge algorithm, the sorted *m* sub-arrays will be merged into the final output sorted array. The proposed algorithm keeps the stability of the keys intact. The scanning process requires linear time complexity *(O(n))* in the best case, and *O(n log m)* in the worst case, and the *m*-way merge process requires *O (n log m)* time complexity. The proposed algorithm has a time complexity of *O (n log m)* element comparisons. The performed experimental results have shown that the proposed algorithm outperforms other stable sorting algorithms that are designed for join-based queries.

Key Words- Sorting, Stable sorting, Auxiliary storage sorting, Merging.

1. INTRODUCTION

A stable sort is a sorting algorithm which when applied to records with same key values, they retain their order. In other words, for two records with the same key value, the outcome of the sorting algorithm will have the records in the same order, although their positions relative to other records may change [1, 2, 3]. Stability of a sorting algorithm is a property of the algorithm, not of the comparison mechanism [4, 5, 6, 7]. For instance, quick Sorting algorithm is not stable while Merge Sort algorithm is stable.

Sorting is an integral component of most database management systems (DBMSs) [8, 2, 9]. Sorting can be both computation-intensive as well as memory intensive [10, 7, 11, 12]. The performance of DBMS queries is often dominated by the cost of the sorting algorithm. Most DBMS queries require sorting with stable results [4, 10, 13].

External memory sorting algorithms reorganize large datasets. They typically perform two phases [4, 14, 15, 16]. The first phase produces a set of ordered sub-files, while the second phase processes these sub-files to produce a totally ordered output data file. A popular and important class of the external memory sorting algorithms is the Merge-Based Sorting algorithm, where input data is partitioned into data chunks of approximately equal size, sorts these data chunks in main memory and writes the “runs” to disk. The second phase merges the runs in main memory and writes the sorted output to the disk. Our proposed algorithm is inspired by the Merge-Based Sorting algorithm. In order to sort *n* elements placed in memory array *A*, the proposed algorithm divides the *n* elements of *A* into *m* ordered sub-arrays, each with length *l*; *l= n/m*. Initially, all sub-arrays are empty, then elements of array *A* are scanned and inserted into the ordered sub-arrays based on the fullness of sub-arrays, and whether the inserted element is greater than the last element of the sub-arrays or not. If all sub-arrays do not satisfy the insertion criteria, the scanned element is inserted in place into a temporary sub-array, *temp*, with length *l*. If the temporary sub-array is full, all sub-arrays and sub-array *temp* are merged. The merged elements are placed into the sub-arrays by fully filling them in order. The process continues with the elements left in array *A*. After scanning all elements of *A*, the produced m sub-arrays are merged to form the final sorted list.

External memory sorting performance is often limited by I/O performance [17, 13, 18]. 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 will not fit in RAM so we must sort the data in at least two passes. Each pass reads and writes to the disk. CPU-based sorting algorithms incur significant cache misses on data sets that do not fit in the data caches [19, 18]. Therefore, it is not efficient to sort partitions comparable to the size of main memory [20, 16, 21]. These results in a tradeoff between disk I/O performance and CPU computation time spent in sorting the partitions. For example, 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 stable sorting algorithm that reduces the number of comparisons, element moves, and required auxiliary storage will be presented. Our algorithm operates when used as an internal sorter 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 databases, and numerical statistic queries on data streams. The benchmarks used in the performance studies are databases consisting of up to one million values.

In the following sections, the proposed algorithm, and the computational requirement analysis will be presented in details. The proposed 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 proposed algorithm are discussed in section 4. Experimental results are presented in section 5. Section 6 concludes the paper.

2. THE PROPOSED STABLE SORTING ALGORITHM

There are some issues that have to be considered to guarantee stability. First, 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 segment, the element should be placed in the first segment that meets the condition that this element is greater than or equal to the last element in the sub-array. Also, some issues have to be considered in merging. The m-way-merge operation that produces the output list, should select values from segments $\square_1$, $\square_2$, …, $\square_m$ 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, $i=1,\dots, n$ of the input array A with length n into some disjoint segments $\sigma_0, \sigma_1, \dots, \sigma_m$, of equal lengths l; $l= n/m$. All elements in segments σ_i's, $i=1,\dots, m,$ are sorted according to key k. The algorithm starts with empty segments $\sigma_0, \sigma_1, \dots, \sigma_m$, i.e., for each segment σ_i , $i=1,\dots, m$, the pointer to its last element $last_i=0$. a_i is compared to the last element of the not full segments $\sigma_j, j=1,\dots, m$. a_i should fall in one of the following two cases:

- $a_i \geq \sigma_j\ (last_j)$, where $j=1,\dots, m$:

  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= $\left\lceil \frac{L}{l} \right\rceil$; 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 $a_{i_1} = a_{i_2}$, then in the merged segment, a_{i_1} is placed before a_{i_2}. The final sorted array is obtained by using an m-way merging algorithm. The new proposed algorithm is given below.

Algorithm Stable-Sorting

```
{
  // The first part of the proposed algorithm is the stable formation of segments. An additional $m$
  sub-arrays $\sigma_j$, and $last_j = 0, j=1,\dots, m$ 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 ($i \leq n$)
    {
      STEP1: Read element $a_i$ from the input array;
      STEP2:
        $J=b$; append =0;
        do until (append =1 or $j>m$)
          {if $a_i \geq \sigma_j\ (last_j)$ and $last_j \neq l$ // subarray $\sigma_j$ is not full, and $a_i \geq$ last-element in $\sigma_j$;
            then { $last_j$ +=1; $\sigma_j\ (last_j) = a_i$ ; //append $a_i$ to $\sigma_j$;
                   append = 1;}
        else j++;
      }
      if (append = 1 )
        then { append = 0; i++ ; goto STEP1;};
          else {insert $a_i$ in its right position in $\sigma_{temp}$;$last_{temp}$ +1; i++ ; // all subarrays $\sigma_j$ are
          not suitable for insertion (either full or $a_i$< last-element in $\sigma_j$);
          if $last_{temp}$ > l then
           {(m-b+1)-way-merge segments $\sigma_j$'s, j=b,..., m;
            2-way-merge the results with segment $\sigma_{temp}$;
          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 $\sigma_j$'s, i=1,..., m into final output array;
}
```

The following lemma states that the Stable-Sorting algorithm is stable.

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

Following the steps of the Stable Sort algorithm, the proof of the above lemma is obvious.

3. THE MERGING

In this section, the m -way-merge algorithm is going to be discussed. *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 elements and B with n_2 elements of presorted data values is to compare the values at the lowest (smallest) locations in both lists and then output the smaller of the two. The next two smallest values are compared in a similar manner and the smaller value output. This process is repeated until one of the lists is exhausted. Merging in this way 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 each value in the A and B list is selected for output to the C list is labeled by their subscript in the second listing of A and B. Merging in this manner is easily made stable.

In addition, this lower limit 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.

Example: Let *A* and *B* be two sorted lists with the values given in Figure 1.a. The *2*-way merge algorithm merges lists *A* and *B* into list *C* as given in Figure 1.b.

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

Figure 1.a

	C
1 from A_1	1
2 from B_1	2
3 from A_2	2
4 from A_3	6
5 from B_2	7
6 from B_3	7
7 from B_4	9
8 from A_4	11
9 from B_5	11
10 from A_5	15
11 from B_6	15
12 from A_6	17
13 from A_7	17
14 from B_7	17

Figure 1.b

Figure 1. The *2*-way merging of *A* and *B*

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 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

3.1. 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 $r \rightarrow \infty$; $(m \rightarrow 1, l \rightarrow n)$ the proposed approach could have *O(n log n)* cost. From fig. 2, 3, and 4, we conclude that the best value for *r* is 4.

Best Case:

<= n+ n log(m)+l log(l) ➔ 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 $\le \sum_{i=1}^{m}(\mathrm{m}-\mathrm{i}) \ +\mathrm{l}\log(\mathrm{l}) \ + \ (\mathrm{im}-\mathrm{i})\log(\mathrm{m}) \ + \ 2(\mathrm{im}- \ \mathrm{i}+\mathrm{l})$, or

Total cost $\le \mathrm{m}(\mathrm{m}+\mathrm{l}+\mathrm{l}\log\mathrm{l}-2)+\frac{\mathrm{m}(\mathrm{m}+1)}{2}(2(\mathrm{m}-1)+(\mathrm{m}-1)\log\mathrm{m}-1)$

$$\le \mathrm{n} + \mathrm{m}^3 + \frac{\mathrm{m}^2}{2} - \frac{3\mathrm{m}}{2} + \mathrm{n}\log\mathrm{l} + \frac{\mathrm{m}^3 - \mathrm{m}}{2}\log\mathrm{m}$$

<u>In case that n=m^3, then l=m^2</u>

Total cost $\le 2\mathrm{n}+\frac{1}{2}\mathrm{n}^{\frac{2}{3}}-\frac{3}{2}\mathrm{m}+2\mathrm{n}\log\mathrm{m}+\frac{1}{2}(\mathrm{n}-\mathrm{m})\log\mathrm{m}$

Total cost is O(n log m)

<u>if n=m^4, l=m^3</u>

Total cost$\le \mathrm{n}+\mathrm{n}^{\frac{3}{4}}+\frac{1}{2}\mathrm{n}^{\frac{1}{2}}-\frac{3}{2}\mathrm{m}+3\,\mathrm{n}\log\mathrm{m}+\frac{1}{2}\left(\mathrm{n}^{\frac{3}{4}}-\mathrm{m}\right)\log\mathrm{m}$

Total cost is O(n log m)

<u>In general, if n=m^r, l=m^{r-1}</u>

Total cost $\le \mathrm{n}+\mathrm{n}^{\frac{3}{r}}+\frac{1}{2}\mathrm{n}^{\frac{2}{r}}-\frac{3}{2}\mathrm{m}+(\mathrm{r}-1)\,\mathrm{n}\log\mathrm{m}+\frac{1}{2}\left(\mathrm{n}^{\frac{3}{r}}-\mathrm{m}\right)\log\mathrm{m}$

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. From Figures 2, 3, and 4, the best value of r is 4.

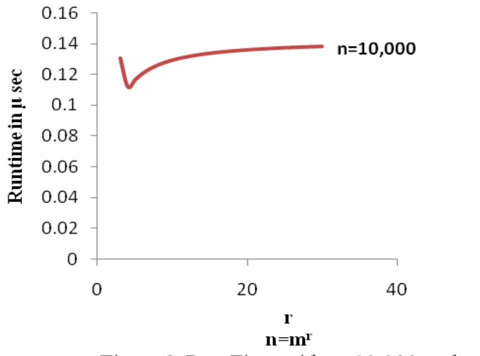


Figure 2. Run Time with n=10,000 tuples

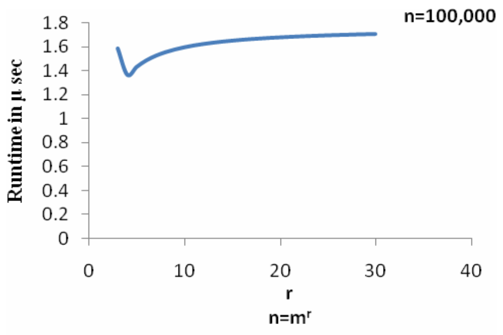


Figure 3. Run Time with n=100,000 tuples

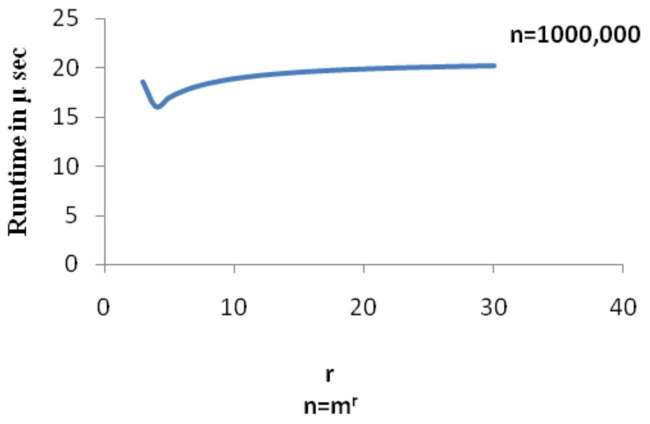


Figure 4. Run Time with n=1000,000 tuples

3.2. Space Complexity

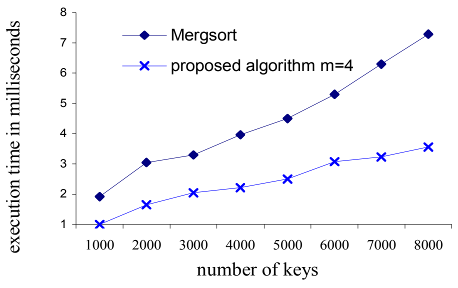
There is extra auxiliary storage equal to an array of the same size as the input array. Clever implementation of the proposed Stable-Sorting algorithm can decrease the required auxiliary storage. There are no indices or any other hidden data structures required for the execution of the algorithm. Auxiliary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

3.3. External Sorting Version of The Proposed Algorithm

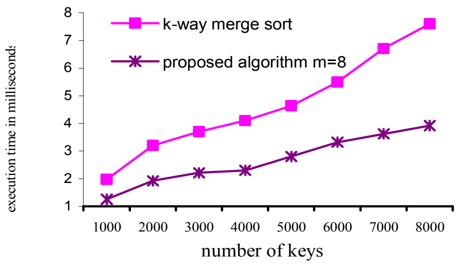
In the proposed implementation of our algorithm, the input file is sorted in two passes. Each pass reads and writes to the disk. The proposed algorithm is adapted to perform external sorting using run sizes comparable to the available RAM size to minimize the time required for the second part of the algorithm which is the final merging operation. This is done by assuming the input array will be sorted in a file on the hard disk called *f*. The file f is much larger than the available space in main memory. For an available main memory of *g* bytes, we divide the space available in the main memory into two partitions, g_1 and g_2; $g=g_1+g_2$. The first g_1 bytes of the 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 the 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 performed externally.

5. EXPERIMENTAL RESULTS

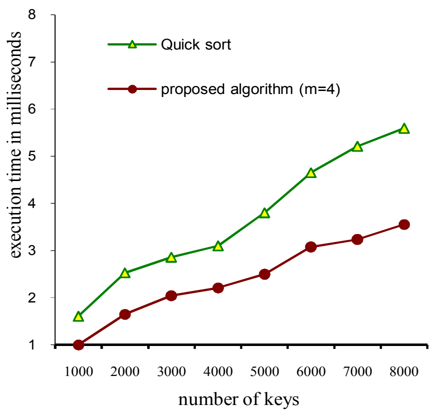
Performance study is being carried on our proposed algorithm as an internal sorting algorithm and its performance is compared to other well known internal sorting algorithms. These sorting algorithms are Merge-sort, K-way merge sort and quick sort algorithms.

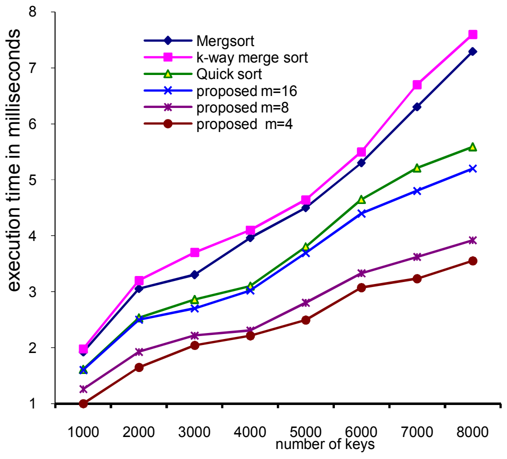


(a)



(b)





(c) (d)

Figure 5. The proposed algorithm vs : (a) Mergesort, (b) K-way Mergesort, and (c) Quicksort (d) A comparison of various sorting algorithms

The performance improvement obtained by using our proposed algorithm with *m=4*, against mergesort (Figure 5.a) is gaining on the average a 45% speedup over mergesort optimized CPU implementation. With *m=8*, our proposed algorithm on average is 40% faster than k-way mergesort optimized CPU implementation (Figure 5.b).

The performance improvement obtained by using our proposed algorithm with *m=4*, against Quick sort (Figure 5.c) is a gaining of 35% on the average over Quick sort optimized CPU implementation, and finally, the performance improvement obtained using our proposed algorithm against Quick sort (Figure 5.d) is gaining on the average a 20% speedup over Quick sort optimized CPU implementation. Figure 5.d depicts the running times of sorting algorithms in milliseconds for a number of keys to be sorted ranging from 1000 to 8000 keys each key is 20 characters long.

6. CONCLUSIONS

In this paper, a new sorting 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. Besides, this algorithm was shown to be of practical interest, in join-based queries. 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

7. REFERENCES

1. 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,.
2. R.A. Krishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Shrivastava, Efficient approximation of correlated sums on data streams, *TKDE*, 2003.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd edition MIT Press, Cambridge, MA, 2001.
4. D. Knuth, Sorting and Searching, Volume 3 of the Art of Computer Programming, Addison-Wesley Publishing Company, Reading, MA, 1973.
5. A. Broder and M. Mitzenmacher, Using multiple hash functions to improve IP lookups, in *Proceedings of IEEE INFOCOM*, 2001.
6. N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi, Hardware acceleration in commercial databases: A case study of spatial operations, Proceedings of the Thirtieth international conference on Very large data bases, **30**, 1021-1032, 2004.
7. S. Manegold, P.A. Boncz, and M.L. Kersten, What happens during a join? Dissecting CPU and memory optimization effects, in *Proceedings of 26th International Conference on Very Large Data Bases*, **26**, 339–350, 2000.
8. A. LaMarca and R. E. Ladner, The influence of caches on the performance of heaps, ACM *Journal of Experimental Algorithmics*, **1**, 4-es, 1996.

9. A. Das, J. Gehrke, and M. Riedewald, Approximate join processing over data streams, in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, ACM Press, 40-51, 2003.
10. J.V. Lunteren, Searching very large routing tables in wide embedded memory, in *Proceedings of IEEE Globecom*, November 2001.
11. J.L. Bentley and R. Sedgewick, Fast algorithms for sorting and searching strings, *ACM-SIAM SODA '97*, 360–369, 1997.
12. G. Franceschini and V. Geffert. An In-Place Sorting with O(n log n) Comparisons and O(n) Moves, *IEEE FOCS '03*, 242–250, 2003.
13. L. Arge, P. Ferragina, R. Grossi, and J.S. Vitter, On sorting strings in external memory, *ACM STOC '97*, 540–548, 1997.
14. R.C. Agarwal, A super scalar sort algorithm for RISC processors, *SIGMOD Record* (ACM Special Interest Group on Management of Data), **25(2)**, 240–246, June 1996.
15. A. Arasu and G. S. Manku, Approximate counts and quantiles over sliding windows, *PODS*, 2004.
16. M.A. Bender, E.D. Demaine, and M. Farach-Colton, Cache-oblivious B-trees, *IEEE FOCS '00*, 399–409, 2000.
17. P.A. Boncz, S. Manegold, and M.L. Kersten, Database architecture optimized for the new bottleneck: Memory access, in *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, **25**, 54–65, 1999.
18. G. Franceschini, Proximity mergesort: Optimal in-place sorting in the cacheoblivious model, *ACM-SIAM SODA '04*, 284–292, 2004.
19. A. Andersson, T. Hagerup, J. H°astad, and O. Petersson, Tight bounds for searching a sorted array of strings, *SIAM Journal on Computing*, **30(5)**, 1552–1578, 2001.
20. A. LaMarca and R. Ladner, The influence of caches on the performance of sorting, in *Proc. of the ACM/SIAM SODA*, 370–379, 1997.
21. G. Franceschini, Sorting stably, in-place, with O(n log n) comparisons and O(n) moves, *STACS '05*, 2005.