J. King Saud University, Vol. 22, Comp. & Info. *Sci*., pp. 1-11,Riyadh (1431H./2010)

A Join Algorithm for Large Databases: A Quadtrees Structure Approach

Hatim A. Aboalsamh
Department of Computer Sciences, King Saud University
P.O. Box 2454 Riyadh 11451, Saudi Arabia
Email: hatim@ksu.edu.sa

(Received 20/12/2008; accepted for publication 06/05/2009)

***Keywords*:** Quadtrees, Database Systems, Relational Databases, Join Based Queries, Hash Join.

Abstract. Enhancing the performance of large database systems depends heavily on the cost of performing join operations. When two very large tables are joined, optimizing such operation is considered one of the interesting research topics to many researchers, especially when both tables, to be joined, are very large to fit in main memory. In such case, join is usually performed by any other method than hash Join algorithms. In this paper, a novel join algorithm that is based on the use of quadtrees, is introduced. Applying the proposed algorithm on two very large tables, that are too large to fit in main memory, is proven to be fast and efficient. In the proposed new algorithm, both tables are represented by a storage efficient quadtree that is designed to handle one-dimensional arrays (1-D arrays). The algorithm works on the two 1-D arrays of the two tables to perform join operations. For the new algorithm, time and space complexities are studied. Experimental studies show the efficiency and superiority of this algorithm. The proposed join algorithm requires minimum number of I/O operations and operates in main memory with O(n log (n/k)) time complexity, where k is number of key groups with same first letter, and (n/k) is much smaller than n.

1. Introduction

Time required for data transfer between main memory and external storage devices, is considered as one of the major problems that face database designers. Transferring data from/to external storage devices, where bandwidth is significantly low, to/from main memory, where bandwidth is significantly high, is the main factor that affects database systems performance [1, 10, 9]. One of the major problems that is addressed by many researchers is to minimize the amount of data written to and read from disks. Usually database tables are quite large and fitting them in main memory is not possible, even when database systems are operated on main frame computers. Since large tables do not fit in main memory, sorting must be done in at least two passes [3, 5]. 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 L1, L2 or L3 data caches [4,7]. In such cases, sorting partitions comparable to the size of main memory is not efficient. Based on that, in most database systems, we face a tradeoff between disk I/O performance and CPU computation time spent in sorting the partitions. In [3, 11], merge-based external sorting algorithms spent time in Phase 1 can be reduced by choosing run sizes comparable to the CPU cache sizes. This choice affects the time spent in Phase 2, when the merge operations are done on a large number of small runs [2, 12].

The new proposed join algorithm is based on the use of quadtrees in indexing. Each table index is represented by a 1-D quadtree tree [1]. Originally, quadtrees have been use to represent two-dimensional images. In this paper, we adopt the concept of quadtress and apply it on 1-D arrays. The utilization of 1-D quadtree to accelerate join operations on large databases will be investigated.

The rest of the paper is organized as follows: the representation of 1-D arrays using quadtrees is

depicted in section 2. The proposed technique and algorithm are described in section 3. In section 4, the time complexity analysis is presented. The performance results that represent the quadtrees and keyed quadtrees sizes, time required to build the keyed quadtree, and the FQJOIN algorithm performance are demonstrated in section 5. Conclusions and references are followed.

2. Quadtrees for 1-D Arrays

A quadtree for 1-D arrays is an adaptation of the original quadtree defined in references [6], and [8]. It is adapted to represent 1-D textual data [1].

2.1. The original quadtree

The original quadtree represents 2-d binary images by splitting the image recursively into 4 regions until each region contains only black or white pixels [13]. An example demonstrating the original quadtree is shown in Fig. 1.

1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1

Fig. 1.a. A binary image.

2.2. The quadtree for 1-D arrays

The quadtree for 1-D arrays represents 1-d binary data (binary 1-d array) [1]. The representation of 1-d data by quadtree for 1-D arrays is presented in Fig. 2.

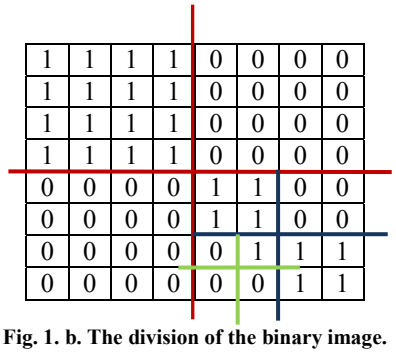


Fig. 1. b. The division of the binary image.

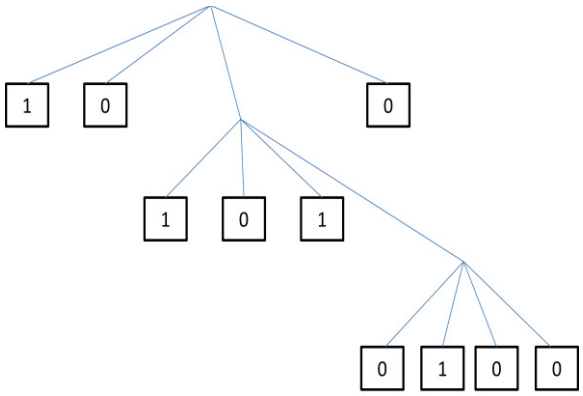


Fig.1. c. Quadtree representing image in Fig. 1.a.

Fig. 1. Representation of 2-D Binary Images.

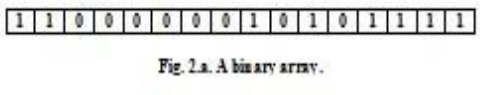


Fig. 2.a. A binary array.

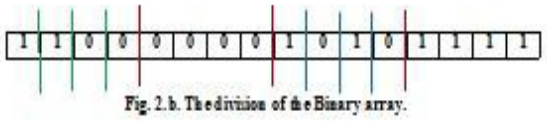
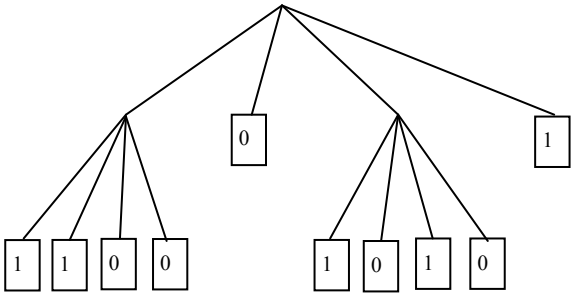


Fig. 2.b. The division of the Binary array.



1111000011101001

Fig. 2.c. The quadtree for 1-D arrays.

(binary representation is 1111000011101001)

Fig. 2. The representation of 1-d arrays by 1-D quadtrees.

An example showing how to represent non-binary array using the quadtree for 1-D arrays is studied. The formation of a 1-D tree for a non-binary 1-d array is shown in Fig. 3. For the rest of this paper, the term "quadtree" is analogous to the term "quadtree for 1-D array".

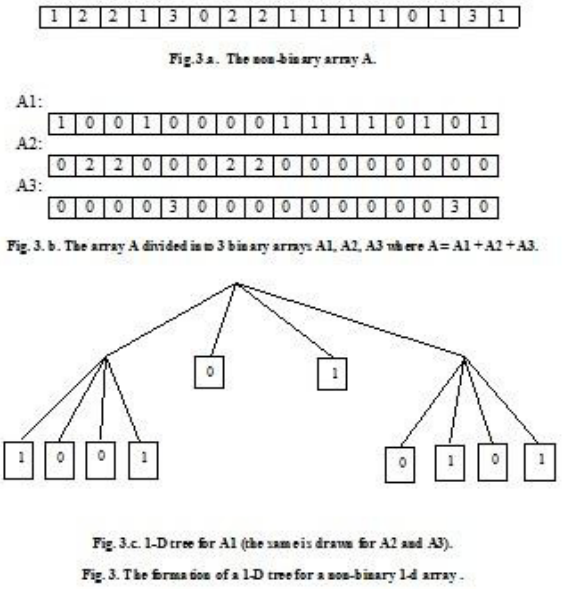


Fig.3.a. The non-binary array A.

Fig. 3. b. The array A divided into 3 binary arrays: A1, A2, A3 where A = A1 + A2 + A3.

Fig. 3.c. 1-D tree for A1 (the same is drawn for A2 and A3).

Fig. 3. The formation of a 1-D tree for a non-binary 1-d array .

The binary representation for A1 is 111001000110101. As it can be concluded from literature, quadtrees are very compact representations for very large sparse binary data [6]. Therefore decomposing non binary arrays into several binary arrays is represented in a compact manner using 1-D trees.

3. The Proposed JOIN Algorithm: FQJOIN

In this paper, our main concern is the efficient execution of the equijoin operation, which is one of the most common database operations. Several techniques have been developed for performing join operations efficiently. Among many methods, hash-based join algorithms are considered to be fast and easy-to-implement. But, joins on large datasets require high computing power and large memory space to maintain intermediate data structures (i.e., the hash table). Therefore, we approached the problem by preprocessing and building quadtree indices for the tables to be joined, which can fit in memory and perform fast and efficient join algorithms. Building the quad tree indices can be done on partitions of the tables, therefore there is no restriction even if the smallest table does not fit in main memory.

3.1 Keyed quadtrees

The proposed join algorithm is tailored especially for large databases. FQJOIN is used to join two tables R1 and R2. Neither R1 nor R2 can fit in main memory. As a matter of fact, both tables are magnitude larger than the size of main memory. FQJOIN uses quadtrees to expedite the join process. FQJOIN joins two tables R1 and R2 by building progressively quadtrees for each of them depending on the first letter of different keys. Algorithms to build quadtree for a large table R whose size is n, where n is magnitude larger than the main memory, is presented below. A keyed quadtree is built in such algorithms. The definition of a keyed quadtree is a quadtree with keys attached to it. A clarification is shown in Fig. 4.

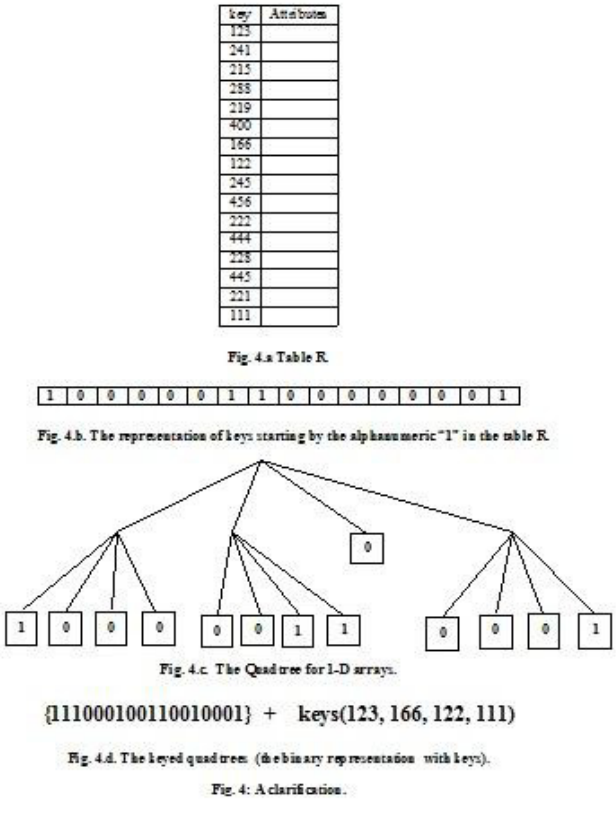


Fig. 4.a Table R.

Fig. 4.b. The representation of keys starting by the alphanumeric "1" in the table R.

Fig. 4.c. The Quadtree for 1-D arrays.

Fig. 4.d. The keyed quadtree (the binary representation with keys).

Fig. 4: A clarification.

Algorithm BUILD-QUADTREE (R , n)

{

i=1

1. repeat

{

a) Bring block B_i of R from hard disk to main memory;
(size of B_i should be less than main memory and should be in

the form 2^{4l} where l is an integer greater than zero)

b) Scan B_i and form the part P_i of the keyed quadtree for this part for each set of keys starting with the same character.

c) i=i+1

}

until i > log(n)

}

The sizes of the keyed quadtrees are very small and all of them, for the two tables, can be fitted in main memory for further processing. In section 5, we give the experimental results that show the storage requirements of the keyed quadtrees, and a comparison between the memory requirements of keyed quadtrees and ordinary join indexes.

3.2 The Joining *: FQJOIN Algorithm*

In this section, the proposed join algorithm, namely the FQJOIN algorithm, is presented and discussed. The proposed algorithm considers only equijoin queries. The algorithm may need minor modification to be applied on non-equijoin queries. The algorithm is based on using keyed quadtrees. The new technique results in enhanced time complexity. The proposed algorithm consists of two parts. In the first part, the FQJOIN-M is called k times, where k > 0, to join k pairs of quadtree groups with same first letter. In the second part, FQJOIN-IO arranges the tuples into nearby clusters that belong to the same hard disk blocks.

Algorithm FQJOIN(R1, R2)

{

For each pair of keyed quadtrees $Q1_i$ and $Q2_i$, i =1 to k, where k is the number of partitions

FQJOIN-M($Q1_i$, $Q2_i$);

FQJOIN-IO (Pos1, Pos2);

}

Algorithm FQJOIN-M($Q1_{ki}$, $Q2_{ki}$)

{j=0;

{

$S1_i$ = Sort key($Q1_i$);

$S2_i$ = Sort key($Q2_i$);

M_i($p1_i$, $p2_i$) = Merge-join ($S1_i$, $S2_i$) ;

For each tuple in M_i

{

Pos1(j) = pointer($p1_i$) in Table R1;

Pos2(j) = pointer($p2_i$) in Table R2;

j++

}

}

Algorithm FQJOIN-IO(Pos1, Pos2)

{

1. *Optimize the I/O operation by arranging the required tuples indicated by Pos1 and Pos2 into nearby clusters that belongs to the same hard disk blocks;*
2. *Perform the required I/O operations to bring tuples of all M_i's;*
3. *Display the result;*

}

4. Time Complexity

In this section, the time complexity of the new proposed joining algorithm is investigated. The performance study is being carried on to study the total performance of the algorithms both from time complexity and I/O requirements.

The new proposed joining algorithm is composed of two execution of the algorithm BUILD-QUADTREE and one execution of the FQJOIN algorithm. In its turn the FQJOIN algorithm is composed of several executions of FQJOIN-M which is performed in main memory and one execution of FQJOIN-IO which requires I/O operations. Performance study is being carried on and it studies the total performance of the algorithms both from time complexity and I/O requirements. The performance is indicated as execution time of the FQJOIN algorithm.

4.1 Time complexity analysis of build-quadtree

The process of scanning the elements of a table of n tuples, and extracting the keyed quadtrees, is done in linear time, requires O(n) comparison. It requires n/b I/O operations where b is the size of I/O block. This is indicated in equations (1) and (2).

$$T_{\text{BUILD-QUADTREE}} \; is \; \mathrm{O}(n) \qquad (1)$$

$$I/O_{\text{BUILD-QUADTREE}} = (n/b) \quad (2)$$

4.2 Time complexity analysis of FQJOIN-M

The execution time of FQJOIN-M depends on the size of the quadtree for a specific key. We assume that the first character of all keys is uniformly distributed. Therefore, all quadtrees of different keys are of the same size. Therefore, time complexity of FQJOIN-M($Q1_i$, $Q2_i$), where both $Q1_i$, $Q2_i$ are of size n/k, k is the number of different key groups with same first letter, is indicated in equation 3.

$$T_{\text{FQJOIN-M(Q1ki,Q2ki)}}\ is\ \text{O}(n/k\log(n/k)) \quad (3)$$

The explanation of equation 3 is that FQJOIN-M($Q1_i$, $Q2_i$) is a merge-join algorithm, where sorting of keys of both quadtrees (for each key) are performed first which is done in the order of $\text{O}(n/k\log(n/k))$ and then merging is done between the two sets of keys in time proportional to n/k.

The execution of the algorithm FQJOIN-M is done k times, leading to deriving equation (4) from equation (3).

$$T_{\text{FQJOIN-M}(Q1_i, Q2_i)\text{for all i}}\ is\ O(k(n/k\log(n/k)))$$
$$or\ T_{\text{FQJOIN-M}(Q1_i, Q2_i)\text{for all i}}\ is\ O(n\log(n/k)) \quad (4)$$

The algorithm FQJOIN-M is done entirely in memory and doesn't need any I/O operations this is indicated in equation (5).

$$I/O_{\text{FQJOIN-M}(Q1_i, Q2_i)\text{for all i}} = 0 \quad (5)$$

4.3 I/O requirements of FQJOIN-IO

The algorithm FQJOIN-IO is I/O bound algorithm with no significance computational load in main memory. Its I/O requirements depend on the size of the resultant joined table R_{join} and the distribution of its tuples among the different blocks. Assume that the tuples of R_{join} are distributed among m blocks. The I/O requirements are indicated in equation (6).

$$I/O_{\text{FQJOIN-IO}} = m \quad (6)$$

4.4 Time complexity analysis of the Algorithm FQJOIN(R1, R2)

The time complexity and the I/O requirements of the entire joining process are derived from equations (1) to (6) and are depicted in equations (7) and (8).

$$T_{\text{FQJOIN}}\ is\ \text{O}(n\log(n/k)+n) \quad or$$
$$T_{\text{FQJOIN}}\ is\ \text{O}(n\log(n/k)) \quad (7)$$

$$I/O_{\text{FQJOIN}} = (n/b)+m \quad (8)$$

In this paper, we compare our results with the sort-merge-join algorithm. It can be shown that our proposed join algorithm has time complexity of order less than the sort-merge join algorithms. Also, it can perform decently even if both tables are magnitude larger than main memory, which cannot be done in case of hash-based join algorithms. I/O requirements depend on two factors, the block size and the size of the resultant joined table. There is a huge room for the optimization of the times of the I/O operations by varying size of b if possible or by optimizing the I/O operations performed to display the resultant joined table.

5. Performance Results

Several experiments have been performed and analyzed to show the system performance on the following aspect:

- Quadtrees and keyed quadtrees sizes.
- The time required to build the keyed quadtree.
- The FQJOIN algorithm performance

The results are compared with ordinary join indexes to show the significance of using quadtrees in join operations.

5.1 Quadtrees and Keyed Quadtrees Sizes

In our experimental results, we have used twenty tables of sizes ranging from 10,000 tuples to 80,000 tuples for each experiment. The simulation results in Figure 5, have shown the average number of bits required to build quadtrees for different tables sizes. In figure 6, the experimental results have shown a comparison between the average number of bits required by keyed quadtree and ordinary index. The results showed that, for very large tables, the proposed keyed quadtree is using almost half of the memory required by ordinary indexes. This difference is due to the extra size needed for addresses in ordinary indexes which is replaced by the binary representation of quadtrees.

5.2 The Time Required to Build Keyed Quadtrees

In Figure 7, the time required to build the keyed quadtree and the ordinary index is presented. The experimental results have used tables of sizes ranging from 10,000 to 80,000 tuples, assuming the block size is 5000 tuples. It is clear that the quadtrees building requires O(n) time, while building ordinary indexes requires O(n log n) time.

5.3 The FQJOIN algorithm performance

Performance study is being carried on and it studies the total performance of the algorithms both from time complexity and I/O requirements. The performance is indicated as execution time of the FQJOIN algorithm. The experiments are held on different sized tables with different sized resultant joined tables. The block size is fixed to 5000 tuples in the first set of experiments in the simulation study. In the second set of experiments, the block size varies from 5000 to 40,000 to study its effect on the execution time.

The second fold of the performance study is to compare the execution time of the proposed algorithm to other join algorithms. The proposed algorithm is compared to the merge sort based Join algorithm. The sort-merge Join algorithm is studied as an alternative to join large tables, nested join algorithm and hash-based join are not suitable for large tables that neither one of them fit in main memory. The following figures show the simulation results.

The performance of join operations has been improved by applying the proposed quadtree join algorithm and compared its performance against other optimized sort-merge join algorithms. The FQJOIN algorithm is implemented and its performance is evaluated on equijoin queries. These queries are performed on two tables with variable sizes. The experiments have been repeated on different sets of tables and the average performance is calculated. The experiments showed the advantage of the quadtree join algorithms in terms of execution time. From figures 8 and 9, it is noticed that FQJOIN is 4 times faster than the sort-merge join algorithm. It is predicted that it even performs much faster in very large databases.

6. Conclusions

The performance of most DBMS and DSMS queries is dominated by the cost of Join Queries. It is very crucial to optimize join algorithms especially for large database, when both relations, to be joined, are very large to fit in main memory. In this case, usually join is performed by any other method than hash Join algorithms. In this paper, a new join algorithm for large databases has been introduced. The algorithm has been proven to be fast and efficient even if both relations to be joined are too large to fit in main memory. The new join algorithm represents both relations by a storage efficient quadtrees for 1-D arrays. Tailored mechanism has been carried out that expedites the Join operations. The new algorithm has been presented; time and space complexities have been studied. Experimental studies have shown the efficiency and superiority of this algorithm. The proposed join algorithm requires minimum number of I/O operations and operates in main memory with $O(n \log (n/k))$ time complexity, where k is number of key groups with same first letter, and (n/k) is much smaller than n. 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. In conclusion, the main contribution of this research has been the introduction of the quadtree and its application in speeding up join algorithms.

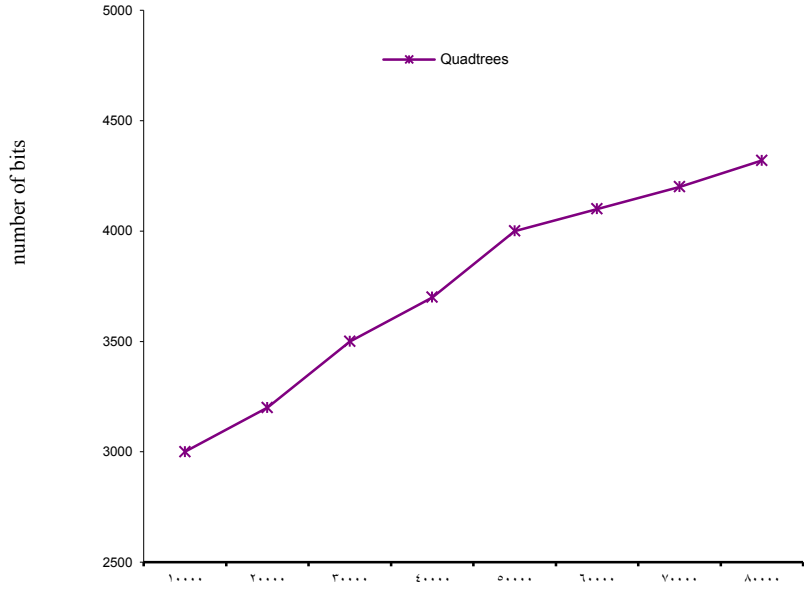


Fig. 5. Number of bits required by quadtrees

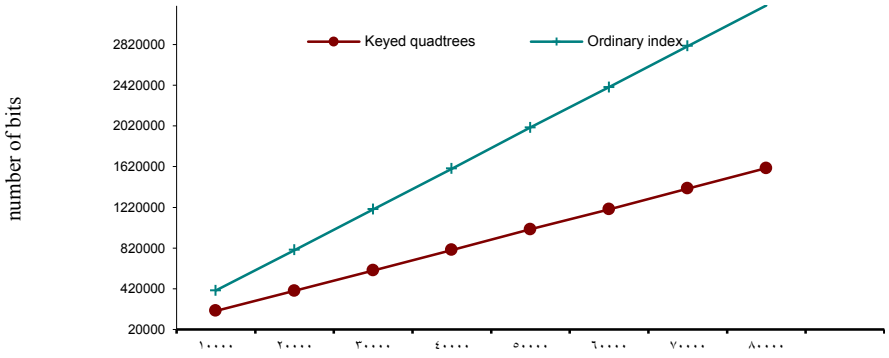


Fig. 6. The average number of bits required by keyed quadtree and ordinary index

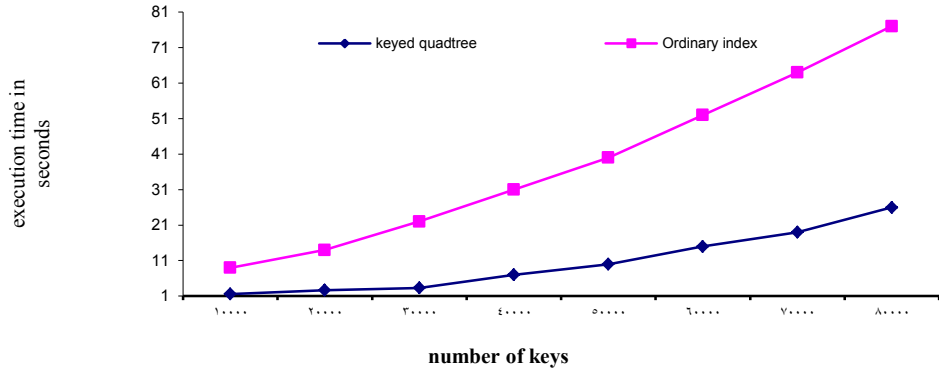


Fig. 7. Execution time to build keyed quad tree and to build ordinary index

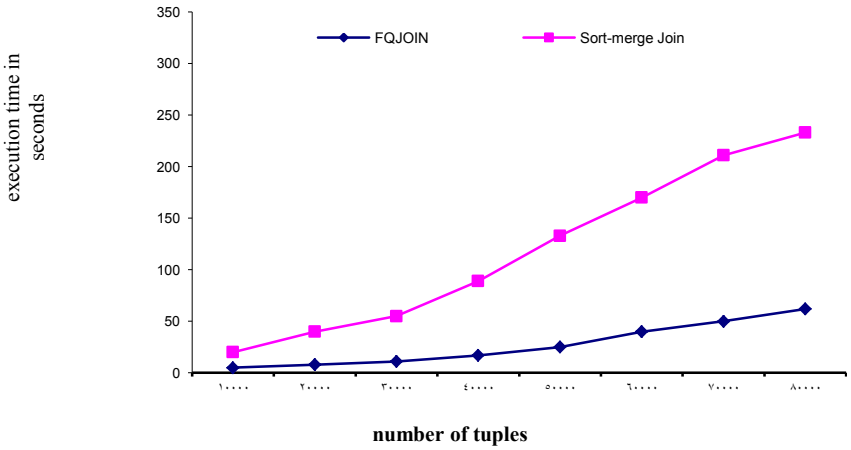


Fig. 8a .Data set 1 FQJOIN versus Sort-merge Join.

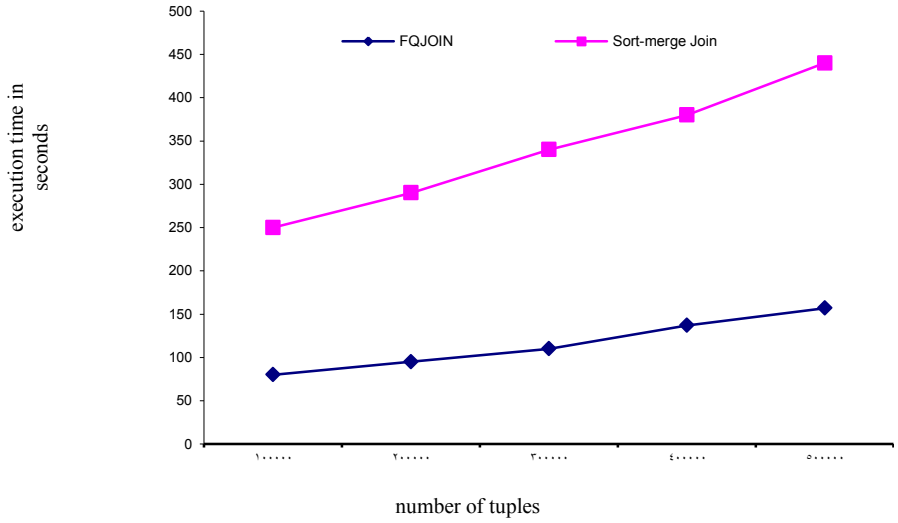


Fig. 8a .Data set 2 FQJOIN versus Sort-merge Join.

Fig. 8. FQJOIN versus Sort-merge Join for variable sized tables and fixed block size of 5000 tuples.

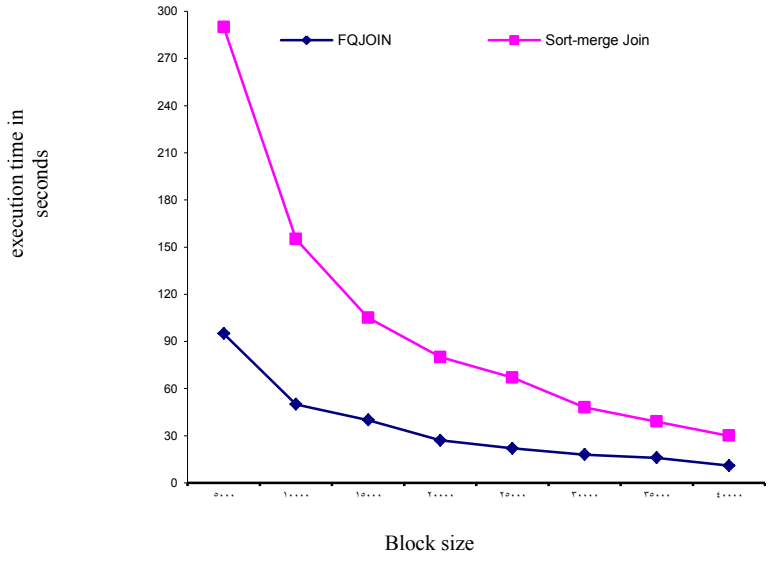


Fig. 9. FQJOIN versus Sort-merge Join with Varying block size and tables with fixed number of tuples equal to 20,000 tuples.

References

[1] H. A. Aboalsamh, "An Efficient Quadtree-Based Join Algorithm for Large Databases", Proceedings of World Academy of Science, Engineering and Technology, Vol. 38, pp 1113-1118, February 2009.

[2] N. Bandi, C. Sun, A. El-Abbadi, and D. Agrawal, "Hardware acceleration in commercial databases: A case study of spatial operations." In VLDB'04, Toronto, Canada, Aug. 2004, pp. 1021–1032.

[3] A. Broder and M. Mitzenmacher "Using multiple hash functions to improve IP lookups," in Proceedings of IEEE INFOCOM, 2001.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. MIT Press, Cambridge, MA, 2nd edition, pp 498, 2001.

[5] 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, 2003, pp.40-51.

[6] D. Eppstein, M.T. Goodrich, and J. Z. Sun "The skip quadtree: a simple dynamic data structure for multidimensional data", Proceedings of 21st Symposium of Computational Geometry, SCG '05, ACM, Jun 2005, pp. 296–305.

[7] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in SIGMOD'04, Paris, France, June 2004, pp. 215–226.

[8] G. Klajnšek, B. Žalik, F. Novak, G. Papa, "A quadtree-based progressive lossless compression technique for volumetric data sets", Journal of Information Sciences and Engineering, Vol. 24. no. 4" pp 1187-1195 (2008).

[9] D. Knuth, The Art of Computer Programming: Volume 3 / Sorting and Searching, Addison-Wesley Publishing Company, 1973.

[10] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of heaps," Journal of Experimental Algorithmics, vol. 1, Article 4, 1996.

[11] J. V. Lunteren, "Searching very large routing tables in wide embedded memory," in Proceedings of IEEE Globecom, November 2001.

[12] D. A. Schneider and D. J. DeWitt, "Tradeoffs in Processing Multi-Way Join Queries via Hashing in Multiprocessor Database Machines", In Proceedings of the 16th International Conference on Very Large Data Bases, VLDB '90.

[13] G. Ziegler, R. Dimitrov, C. Theobalt, and H. P. Seidel, "Real-time Quadtree Analysis using HistoPyramids" , in Real-Time Image Processing 2007, San Jose, USA 2007, 1-11.

خوارزمية ضم الجداول لقواعد البيانات الضخمة باستخدام هياكل البيانات من نوع المتفرعات الرباعية

حاتم بن عبدالرحمن أبو السمح

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات

ص.ب: ٢٤٥٤ الرياض١١٤٥١، المملكة العربية السعودية

hatim@ksu.edu.sa

(قدم للنشر في ٢٠٠٨/١٢/٢٠م؛ وقبل للنشر في ٢٠٠٩/٠٥/٠٦م)

ملخص البحث. تعزيز أداء نظم قواعد البيانات الكبيرة يعتمد بشكل كبير على تكلفة أداء عمليات الانضمام (Join Operations). فعندما يتم تنفيذ عملية ضم جدوليين كبيرين فإن التنفيذ المثالي لتلك العملية يعد احد مواضيع البحث التي تحوزعلى اهتمام العديد من الباحثين، خاصة عندما يكون الجدوليين المراد ضمهما كبيرين جداً ولا يمكن وضعهم في الذاكرة الرئيسية. وفي هذه الحالة، يتم تنفيذ الضم باستخدام أي أسلوب آخر غير استخدام خوارزميات البعثرة للضم. في هذه الورقة، يتم تقديم خوارزم جديد للضم يقوم علي استخدام شجرة الكواد(Quadtree). وقد تم اثبات أنه بتطبيق الخوارزم المقترح على اثنين من الجداول الكبيرة جدا والتي لا يمكن وضعها في الذاكرة الرئيسية ، فإن الضم يتم بسرعة وفعالية. وفي الخوارزم الجديدالمقترح يتم تمثيل الجدوليين المراد ضمهما بفاعلية باستخدام الـ Quadtree لتخزيين الجدوليين والتي تم تصميمها للتعامل مع المصفوفات ذات البعد الواحد او المصفوفات الاحادية (1-D arrays) لتنفيذ عملية الضم. وقد تم دراسة معدلات سرعة العمليات والمساحة اللازمة لتفيذ الخوارزم الجديد وقد اظهرت الدراسات التجريبية ان هذا الخوارزم يمتاز بالكفاءة والتفوق.وخوارزم الضم المقترح يتطلب حد ادني من عمليات الإدخال و الإخراج ويعمل في الذاكرة الرئيسية بمعدلات للزمن تتناسب مع $O(n \log (n/k))$ حيث k تمثل عدد المجموعات ذات المفاتيح والتي لها نفس الحرف الاول حيث (n/k)أصغر بكثير من n .