



**IN THE NAME OF ALLAH,
MOST GRACIOUS, MOST MERCIFUL**

A Join Algorithm for Large Databases: A Quadrees Structure Approach

Hatim A. Aboalsamh

*Department of Computer Sciences, College of Computer and Information Sciences,
King Saud University, P.O. Box 2454, Riyadh 11451, Saudi Arabia*

(Received 20/12/2008; accepted for publication 6/5/2009)

Keywords: Quadrees, 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 quadrees, 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 a minimum number of I/O operations and operates in main memory with $O(n \log(n/k))$ time complexity, where k is the number of key groups with the 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 (Aboalsamh, 2009; Knuth, 1973; LaMarca and Ladner, 1996). 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 (Broder and Mitzenmacher, 2001; Das, Gehrke and Riedewald, 2003). 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 (Cormen et al., 2001; Govindaraju et al., 2004). 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 Broder and Mitzenmacher (2001) and Lunteren (2001), 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 (Bandi et al., 2004; Schneider and DeWitt, 1990).

The new proposed join algorithm is based on the use of quadrees in indexing. Each table index is represented by a 1-D quadtree tree (Aboalsamh, 2009). Originally, quadrees have been used to represent two-dimensional images. In this paper, we adopt the concept of quadrees 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 quadrees 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 quadrees 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. Quadtree for 1-D Arrays

A quadtree for 1-D arrays is an adaptation of the original quadtree defined in Eppstein, Goodrich and Sun (2005), and Klajnšek et al. (2008). It is adapted to represent 1-D textual data (Aboalsamh, 2009).

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 (Ziegler et al., 2007). An example demonstrating the original quadtree is shown in Fig. 1.

2.2. The quadtree for 1-D arrays

The quadtree for 1-D arrays represents 1-D binary data (binary 1-D array) (Aboalsamh, 2009). The representation of 1-d data by quadtree for 1-D arrays is presented in Fig. 2.

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”.

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. 1a. A binary image.

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. 1b. The division of the binary image.

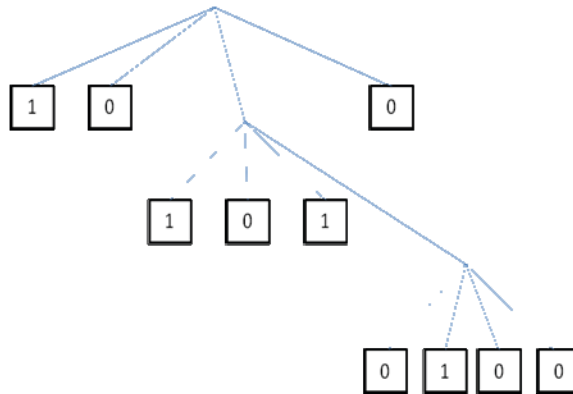


Fig. 1c. Quadtree representing image in Fig. 1a.

Fig. 1. Representation of 2-D binary images.

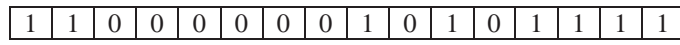


Fig. 2a. A binary array.

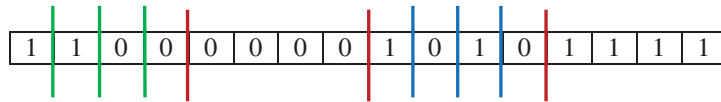
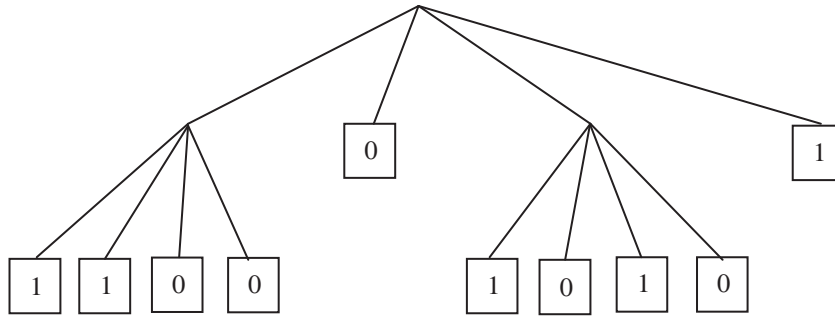


Fig. 2b. The division of the binary array.



1111000011101001

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

(binary representation is 1111000011101001)

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

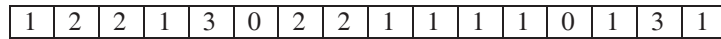


Fig. 3a. The non-binary array A.

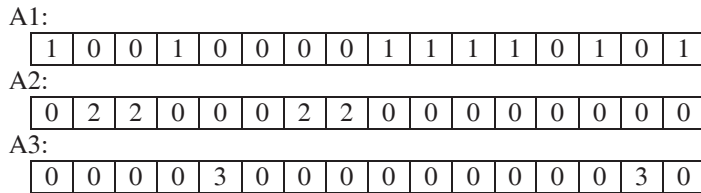


Fig. 3b. The array A divided into 3 binary arrays A1, A2, A3 where $A = A1 + A2 + A3$.

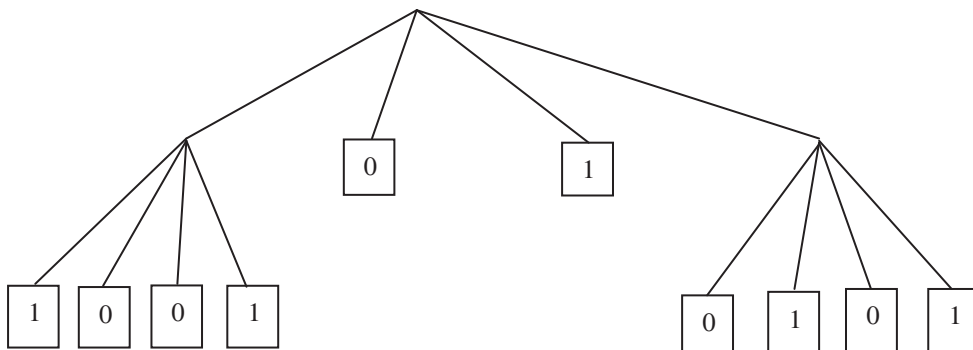


Fig. 3c. 1-D tree for A1 (the same is drawn for A2 and A3).

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, quadrees are very compact representations for very large sparse binary data (Eppstein, Goodrich and Sun, 2005). 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 quadree indices for the tables to be joined, which can fit in memory and perform fast and efficient join algorithms. Building the quadree 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 quadrees

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 quadrees to expedite the join process. FQJOIN joins two tables R1 and R2 by building

progressively quadrees for each of them depending on the first letter of different keys. Algorithms to build quadree for a large table R whose size is n, where n is magnitude larger than the main memory, is presented below. A keyed quadree is built in such algorithms. The definition of a keyed quadree is a quadree with keys attached to it. A clarification is shown in Fig. 4.

Algorithm BUILD-QUADTREE (R, n)

```

{
  i=1
  1. repeat
    {
      a) Bring block Bi of R from hard disk
         to main memory;
         (size of Bi should be less than
         main memory and should be in the
         form 2ll where l is an integer
         greater than zero)
      b) Scan Bi and form the part Pi of
         the keyed quadree 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 quadrees 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 quadrees, and a comparison between the memory requirements of keyed quadrees and ordinary join indexes.

key	Attributes
123	
241	
215	
288	
219	
400	
166	
122	
245	
456	
222	
444	
228	
445	
221	
111	

Fig. 4a Table R.

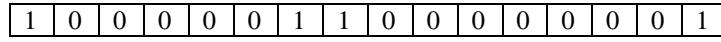


Fig. 4b. The representation of keys starting by the alphanumeric “1” in the table R.

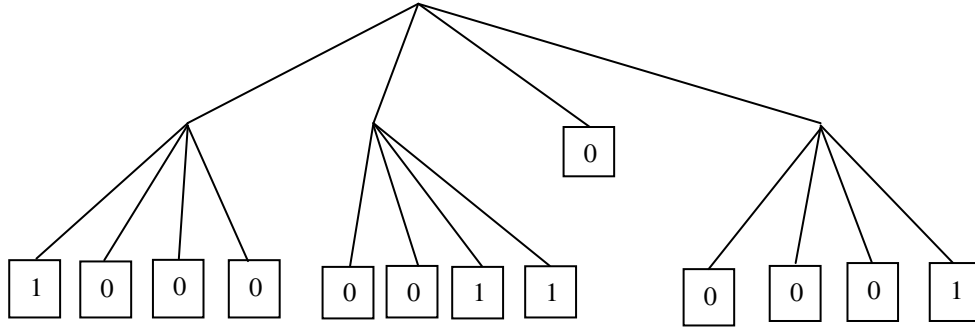


Figure 4c. The Quadtree for 1-D arrays

{111000100110010001} + keys (123, 166, 122, 111)

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

Fig. 4. A clarification.

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;
{
 $S1_i = \text{Sort key}(Q1_i)$ ;
 $S2_i = \text{Sort key}(Q2_i)$ ;
 $M_i(p1_i, p2_i) = \text{Merge-join}(S1_i, S2_i)$ ;
For each tuple in  $M_i$ 
{
 $Pos1(j) = \text{pointer}(p1_i)$  in Table R1;
```

```
 $Pos2(j) = \text{pointer}(p2_i)$  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 quadrees, 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 Eqs. (1) and (2).

$$T_{\text{BUILD-QUADTREE}} \text{ is } O(n) \quad (1)$$

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

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 quadrees of different keys are of the same size. Therefore, time complexity of FQJOIN-M(Q_{1_i} , Q_{2_i}), where both Q_{1_i} and Q_{2_i} are of size n/k , k is the number of different key groups with same first letter, is indicated in Eq. (3).

$$T_{\text{FQJOIN-M}(Q_{1_i}, Q_{2_i})} \text{ is } O(n/k \log(n/k)) \quad (3)$$

The explanation of Eq. (3) is that FQJOIN-M(Q_{1_i} , Q_{2_i}) is a merge-join algorithm, where sorting of keys of both quadrees (for each key) are performed first which is done in the order of $O(n/k \log(n/k))$, 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 Eq. (4) from Eq. (3).

$$T_{\text{FQJOIN-M}(Q_{1_i}, Q_{2_i}) \text{ for all } i} \text{ is } O(k(n/k \log(n/k))) \\ \text{or } T_{\text{FQJOIN-M}(Q_{1_i}, Q_{2_i}) \text{ for all } i} \text{ is } O(n \log(n/k)) \quad (4)$$

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

$$I/O_{\text{FQJOIN-M}(Q_{1_i}, Q_{2_i}) \text{ for all } i} = 0 \quad (5)$$

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 Eq. (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 Eqs. (1) to (6) and are depicted in Eqs. (7) and (8).

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

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

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 the 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:

- Quadrees and keyed quadrees 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 quadrees in join operations.

5.1. Quadrees and keyed quadrees sizes

In our experimental results, we have used 20 tables of sizes ranging from 10,000 tuples to 80,000 tuples for each experiment. The simulation results in Fig. 5 have shown the average number of bits required to build quadrees for different tables sizes. In Fig. 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 quadrees.

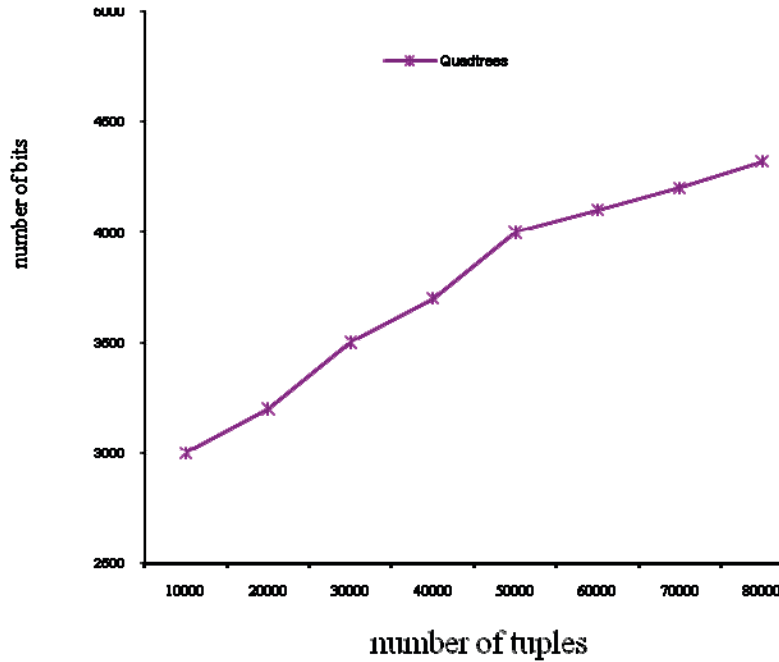


Fig. 5. Number of bits required by quadrees.

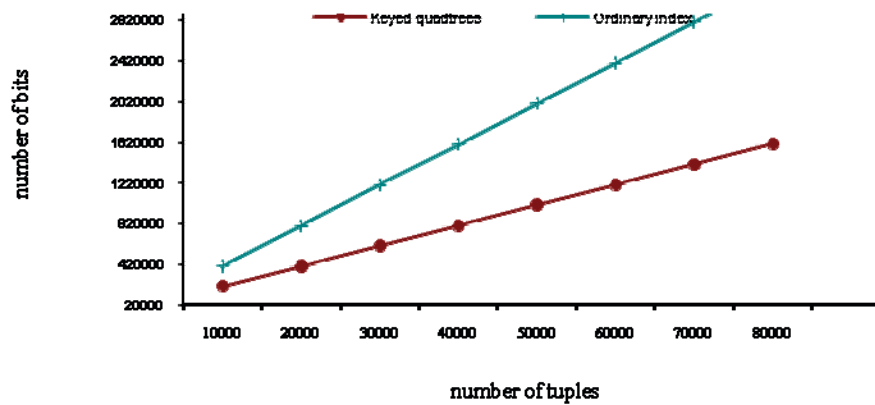


Fig. 6. The average number of bits required by keyed quadtree and ordinary index.

5.2. The time required to build keyed quadrees

In Fig. 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 5,000 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 5,000 tuples in the first set of experiments in the simulation study. In the second set of experiments, the block size varies from 5,000 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 Figs. 8 and 9, it is noticed that FQJOIN is four times faster than the sort-merge join algorithm. It is predicted that it even performs much faster in very large databases.

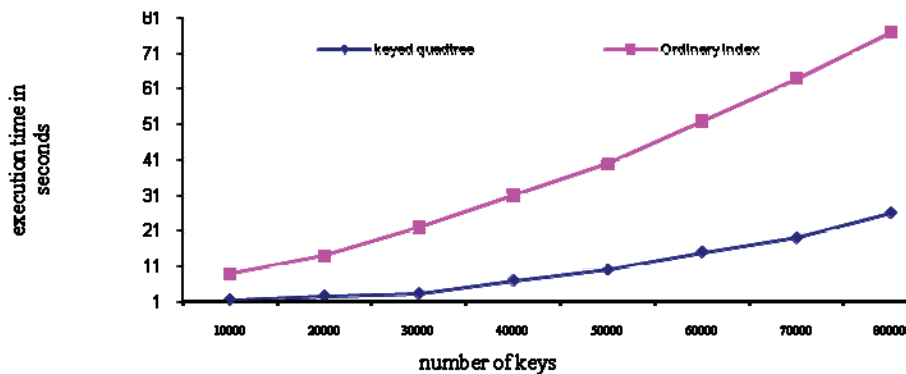


Fig. 7. Execution time to build keyed quadtree and to build ordinary index.

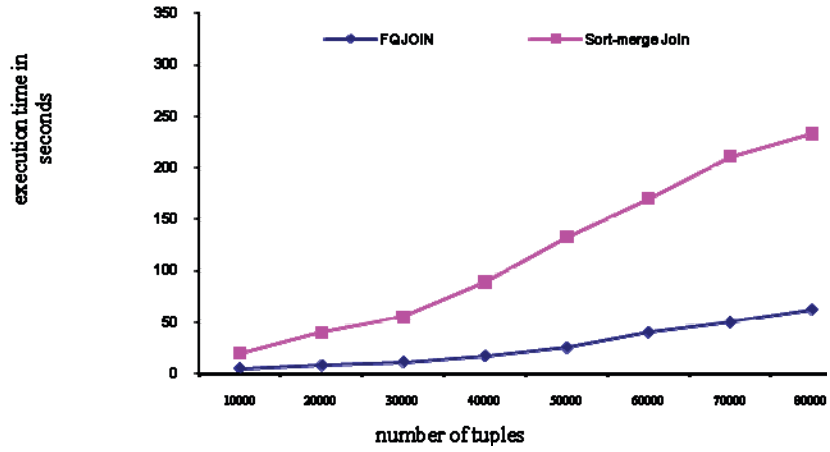


Fig. 8a. Data set 1 FQJOIN versus sort-merge join.

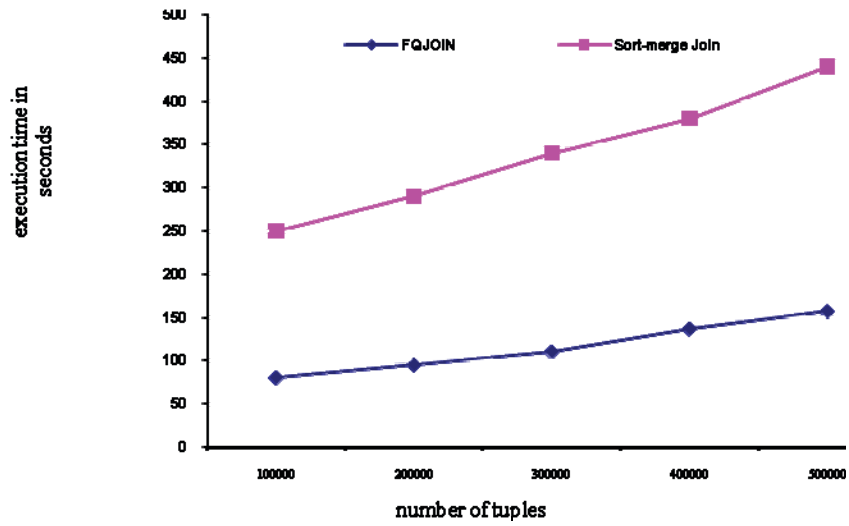


Fig. 8b. Data set 2 FQJOIN versus sort-merge join.

Fig. 8. FQJOIN versus sort-merge join for variable sized tables and fixed block size of 5,000 tuples.

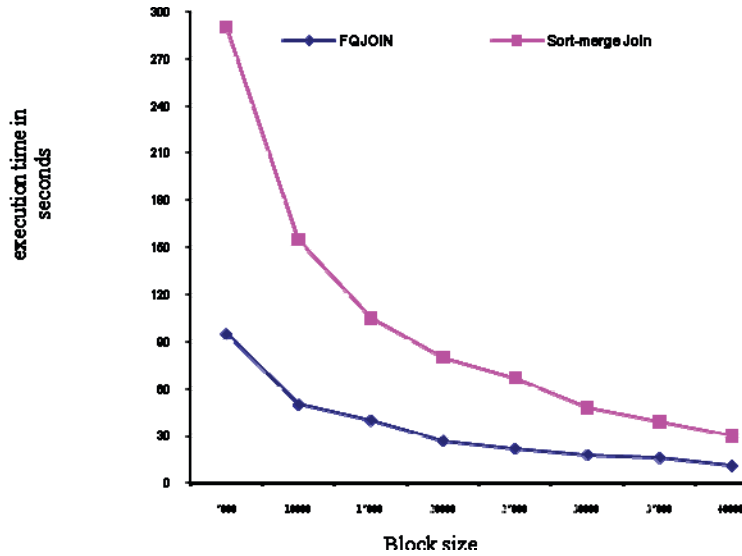


Fig. 9. FQJOIN versus sort-merge join with varying block size and tables with fixed number of tuples equal to 20,000 tuples.

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 quadrees 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 a minimum number of I/O operations and operates in main memory with $O(n \log(n/k))$ time complexity, where k is the number of key groups with same first letter, and (n/k) is much smaller than n . The 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.

References

- Aboalsamh, H. A. "An Efficient Quadtree-based Join Algorithm for Large Databases." *Proceedings of World Academy of Science, Engineering and Technology*, Vol. 38, (February 2009), 1113-1118.
- Bandi, N.; Sun, C.; El-Abbadi, A. and Agrawal, D. "Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations." *VLDB'04*, Toronto, Canada, (August 2004), 1021-1032.
- Broder, A. and Mitzenmacher, M. "Using Multiple Hash Functions to Improve IP Lookups." *Proceedings of IEEE INFOCOM*, (2001).
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. and Stein, C. *Introduction to Algorithms*. 2nd ed., Cambridge, MA: MIT Press, (2001), 498 p.
- Das, A.; Gehrke, J. and Riedewald, M. "Approximate Join Processing Over Data Streams." *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ACM Press, (2003), 40-51.
- Eppstein, D.; Goodrich, M. T. and Sun, J. Z. "The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data." *Proceedings of the 21st Symposium of Computational Geometry (SCG'05)*, ACM, (June 2005), 296-305.
- Govindaraju, N. K.; Lloyd, B.; Wang, W.; Lin, M. and Manocha, D. "Fast Computation of Database Operations Using Graphics Processors." *SIGMOD'04*, Paris, France, (June 2004), 215-226.
- Klajnšek, G.; Žalik, B.; Novak, F. and Papa, G. "A Quadtree-based Progressive Lossless Compression Technique for Volumetric Data Sets." *Journal of Information Sciences and Engineering*, Vol. 24, No. (4), (2008), 1187-1195.
- Knuth, D. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, (1973).
- LaMarca, A. and Ladner, R. E. "The Influence of Caches on the Performance of Heaps." *Journal of Experimental Algorithmics*, Vol. 1, (1996).

Lunten, J. V. "Searching Very Large Routing Tables in Wide Embedded Memory." *Proceedings of IEEE Globecom*, (November 2001).

Schneider, D. A. and DeWitt, D. J. "Tradeoffs in Processing Multi-way Join Queries via Hashing in Multiprocessor Database Machines." *Proceedings of the 16th*

International Conference on Very Large Data Bases (VLDB'90), (1990).

Ziegler, G.; Dimitrov, R.; Theobalt, C. and Seidel, H. P. "Real-time Quadtree Analysis Using Histo Pyramids." *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، جامعة الملك سعود،

ص ب ٢٤٥٤، الرياض ١١٤٥١، المملكة العربية السعودية

hatim@ksu.edu.sa

(قدم للنشر في ٢٠/١٢/٢٠٠٨ م؛ وقبل للنشر في ٦/٥/٢٠٠٩ م)

ملخص البحث. تعزيز أداء نظم قواعد البيانات الكبيرة يعتمد بشكل كبير على تكلفة أداء عمليات الانضمام (Join Operations). فعندما يتم تنفيذ عملية ضم جدولين كبيرين فإن التنفيذ المثالي لتلك العملية يعد أحد مواضيع البحث التي تحوز على اهتمام العديد من الباحثين، خاصة عندما يكون الجدولين المراد ضمهما كبيرين جداً ولا يمكن وضعهما 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 على استخدام شجرة الكواد (Quadtree). وقد تم إثبات أنه بتطبيق الخوارزم المقترح على اثنين من الجداول الكبيرة جداً والتي لا يمكن وضعهما في الذاكرة الرئيسية، فإن الضم يتم بسرعة وفعالية. وفي الخوارزم الجديد المقترح يتم تمثيل الجدولين المراد ضمهما بفاعلية باستخدام Quadtree لتخزين الجدولين والتي تم تصميمها للتعامل مع المصفوفات ذات البعد الواحد أو المصفوفات الأحادية (1-D Arrays) لتنفيذ عملية الضم. وقد تم دراسة معدلات سرعة العمليات والمساحة اللازمة لتنفيذ الخوارزم الجديد، وقد أظهرت الدراسات التجريبية أن هذا الخوارزم يمتاز بالكفاءة والتفوق. وخوارزم الضم المقترح يتطلب حداً أدنى من عمليات الإدخال والإخراج ويعمل في الذاكرة الرئيسية بمعدلات للزمن تتناسب مع $O(n \log(n/k))$ حيث k تمثل عدد المجموعات ذات المفاتيح والتي لها نفس الحرف الأول، و (n/k) أصغر بكثير من n .