International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:5, 2008

An Efficient Multi Join Algorithm Utilizing a Lattice of Double Indices

Hanan A. M. Abd Alla, and Lilac A. E. Al-Safadi

Abstract**—**In this paper, a novel multi join algorithm to join multiple relations will be introduced. The novel algorithm is based on a hashed-based join algorithm of two relations to produce a double index. This is done by scanning the two relations once. But instead of moving the records into buckets, a double index will be built. This will eliminate the collision that can happen from a complete hash algorithm. The double index will be divided into join buckets of similar categories from the two relations. The algorithm then joins buckets with similar keys to produce joined buckets. This will lead at the end to a complete join index of the two relations. without actually joining the actual relations. The time complexity required to build the join index of two categories is Om log m where m is the size of each category. Totaling time complexity to O n log m for all buckets. The join index will be used to materialize the joined relation if required. Otherwise, it will be used along with other join indices of other relations to build a lattice to be used in multi-join operations with minimal I/O requirements. The lattice of the join indices can be fitted into the main memory to reduce time complexity of the multi join algorithm.

Keywords**—**Multi join, Relation, Lattice, Join indices.

I. INTRODUCTION

THE performance of several DBMS and DSMS queries is dominated by the cost of Join Queries. Join queries are always expensive in terms of time complexity and specifically on number of I/O blocks required to be fetched. Also, Sorting is an integral component of most database management systems (DBMSs). Sorting can be both computation-intensive as well as memory intensive [1], [2], [3]. The performance of several DBMS join queries is often dominated by the cost of the sorting algorithm. Most DBMS queries will require sorting with stable results. A stable sort is a sorting algorithm which when applied two records will retain their order when sorted according to a key, even when the two keys are equal [5], [6]. Thus those two records come out in the same relative order that they were in before sorting, although their positions relative to other records may change. Stability of a sorting algorithm is a property of the sorting algorithm, not the comparison mechanism. For instance, quick Sorting algorithm is not stable while Merge Sort algorithm is stable [7], [8].

Manuscript received June 24, 2008

Hanan A. M. Abd Alla is with the Department of Information Technology, College of Computer & Information Science, King Saud University, Riyadh, Saudi Arabia (e-mail: hanan.hosni@yahoo.com).

L. A. Al-Safadi, is with the Department of Information Technology, College of Computer & Information Science, King Saud University, Riyadh, Saudi Arabia (e-mail: lsafadi@ksu.edu.sa).

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 will not fit in RAM so we must sort the data in at least two passes but two passes are enough to sort huge files [9], [10], [11]. 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 [12], [13], [14]. Therefore, it is not efficient to sort partitions comparable to the size of main memory. This 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.

Therefore in this paper, a new join algorithm is introduced that is not going to sort relations before joining them. The new algorithm is used in a novel multi-join algorithm.

The proposed algorithms have been tested in performance studies. The benchmarks used in the performance studies are databases consisting of up to 10000 values, and up to 8 relations to be multi joined.

In the following sections, the proposed algorithm, and the computational requirement analysis will be presented in details. The proposed algorithm with the proof of stability will be presented in section II. Analysis of the time complexity will be carried in section III. Experimental results are presented in section IV. Conclusions and references will follow.

II. THE PROPOSED JOIN ALGORITHM

In this paper, we present a novel algorithm to build a join index based on stable categorization algorithm. The algorithm scans each of the unsorted input relations and using a hashing function it insert an appropriate entry in the double index. Resulting into m unsorted segments Sj based on the join key K, All elements in a segment Sj have keys of the same category and this category precedes chronologically the category of all elements in segment Sj+1, with no order imposed in the segment itself. This scan requires linear time complexity. The proposed join algorithm then joins the unsorted segments of the double index to produce the join index. The stability of the keys is kept enacted, which is a requirement in database operations.

The main idea of the proposed algorithm is to divide the elements of each of the input relations into some disjoint segments S0, S1, . . . , Sm, of equal lengths L, where L is equal to n/m, such that all elements in a segment Sj is not sorted according to any order. The main condition is that all elements in a segment Sj precedes all elements in segment Sj+1. But instead of actually dividing the relation into segments which is physically expensive, we build an index using hashing function to populate it indexing the elements of the disjoint segments Si.

The New Proposed Algorithm: Build Double Index DI for two n-tuple relations R1 and R2 is presented below.

```
Algorithm Build Double Index BDI($DI_{1,2}$ , R1,R2)
Begin
    1. Scan R1 and R2 to determine the number of elements of the categories $S_0, S_1, . . . , S_m$ of R1 and categories $C_0, C_1, . . . , C_m$ of R2.

    2. Build an empty double index with 2n entries, n entries for R1 and n1 entries for R2, designate the start and end of the categories of R1 and of R2.

    3. Determine a hash function H(key), where it will determine the category of tuple t according to its key which is the join attribute.

    4. For each tuple t in R1
        a. Find category $S_j$ of t by applying the hash function H.
        b. Add an entry of the two fields which are the key and tuple number in the double index in an empty place designated to category $S_j$.

    5. For each tuple p in R2
        a. Find category $C_j$ of p by applying the hash function H.
        b. Add an entry of the two fields which are the key and tuple number in the double index in an empty place designated to category $C_j$.
End
```

There are some issues that have to be considered to guarantee stability. First, the input relation has to be scanned in sequential order from first element to last element.

Example

Assume the following two relations R1 and R2 as shown in figure 1, their double index is build according to algorithm BDI(R1,R2) using the join attribute A1. The resultant index is shown in Fig. 1.b.

R1

A1	A2	A3
41	-	-
32	-	-
43		
21		
20		
35		
34		

R2

A1	b2	b3
42	-	-
53	-	-
41		
45		
22		
26		
20		

Fig. 1.a. Relation R1 and R2

For R1 there are 4 categories S_0, S_1, S_2 and S_3. S_0 contains tuples whose category is "2" and it contains two elements. S_1 contains three elements of category 3. S_2 contains two elements of category 4. S_3 is empty.

For R2 there are 4 categories C_0, C_1, C_2 and C_3 . C_0 contains tuples whose category is "2" and it contains three elements. C_2 contains three element of category "4". C_3 contains one element of category "5".

$DI_{1,2}$(R1,R2)

A1 of R1	Tuple# in R1	A1 of R2	Tuple# in R2
21	4	22	5
20	5	26	6
32	2	20	7
35	6	42	1
34	7	41	3
41	1	45	4
43	3	53	2

Fig. 1.b The double index $DI_{1,2}$(R1,R2)

After building the double index, the corresponding categories in the DI will be joined to form the join index JI. The following algorithm will detail this technique of joining corresponding categories to form the join index.

```
Algorithm BJI($JI_{1,2}$, $DI_{1,2}$)
    Begin
    1. Scan the first corresponding categories (have similar value of the join attribute) and form the join index between them either by sorting each of the category or by using nested join.
    2. Repeat the previous step for all corresponding categories who share the same join attribute.
    End
```

Fig. 2 shows the steps of building the join index JI(DI)

S_0 and C_0

A1 of R1	Tuple# in R1	A1 of R2	Tuple# in R2
21	4	22	5
20	5	26	6
		20	7

S_1 and C_1

A1 of R1	Tuple# in R1	A1 of R2	Tuple# in R2
32	2		
35	6		
34	7		

S_2 and C_2

A1 of R1	Tuple# in R1	A1 of R2	Tuple# in R2
41	1	42	1
43	3	41	3
		45	4

S_3 and C_3

A1 of R1	Tuple # in R1	A1 of R2	Tuple # in R2
		53	2

Fig. 2.a the categories of the double index $DI_{1,2}$

$JI(DI_{1,2})$

Att1: (A1 of R1=A1 of R2)	Att2: (Tuple# in R1)	Att3: (Tuple# in R2)
20	5	7
41	1	3

Fig. 2.b. $JI_{1,2}$: join index of R1 and R2

Materializing the resultant joined relation R from the joining of two relations R1 and R2 is presented below. The following technique is going to be used to build the real joined relation in an optimized way in terms of I/O requirements. This algorithm is an attempt to minimize the number of blocks required to materialize the join operations. It should be considered that the join index is small enough to fit in main memory.

```
Algorithm Materialize(R, $JI_{1,2}$)
    Begin
    1. Use a hash function to hash the join index $JI_{1,2}$ on the
       attribute Att2.
    2. Use the results from the hash to bring tuples of R1 that
       are in the same block in one I/O operation. Populate
       the part of relation R designated for R1.
    3. Repeat step 2 for the rest of tuples of R1.
    4. Repeat step 1 using Att3.
    5. Repeat step 2 and 3 using the relation R2.
    End
```

To facilitate multi join between n relations R1 ,R2 ,…Rn, double indices DIj,j+1 between two relations Ri and Rj+1 - where j is an odd number- should be built. For n relations, n/2 DIs should be formed. A lattice between the DIs should be used to form the multi join index -MJI- between m different relations. Materializing the resultant relation R should be done at the last step using the MJI. The following example is a clarification of forming the MJI.

Example:

Assume we have 20 relations R_1, R_2, ….R_{20}, then 10 DIs will be built $DI_{1,2}$, $DI_{3,4}$, $DI_{5,6}$ ……., $DI_{19,20}$. To join the relations R_1, R_3, R_5, R_9. The lattice between the indices is shown in Fig. 3.

$$\begin{tikzcd}
DI_{1,2} \arrow[dr] & & DI_{3,4} \arrow[dl] & DI_{5,6} \arrow[dr] & & \cdots\cdots\; DI_{9,10} \arrow[dl] \\
& D_{1,3} \arrow[d] & & & D_{5,9} \arrow[d] & \\
& JI_{1,2} \arrow[dr] & & & JI_{5,9} \arrow[dll] & \\
& & MJI_{1,2,5,9} & & &
\end{tikzcd}$$

Fig. 3 The formation of $MJI_{1,2,5,9}$

If the join algorithm is 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 join, the algorithm is said to produce stable output. Otherwise, the algorithm is said to be unstable. Stability is defined such that values from relation are always given preference whenever a tie occurs between values in relation A and in relation B, and that the indices of equal set of values are kept in sorted order at the end of the join operation. The proposed multi join algorithm preserve stability, which is required from join algorithms.

III. Time Complexity and I/O Requirements

In this subsection we are going to compute the time complexity and number of I/O operations required to built a join index between two relations R1 and R2.

Scanning the elements of two relations Ri, Rj of n elements each to form DJi,j is done in linear time.

$$TDJ \; O \; n \tag{1}$$

Where TDJ is the time complexity to build a Double index between two relation each has n tuples and k blocks.

$$BDJ = 2*k \quad (2)$$

Where BDJ is the number of blocks required to be transferred from secondary storage to main memory to build a Double index between two relations each has m blocks. The physical storage for the two relations can be of great help in reducing the number of I/O required to build the DJ.

$$TJI\ O\ n \quad (3)$$
$$BJJ = 0 \quad (4)$$

Combining equation 1 and 3 will lead to equation 5 that will compute total time complexity (T) to build a join index. Also, equation 6 will combine equations 2 and 4 to present the total I/O operations (B) required to build the join index of two relations.

$$T\ O\ n \quad (5)$$
$$B = 2*k \quad (6)$$

Time complexity TR1,R2 to materialize the join of two relations R1 and R2 according to the algorithm Materialize(R, JI1,2) is computed in equation 7.

$$TR1,R2\ O\ n+n \log m \quad (7)$$

Since n log m > n then equation 7 will be reduced to equation 8.

$$TR1,R2\ O\ n \log m \quad (8)$$

Where m is the average size of each category, assuming there are n/m categories.

Time complexity (TMJI) and I/O requirements (BMJI) to build a multi join index between g relations is depicted in equations 9 and 10.

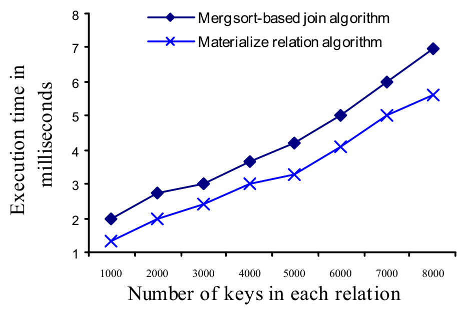
$$TMJI\ O\ n \quad (9)$$
$$BMJI = g/2*k \quad (10)$$

Time complexity Tmulti to materialize the join of g relations using multi join index is computed in equation 11. The I/O requirements (Bmulti) is depicted in equation 12.

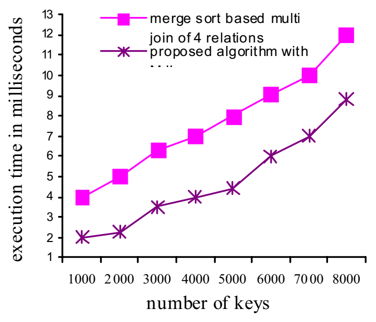
$$Tmulti\ O\ n \log m \quad (11)$$
$$Bmulti = g/2*k \quad (12)$$

IV. Performance Study

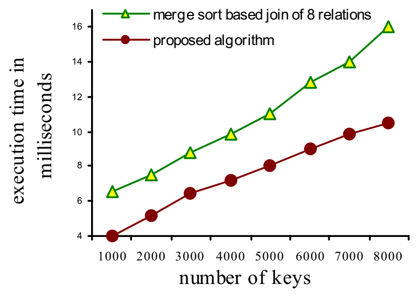
Performance study is being carried on and it comprises two folds. First fold studies the performance of the proposed algorithm to build join index between two relation R1 and R2. The second fold studies the performance of building MJI between several relations. The materialization of the join and multi join will be studies in different situations to present experimental studies showing time complexity and I/O requirements.



(a)



(b)



(c)

Fig. 4 Performance of the proposed algorithm against merge sort join algorithm

The performance improvement obtained by using our proposed algorithm against merge sort join based algorithm is observed as a factor of two times speedup over optimized CPU implementations. We performed the experiments on a 3.4 GHz Pentium IV.

V. Conclusion

In this paper, a novel multi join algorithm to join multiple relations will be introduced. The novel algorithm is based on a hashed-based join algorithm of two relations to produce a double index. This is done by scanning the two relations once. But instead of moving the records into buckets, a double index will be built. This will eliminate the collision that can happen from a complete hash algorithm. The double index will be divided into join buckets of similar categories from the two relations. The algorithm then joins buckets with similar keys to produce joined buckets. This will lead at the end to a complete join index of the two relations without actually joining the actual relations. The time complexity required to build the join index of two categories is Om log m where m is the size of each category. Totaling time complexity to O n log m for all buckets. The join index will be used to materialize the joined relation if required. Otherwise, it will be used along with other join indices of other relations to build a lattice to be used in multi-join operations with minimal I/O requirements. The lattice of the join indices can be fitted into the main memory to reduce time complexity of the multi join algorithm. Time complexity to join multi relations is in the order of n log m.

References

[1] D. Knuth, The Art of Computer Programming: Volume 3 / Sorting and Searching, Addison-Wesley Publishing Company, 1973.
[2] Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced allocations," in Proceedings of 26th ACM Symposium on the Theory of Computing, 1994, pp.593-602.
[3] Jan Van Lunteren, "Searching very large routing tables in wide embedded memory," in Proceedings of IEEE Globecom, November 2001.
[4] R. Anantha Krishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Shrivastava, "Efficient approximation of correlated sums on data streams," TKDE, 2003.
[5] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," PODS, 2004.
[6] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration in commercial databases: A case study of spatial operations," VLDB, 2004.
[7] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in Proceedings of the Twenty-fifth International Conference on Very Large Databases, 1999, pp. 54–65.
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. MIT Press, Cambridge, MA, 2nd edition, 2001.
[9] Abhinandan Das, Johannes Gehrke, and Mirek 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.
[10] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten, "What happens during a join? Dissecting CPU and memory optimization effects," in Proceedings of 26th International Conference on Very Large Data Bases, 2000, pp. 339–350.
[11] 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.
[12] L. Arge, P. Ferragina, R. Grossi, and J.S. Vitter, "On sorting strings in external memory," ACM STOC '97, 1997, pp.540–548.
[13] M.A. Bender, E.D. Demaine, andM. Farach-Colton, "Cache-oblivious B-trees," IEEE FOCS '00, 2000, pp.399–409.
[14] J.L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," ACM-SIAM SODA '97, 1997, pp.360–369.