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 $O(n \log m)$ 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].

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 $S_j$ based on the join key $K$. All elements in a segment $S_j$ have keys of the same category and this category precedes chronologically the category of all elements in segment $S_{j+1}$, 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 $S_0, S_1, \ldots, S_m$, of equal lengths $L$, where $L$ is equal to $n/m$, such that all elements in a segment $S_j$ is not sorted according to any order. The main condition is that all elements in a segment $S_j$ precedes all elements in segment $S_{j+1}$. 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 $S_i$.

The New Proposed Algorithm: Build Double Index $DI$ for two n-tuple relations $R_1$ and $R_2$ is presented below.

**Algorithm Build Double Index $BDI(DI_{1,2}, R_1, R_2)$**

*Begin*

1. Scan $R_1$ and $R_2$ to determine the number of elements of the categories $S_0, S_1, \ldots, S_m$ of $R_1$ and categories $C_0, C_1, \ldots, C_m$ of $R_2$.

2. Build an empty double index with $2n$ entries, $n$ entries for $R_1$ and $n_1$ entries for $R_2$, designate the start and end of the categories of $R_1$ and of $R_2$.

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 $R_1$
   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 $R_2$
   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 $R_1$ and $R_2$ as shown in figure 1, their double index is build according to algorithm $BDI(R_1, R_2)$ using the join attribute $A_1$. The resultant index is shown in Fig. 1.b.

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>21</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_2$</th>
<th>$A_1$</th>
<th>$b_2$</th>
<th>$b_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>45</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1.a Relation $R_1$ and $R_2$**

For $R_1$ 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 $R_2$ 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 elements of category “4”. $C_3$ contains one element of category “5”.

<table>
<thead>
<tr>
<th>$DI_{1,2}(R_1, R_2)$</th>
<th>$A_1$ of $R_1$</th>
<th>Tuple# in $R_1$</th>
<th>$A_1$ of $R_2$</th>
<th>Tuple# in $R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>4</td>
<td>22</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>26</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>20</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>6</td>
<td>42</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>7</td>
<td>41</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>45</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td>53</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1.b The double index $DI_{1,2}(R_1, R_2)$**

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 \( JI(DI) \)

<table>
<thead>
<tr>
<th>A1 of R1</th>
<th>Tuple # in R1</th>
<th>A1 of R2</th>
<th>Tuple # in R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>4</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>26</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>7</td>
</tr>
</tbody>
</table>

**S_0 and C_0**

**S_1 and C_1**

<table>
<thead>
<tr>
<th>A1 of R1</th>
<th>Tuple # in R1</th>
<th>A1 of R2</th>
<th>Tuple # in R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**S_2 and C_2**

<table>
<thead>
<tr>
<th>A1 of R1</th>
<th>Tuple # in R1</th>
<th>A1 of R2</th>
<th>Tuple # in R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>1</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>45</td>
<td>4</td>
</tr>
</tbody>
</table>

**S_3 and C_3**

<table>
<thead>
<tr>
<th>A1 of R1</th>
<th>Tuple # in R1</th>
<th>A1 of R2</th>
<th>Tuple # in R2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>53</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 2.a the categories of the double index \( DI_{1,2} \)

<table>
<thead>
<tr>
<th>Att1: ( (A1 \text{ of } R1 = A1 \text{ of } R2) )</th>
<th>Att2: ( (\text{Tuple# in } R1) )</th>
<th>Att3: ( (\text{Tuple# in } R2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>( ? )</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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.

To facilitate multi join between \( n \) relations \( R_1, R_2, \ldots, R_n \), double indices \( DI_{j,j+1} \) between two relations \( R_i \) and \( R_{j+1} \) - 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 \( R_1, R_2, \ldots, R_{20} \), then 10 DIs will be built \( DI_{1,2}, \ DIs_{3,4}, \ DIs_{5,6}, \ldots, DI_{19,20} \). To join the relations \( R_1, R_3, R_5, R_9 \). The lattice between the indices is shown in Fig. 3.

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.

\begin{equation}
TDJ O n
\end{equation}
Where TDJ is the time complexity to build a Double index between two relations each has n tuples and k blocks.

\[ BDJ = 2k \]

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.

\[ TJI \leq n \]

\[ BJI = 0 \]

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.

\[ T = 2n \]

\[ B = 2k \]

Time complexity TR1,R2 to materialize the join of two relations R1 and R2 according to the algorithm Materialize(R, J1I,2) is computed in equation 7.

\[ TR1,R2 \leq n + n \log m \]

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

\[ TR1,R2 \leq n \log m \]

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.

\[ TMJI \leq g/2k \]

\[ BMJI = g/2k \]

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.

\[ Tmulti \leq g/2k \]

\[ Bmulti = g/2k \]

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 MJJI 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.

![Graph](a)

![Graph](b)

![Graph](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 $O(m \log m)$ 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