Journal of Theoretical and Applied Information Technology
10th February 2014. Vol. 60 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

MAXIMIZED RESULT RATE JOIN ALGORITHM

[1]HEMALATHA GUNASEKARAN, [2]THANUSHKODI K
[1]Research Scholar, Anna University, India
[2]Director, Akshaya College of Engineering and Technology, India
E-mail: [1]hemalatha2107@gmail.com, [2]thanush13@gmail.com

ABSTRACT

A large number of interactive queries are being executed day by day. The user expects for an answer without no time after the execution. Even in scientific executions the user needs the intial query results for analysis without waiting for the entire process to complete. The state-of-art join algorithms are not ideal for this settings as most of the algorithms are hash/sort based algorithms, which requires some pre-work before it can produce the results. We propose a new join algorithm, Maximized Result Rate Join Algorithm (MRR), which produces the first few results without much delay. It also produces the maximum join query results during the early stage of the join operation; this is achieved by exploiting the histogram, which is available in database statistics. Histogram provides the frequency of the attribute in a table. The tuples which have high frequency of occurrences are joined during the early stages of the join operation. Further using the histogram, the join operation can be terminated when the required matching tuples are obtained. This improves the overall join performance. Experiment results shows that the new MRR join algorithm produces 60% more resultant tuples than the hash and sort-merge join algorithms. It also produces the result 30-35% early than the traditional join algorithms.

Keywords: *Join Query Optimization; Early Result Rate; Maximized Query Result; Histogram; Query Optimization.*

1. INTRODUCTION

The primary concern of a join algorithm (other than producing an accurate join) is to produce the result quickly by efficiently using the available memory [1]. Traditional join algorithms do not consider the memory limitations and it is optimized to produce the entire joining results. But the internet users will be interested to see first few results without any delay. Thus traditional join algorithms are not ideal for this type of settings [6]. All the recent algorithms which produce result early before it reads the entire input have been proposed based on sorting and hashing. As we know sorting and hashing requires more system resources (memory and CPU time), these algorithms are more suitable only when there is a network latency, delay or source blocking. In such joining algorithms delay in producing the joining result is matched with the delay in the arrival of the input tulles [1]. But these algorithms perform poorly for predictable inputs in centralized database join processing.

In our contribution, we concentrated on achieving the following three objectives:

1. Producing results early without any major pre-work.
2. Maximize the result rate of the join query during the early stage of the join operation.
3. Terminating the join operation once the no. of matching tuples are found.

The organization of this paper is as follows. In section 2, I described previous and related work. In section 3 I also have described the outline of histogram and the implementation details of the MRR join algorithm. In section 4 I discussed the experimental results, and finally with conclusion and references.

2. RELATED WORKS

In this section, we give a brief overview of the join algorithms. The first three join algorithm: Nested Loop Join, Sort Merge Join and Hash Join are the traditional join algorithms, which are followed by new early hash based join algorithms.

The Nested Loop Join In a nested loop join each row of the outer table will be compared with every row of the inner tuple. The comparison in nested loop join is the cross product of the inner and the outer table.

The Sort Merge Join In sort merge join algorithm,

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the join attribute column of the inner and the outer relation are first sorted. The sorted rows of the outer table are compared with the every row of the inner table. When there is a mismatch the outer row is incremented by one, this process is continued until all the rows of the outer table are processed.

The Hash Join In hash join algorithm the smaller relation is selected as the build relation and the other relation is selected as the probe relation. An in-memory hash table is constructed for the build relation; a hash function is selected and applied to the join attribute value of a tuple. Based on the hash value of the tuple, it is distributed in to different buckets. The same hash function is applied to the inner table and the tuples which map to the same buckets are joined.

Early Hash Join [1] Early hash join algorithm works in two modes. In one mode the algorithm is optimized to produce the join results earlier, this happens when there is enough memory available for the join operation. In another mode the algorithm is optimized to reduce the overall execution time, this is done when the memory is full. The algorithm has different reading and flushing policy which can be customized according to the need of the applications.

Hybrid Hash Join [2] In hybrid hash join, the hash function is applied to the smaller relation such that the hash table for each partition of the smaller relation fits into the memory. Now both the outer and the inner table is probed to the hash table in the memory and the corresponding buckets are joined to produce the resultant tuples.

Dynamic Hash Join [3] In hybrid hash join the partition size are pre determined before the join. In the case of dynamic hash join the partition size are varied during the execution. In dynamic hash join as much as inner relation is retained in the memory. When the memory is full, buckets are selected at random to be flushed in to the disk. This process is repeated until the entire inner relation is partitioned. There will be some partitions of the inner relation still available in the memory. Now the outer relation is read to the memory and partitioned. The tuples which falls to the partition of the inner table which is available in the memory is joined to produce the result.

HistoJoin [3] In dynamic hash join the partitions are selected at random to be flushed to disk. It does not consider skew in the join relation. But histojoin exploits the skew distribution of the data to improve the performance of the join operation. In histojoin the tuples which produce more join results are identified by exploiting the histogram and are retained in the memory without flushing to the disk. This algorithm maximizes the join result rate during the early join operation and also reduces the I/O cost.

3. MAXIMIZED RESULT RATE JOIN

3. 1 Introduction to Histogram

A histogram holds the data distribution of values within a column of a table. It holds the number of occurrences for a specific value/range. This histogram is mainly used by CBO to optimize a query. There are two types of histogram:

- Frequency Histogram
- Height-Balanced Histogram

To create a histogram we need to specify the no. of buckets. This number of buckets controls the type of the histogram created. If the distinct value of a column is less than 254 then frequency histogram is created otherwise height-balanced histogram is created.

3.1.1Frequency Histogram [4]

1
1
2
3
3
3
3

VALUE	EP	COUNT
1	2	2
2	3	1
3	7	4

Figure 1 Frequency Histogram

In frequency histogram each value of the column corresponds to a single bucket of the histogram. Each bucket will contain the frequency of that single value. In order to build a frequency histogram (FH) the size N must be <=254, that is FH can be collected only if the column has num_distinct <=254. EP and the Value in Fig 1 are from dba_histograms. The Count derived by computing the difference between the current value and the previous one.

Count (1) = 2 – 0 = 2

Count (2) = 3 – 2 = 1

Count (3) = 7 – 3 = 4

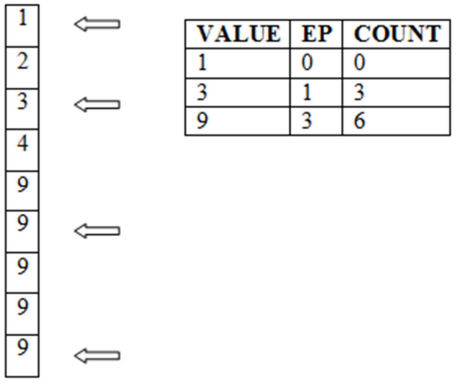
3.1.2 Height-balanced histogram [5]

In height-balanced histogram, the column values are divided into bands so that each band contains

Journal of Theoretical and Applied Information Technology
10th February 2014. Vol. 60 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

approximately the same number of rows. If the no. of distinct value of the Join column is very large ie., greater than 254, height-balanced histogram will be created. To generate height balanced histogram, the rows must be sorted by the join column value. The rows are filled into the buckets, "lapping" into the next bucket as each one fills. Thus a bucket can represent rows for many columns values, or the rows for one column values can spill into several buckets. The column value that spills into several buckets is called the popular values.



VALUE	EP	COUNT
1	0	0
3	1	3
9	3	6

Fig 2 Height Balanced Histogram

EP and Value are from dba_histograms. EP is zero for the first value sampled and indicates that this is the EPth sample.

$$\text{Count (i)} = \frac{\text{num\_rows} * \text{diff\_ep(i)}}{\text{max\_ep}}$$

max_ep = max (EP)

diff_ep (i) = current (EP) – previous (EP)

3.2 Frame Work of MRR Join

The Fig 3 shows the frame work of the MRR join algorithm. This algorithm will be more efficient when there is a limitation in the memory to perform the join operation. Let M be the memory available to perform the join operation. Based on the available memory the window size W is decided. Let R and S are the source relations to be joined. If |R|+|S| > M, then the entire input relations cannot be brought to the memory to perform the join operation. As the objective of the MRR join algorithm is to maximize the resultant rate during the early stages of the join operation, we need to determine which tuples from R and S can be brought to the memory to perform the join operation. The histogram is generated for the relation S on the join attribute column and sorted in the descending order of the frequency of the buckets. For example if the window size W is 3 then top 3 join attribute values is read from histogram and given as input to the read operator. The read operator just read the top 3 tuples which has the maximum join result from R in to the memory to perform join operation. Now the matching tuples from S has to be brought to memory to perform the join operation. To find the matching tuples the Histojoin [3] performs two methods Binary search [3] and bit array method [3] which will add overhead to the join operation.

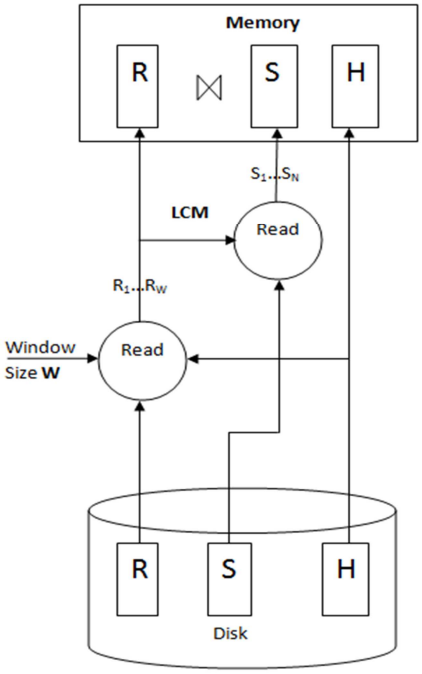


Figure 3 Frame Work of MRR Join Algorithm

The MRR algorithm uses a simple concept of least common divisor to retrieve the matching tuples from S. Lcm is calculated for the join attribute values in the window W and the tuples whose join attribute value is completely divisible by the calculated LCM is read from S. If there is an in-sufficient memory to hold the tuples of S then, only the part of the matching tuples from S which can be accommodated in memory are read to perform the join operation. Once the required tuples are brought to the memory, the simplest join nested loop join is used to perform the join operation. During the second phase of the join operation the remaining tuples which are left over during the previous phase are brought to the memory. During the join process when the sufficient matching tuples are found the join operation is terminated, without further comparison, as it will not yield any resultant tuples.

Journal of Theoretical and Applied Information Technology
10th February 2014. Vol. 60 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3.3 Steps in MRR Join Algorithm

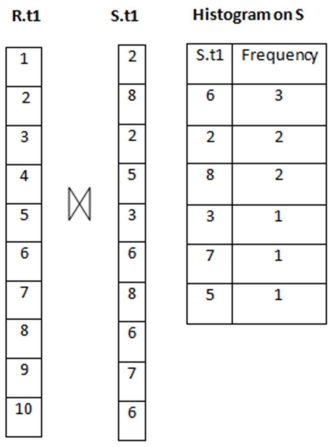


Figure 4 Join Operations on R and S

Case 1: No Memory Overflow

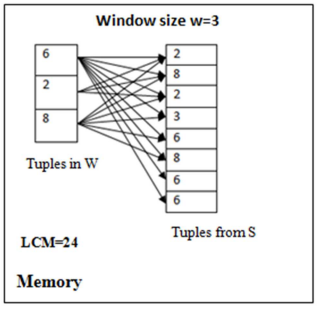


Fig 5.1 Join Phase I

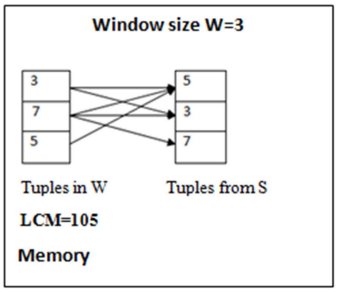


Fig 5.2 Join Phase II

Case 2: Memory Overflow

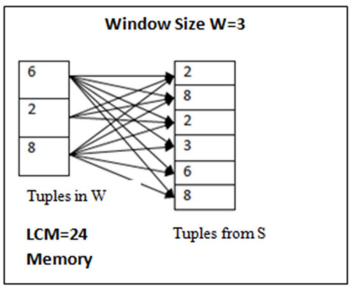


Fig 5.3 Join Phase 1

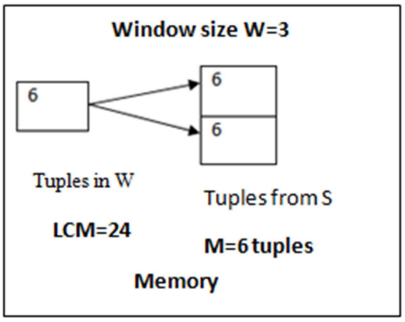


Fig 5.4 Join Phase II

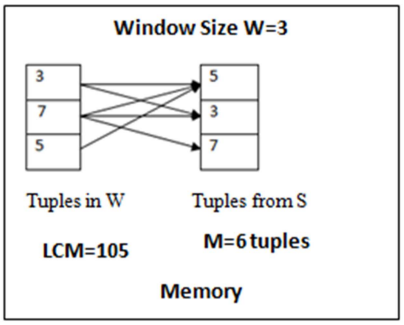


Fig 5.5 Join Phase III

Figure 5 Steps of MRR Join Algorithm

The maximized result rate join algorithm produces the result earlier and it produces the maximum join results during the earlier stage of the join operation. This algorithm is designed to perform one-to-many join as this is the one of the most common type of join that occurs in real-time applications. Consider a primary-to-foreign key join between R (A) and S (B, A) on A, where R is the smaller relation and A is the join attribute. Histogram is collected for the relation S on attribute A. If the distinct value of attribute A is less than 254 then frequency histogram (FH) will be generated, where each bucket value is the join attribute value and the frequency of the bucket is the frequencies of the join attribute value. The histogram is sorted in decreasing order of the frequency of the bucket.

Consider a join memory size of M tuples. If there is a memory constraint then both the join relations R and S cannot be accommodated in the available memory most of the time. So which tuples from R and S can be joined first to produce maximized join result has to be determined. To achieve the maximized join result during the early stage of join the tuple which has the maximum frequency of occurs in the table S is joined first. But how many such tuples can be brought to the memory for join is another major question. So a window of size W is fixed. For example if W is 3,

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

then top three tuples which has the highest frequency in the histogram is read from the table R into the window. Smaller values are selected as window size W; this is to give more space for the matching tuples to accommodate. If the memory is still free after accommodating the tuples from W and the matching tulles from S. The window size can be increased when the user need the result in no time, it is always better to select smaller values as window size. If the user can accommodate the delay in producing the result but needs the memory to be used to its maximum then the window size can be increased. Let $R_1...R_W$ be the tuples which has the maximum join tuples in table S. Now a range checking is performed for each tuple in window W to find the matching tuple in table S. In histojoin algorithm [3], range check is implemented in two methods. The first methods sorts the ranges and performs a binary search [3] using a given value to find if it is any of the ranges. The second approach for integer values uses a bit array [3]. This method requires hashing the input value and check if the bit is set in the bit array. To perform the binary search method sorting is required. To perform bit array method hashing the input value and checking if the bit is set in the bit array is required. This may delay the production of resultant tuple.

In MRR join algorithm we propose a simplified range checking method using the concept of least common divisor. LCM ($R_1...R_W$) is calculated, the tuples which is completely divisible by LCM from table S (Let S_1, S_2 .... S_N be the matching tuples from S for the tuples $R_1...R_W$) is read to the memory to join, if the condition $|R_1 ... R_W| + |S_1, S_{2....} S_N| <= M$ holds, else the the matching tuples from S is read to memory until the memory overflows and the join operation is initiated. Once the join is completed the memory is flushed out and the remaining matching tuples from S which couldn't accommodate in memory in the previous phase are brought to memory to join. Simple nested loop join is performed between the tuples in the memory to perform the join operation. The number of comparison required to produce the phase 1 join result is $|R_1...R_W|\times|S_1,S_2....S_N|$. But MRR join algorithm terminates the join operation once the required matching tuples are found with the help of histogram. So the no.of comparison will be $< |R_1...R_W|*|S_1, S2....S_N|$. This feature will differentiate MRR join algorithm with the histojoin and it produces the maximum resultant tuples with less delay than the Histojoin.

In the case of height balanaced histogram, the values that are sampled more than once are the popular values. In Fig 2 VALUE=9 is sampled twice, but it is reported only once in the histogram, but the EP is increased by 2, not by one. The EP will be increased by k times if it is sampled k times. So the values which are sampled more number of times are selected as the top values from the histogram. Those values are read to the memory to join first.

Consider the Fig 4.1; it includes a Window W of size 3 and a memory M of size 10 tuples. The histogram on table S on the join attribute t1 is shown in the figure 3. The histogram is arranged in the descending order of the frequency. Based on the size of the window, the top three join attribute values are selected from the histogram ie., 6, 2 and 8. The matching tuples from R which has the maximum frequency in the histogram is read to the window W to join. All the tuples which fall in the range of the tuples in Window W is read from table S. The range check is done using the concept of LCM. LCM is calculated for the tuples in windows (6, 2 and 8) and the LCM is 24. All the tuples from R which is completely divisible by the LCM are read from R and brought to the memory to join with the tuples in the window W. If the size of the memory is limited as in fig 4.3 (Memory size = 6 tuples) there is no enough space to accommodate all the tuples in range in the memory, so the join happens in two phases. When the first phase is completed, the memory is cleared and the remaining matching tuples are brought to the memory to join with the tuples in window W. During the join phase the number of matches for each join attribute can be determined from the histogram and the join operation can be terminated when the required matching tuples are found. For example in Fig 4.3, the join operation for the join attribute 2 is terminated without comparing with all the tuples in memory as the no. of required matches are found. The no. of required matches can be determined from the histogram.

3.4 Algorithm

Input: R_A Outer table, R_B Inner table, K Distinct joins keys in R_B, Histogram (which contains the frequency of the join keys in R_B arranged in increasing order of the frequency of the join key), Lcm least common multiplier, W window size, M Join memory size.
Output: Join result for $R_A \bowtie R_B$

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
While (RA.size()>0)   // Until there are elements in RA
{
    Buffer.clear();
    Window.clear();
    for (i=1; i<W; i++)
    {
        Buffer.add(Histogram.get(i));
          RA.delete(Histogram.get(i));
    }
// calculate the lcm for the elements in the buffer
Lcm = lcm (Histogram);
for(k=1;k<RB.size();k++)
{
      Mode=LCM % (Integer)RB.get(k);

 If(mode==0) & (Window.size()<M)
      Window.add(RB.get(k));
      Else
          {
              Print ("Memory overflow")
              Remain=k;
              Break;
          }
//now join the tuples in buffer and the window
Case 1:
//nested loop join between RA and RB
for (i=0;i<Buffer.size();i++)
 for (j=0;j<window.size();j++)
//Checks if the required matches are found
   If(Buffer.get(i).frequency(i)<match)
     If(window.get(i)==Buffer.get(i))
       Print("Match Found")
       Match++;
       Else
       Print("No Matching")
   Else
       {
       Break case 1;
       }
}
```

4. EXPERIMENTAL VALIDATION

We ran the experiments on an Intel® corei5 2.50GHz processor, with 4GB real memory, running windows7 and Java 1.7.0_09. We have written a PL/SQL procedure to populate two tables Project(Project_No, Project_Name, Location) with 250 rows and Employee (EmpNo, ProjectNo) with 1,00,000 rows respectively. Project_No from Project table is the primary key and the ProjectNo from the Employee table is the reference key. Assuming this algorithm suits best when one-to-many relationship holds between the relations. The reference key column values are generated using random generation method. The foreign key table was loaded with 1,00,000, 2,00,000 and 3,00,000 rows respectively and the performance of the join operation is compared. As shown in Fig 6 the number of comparisons of the MRR join is 4%-5% more than the hash and sort merge join but it is promisingly better than the nested loop join. Even though the comparison of the MRR join is more than the hash join and the sort merge join the rate of the resultant tuples produced by the MRR join is more during the initial join phases.

Figure 6 compares the performance of the MRR join algorithm with the traditional join algorithms. We see that the no. of comparisons of MRR join algorithm is less when compared to the nested loop join but it is 4%-6% more than the hash join and sort merge join. But in Figure 7 and Figure 8 we find that the rate of the resultant tuples generated by the MRR join is 60% more than the nested loop, hash and sort merge join. Figure 9 compares the time taken by the join algorithms to produce the first tuple. This figure shows the pre-work required by each algorithm before it could produce the first resultant tuple. Hash join and sort merge join requires hashing and sorting of the join attribute value before it can produce the resultant tuples. Nested loop join does not require any pre-work to produce the join results, but it cannot be preferred in many situations as it requires more comparisons. MRR join produces 60% more tuples during the earlier stages of the join when compared to the traditional join methods. It also produces 20% early join results when compared to sort merge join and 60% early when compared to hash join. We have just compared our join algorithm only with traditional join algorithm as all the recent join algorithms are mostly based on sort and hash based join algorithm.

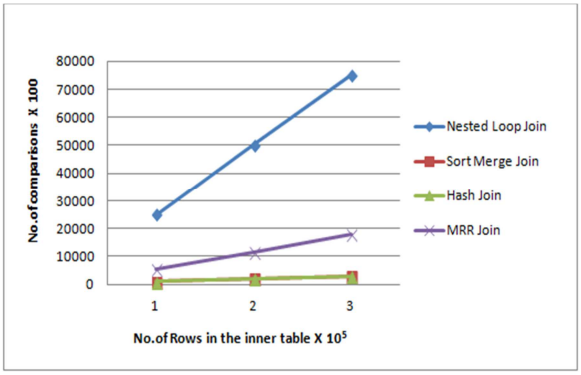


Figure 6 Performance of MRR Join Vs Traditional Join Algorithm

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

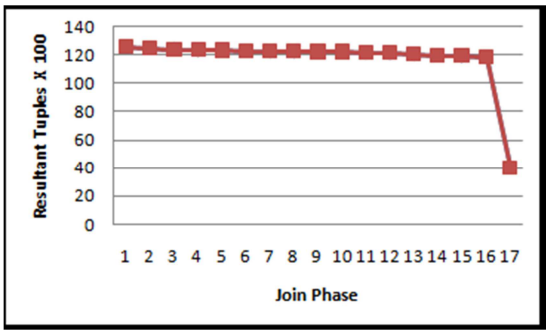


Figure 7 Rate Of The Resultant Tuples Produced By MRR Join

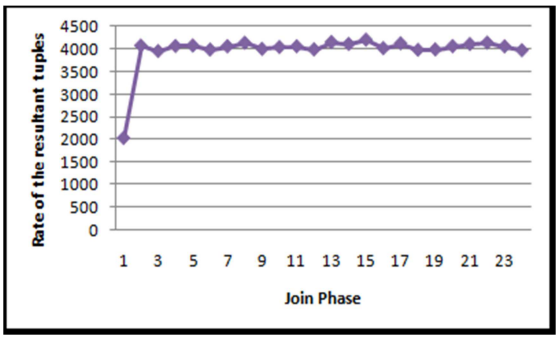


Figure 8 Rate Of Resultant Tuples Produced By Traditional Join Methods

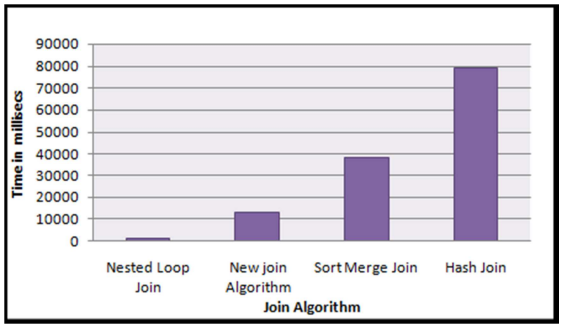


Figure 9 Time Taken To Produce The First Resultant Tuple

5. CONCLUSION

The MRR join algorithm can be used to produce earlier and maximized join results. The MRR join algorithm can be used when we need to perform a join in limited available memory. In the limited available memory MRR join algorithm promises maximized early join results. This is achieved with the help of histograms. The rows which will produce the maximum join results are identified in the inner table and are joined during the earlier stages of the join operation. MRR join algorithm does not require hashing or sorting this reduces the memory and I/O overhead. It applies a concept of LCM to find the matching tuples to bring to the memory for the join operation. The algorithm result shows that it requires 4%-5% more comparison than the hash and sort-merge based join algorithms but it is produces 60% more tuples in the earlier stages of the join operation and the delay in producing the first tuples are 30-35% less in compared to hash and sort merge join algorithms. In MRR join algorithm the join operation is terminated when the required matching tuples are found. This reduces the no.of comparisons required to produce the join result, which in turn reduces the time and I/O overhead.

REFERENCES:

[1] Ramon Lawerence, "Early Hash Join: A configurable Algorithm for the Efficient and Early Production of Join Results", *Proceedings of 31st VLDB Conference*, Trondheim, Norway, 2005 pages 841-852.

[2] M. Mokbel, M. Lu, and W. Aref, "Hash-Merge Join: A Nonblocking Join Algorithm for Producing Fast and Early Join Results", *In Proc ICDE 2004*, pages 251–263.

[3] B. Cutt and R. Lawrence, "Improving Join performance for Skewed Databases" *in Proc IEEE Canadian Conference of Electrical and Computer Engineering*, Canada, May 2008, pages 387-391.

[4] Wolfgang Breitling, "Join, skew and Histograms", *Hotsos Conference*, 2007.

[5] Wolfgang Breitling, Histogram – Myth and Facts, *Hostsos Conference*, 2005.

[6] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation", In *SIGMOD*, pages 287–298.

[7] J.P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive Merge Join: A Generic and Non-blocking Sortbased Join Algorithm", *In Proceedings of the International Conference on Very Large Data Bases, VLDB*, Hong Kong, Aug. 2002, pages 299– 310.

[8] Gang Luo, Curt J. Ellmann, Peter J. Haas, Jeffrey F. Naughton, "A Scalable Hash Ripple Join Algorithm", *in Proc. ACM SIGMOD*, Wisconsin, USA , June 4-6 2002.

[9] T. Urhan and M. Franklin, "XJoin: A Reactively Scheduled Pipelined Join Operator", *IEEE Data Engineering Bulletin*, 2000 23(2):7–18.

[10] Stratis D. Viglas Jeffrey F. Naughton Josef Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources", *in Proc of the 29th VLDB Conference*, Berlin, Germany, 2003.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[11] M. Kitsuregawa, M. Nakayama, and M. Takagi, "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method", *in Proc VLDB*, 1989, pages 257-266.

[12] Dong Keun Shin, Arnold Charles Meltzer, "New Join Algorithm", *in Proc. ACM SIGMOD*, USA, Dec. 1994, Volume 23 Issue 4, Pages 13-20.

[13] Bornea A.Mihaela,Vassalos Vasilis, Kotidis Yannis and Deligiannakis Antonios, "Adaptive Join Operators for Result Rate Optimization on Streaming Inputs", *IEEE transactions on knowledge and data engineering*, volume 22, No.8, pp.1110-1125, August 2010.

[14] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld, "An Adaptive Query Execution System for Data Integration", *In SIGMOD 1999*, pages 299–310.

[15] Y. E. Ioannidis, "The history of histograms (abridged)", *In Proc VLDB*, pages 19-30, 2003.

[16] W. Li, D. Gao, and R. T. Snodgrass, "Skew handling techniques in sort-merge join", *In Proc SIGMOD*, 2002, pages 169 - 180.

[17] Ron Avnur and Joseph M. Hellerstein "Eddies: Continuously Adaptive Query Processing", *in Proc the ACM SIGMOD International Conference on Management of Data*, May 16-18, 2000, volume 29, pages 261-272.

[18] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. "RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization", *In Proc of ACM SIGMOD Conference, Newyork, 2005*, Pg: 371-382.

[19] V. Zadorozhny, L. Raschid, M. Vidal, T. Urhan and L. Bright, " Efficient evaluation of queries in a mediator for web sources" , *in Proc. ACM SIGMOD International Conference on Data*", 2002, pg 85-96.

[20] N. Bruno and S. Chaudhuri, "Exploiting Statistics on Query Expressions for Optimization", *in Proc. ACM SIGMOD*, USA, June 2002, pages 263-274.