Adaptive join algorithms have recently attracted a great deal of attention in emerging applications where data is provided by autonomous data sources through heterogeneous network environment. The primary advantage of the adaptive join over traditional join techniques is that they can start producing join results as soon as the first input tuples are available. It does not wait for one relation to arrive completely to produce the join results. Moreover, and more significantly, in emerging data integration or online aggregation environment, a key performance metric is the rapid availability of first results and a continuous rate of tuple production. By being able to produce results whenever input tuples become available, adaptive join algorithms overcome situations like initial delay, slow data delivery or bursty arrival, which can affect the efficiency of the join operation.

A large number of adaptive join algorithms such as EHJ, RPJ, PermJoin, DHJ, HistoJoin have been proposed. All these algorithms are discussed in detail in section 1.5. EHJ produces early results by biasing the reading strategy and flushing policy to the smaller relation. RPJ is the first adaptive join algorithm which tries to understand and exploit the connection between the memory content and the algorithm output rate. PermJoin intelligently selects a percentage of data to be flushed to the disk based on the expected query throughput contribution. DHJ adapts to the changing memory
conditions, but it does not determine the best partition to be flushed to maximize join performance. It just selects the largest or smallest partition or a random partition to flush to the memory. It does not consider the data distributions to determine the optimal partition to flush. HistoJoin overcomes the limitations of the dynamic join by exploiting the data skew to improve performance of the join. The basic idea is to buffer in the tuples of the primary key tables whose key value occurs most frequently in the foreign key relation. By buffering the tuples in memory that have a higher probability of producing resultant tuples, reduces the number of I/Os performed by the join operation. The histogram is exploited to determine the tuples which contribute more to the join resultant tuples. HistoJoin performs Binary search or Bit array method to find the matching tuples from the inner table. But Binary search and Bit array method requires hashing and sorting techniques respectively. Hashing and sorting are blocking operations, because it creates a delay in result production.

To overcome the limitations of the HistoJoin, Maximized Result Rate Join Algorithm (MRR), which produces the first few results without much delay has been proposed in the present research. 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 the database statistics. Histogram provides the frequency of the attribute in a table. The tuples which have a 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. The main difference between the HistoJoin and MRR join is, the MRR join algorithm uses the concept of Least Common Multiplier (LCM) to find the matching tuples instead of Bit array method and Binary search method in Histojoin. In the Binary search method, the array is sorted and searched for the matching value. For integer values, in bit array method a bit is set in an
array if that index value is available in memory. In range check method the input value is hashed and the hash value is used to set the corresponding bit in the bit array. The Binary search and Bit array method requires sorting and hashing respectively, which are blocking operations. Another difference between MRR join and Histojoin is that it initializes a sliding window, using which the user can control the rate of the output tuples produced. The MRR Join algorithm, concentrates on achieving the following objectives,

1. Producing earlier query results without any major pre-work namely, without using hashing and sorting technique.

2. Maximize the result rate of the join query during the early stage of the join operation, by exploiting the histogram.

3. Terminating the join operation once the number of matching tuples is found.

3.1 BACKGROUND

3.1.1 Optimizer Statistics

Optimizer statistics are a collection of data that describe more details about the database and the database objects. These statistics are used by the query optimizer to choose the best execution plan for each SQL statement. Optimizer statistics include the following,

- Table statistics
  - Number of rows
  - Number of blocks
  - Average row length
- Column statistics
  - Number of Distinct Values (NDV) in column
The optimizer statistics are stored in the data dictionary. They can be viewed using data dictionary views. As the objects in a database can be constantly changing, statistics must be regularly updated so that they accurately describe these database objects. Statistics are maintained automatically by the database or it can be gathered manually using the DBMS_STATS package (available in ORACLE). A GATHER_TABLE_STATS procedure available in DBMS_STATS package is used to gather the statistics on the table. When gathering statistics on a table, DBMS_STATS gathers information about the data distribution of the columns within the table. The most basic information about the data distribution is the maximum value and minimum value of the column. However, this level of statistics may be insufficient for the optimizer's needs if the data within the column is skewed.

For skewed data distribution, histograms can also be created as part of the column statistics to describe the data distribution of a given column. Histograms are specified using the METHOD_OPT argument of the DBMS_STATS gathering procedures. It is recommended to set the METHOD_OPT to FOR_ALL_COLUMNS_SIZE as AUTO. With this setting, the optimizer determines which columns require histograms and the number of buckets (size) of each histogram. The user can also manually
specify which columns should have histograms and the size of each histogram. The following procedure can be executed to collect the statistics on the required table where SYSTEM is the schema name, Shipment is the name of the table.

BEGIN
DBMS_STATS.GATHER_TABLE_STATS ('SYSTEM', 'SHIPMENT',
    METHOD_OPT => 'FOR ALL COLUMNS SIZE AUTO');
END;

3.1.2 Introduction to Histogram

A histogram $H$ for an attribute $A_i \ (1 \leq i \leq n)$ is a list of buckets $H = \{B_1, B_2, ..., B_m\}$ with $m \geq 0$. A histogram is constructed by partitioning the data distribution of the corresponding attribute into a set of mutually disjoint buckets approximating the frequencies and values in the range of each bucket. Thus, a histogram is an approximation of the attribute’s data distribution. Taxonomy of different histogram types can be obtained by using different rules for partitioning values into buckets.

3.1.3 Method to Generate 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 the Cost Base Optimizer (CBO) to optimize a query. The histogram can be collected in the data dictionary table (available in ORACLE) known as DBA_HISTOGRAMS. There are two types of histogram namely,

- Frequency Histogram
- Height-Balanced Histogram
To create a histogram the user needs to specify the number of buckets. The number of buckets controls the type of the histogram created. If the distinct value of a column is less than 254, the frequency histogram is then created, otherwise height-balanced histogram is created.

### 3.1.3.1 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 (Number of buckets) must be $\leq 254$, that is FH can be collected only if the column has NUM_DISTINCT (number of distinct values in the column) $\leq 254$. Table 3.1 shows the histogram for the column CUST_ID on table shipment. ENDPOINT_NUMBER and ENDPOINT_VALUE in Table 3.1 are from DBA_HISTOGRAMS. The Count in Table 3.1 is derived by computing the difference between the current value and the previous value which is available as ENDPOINT_VALUE.

**Table 3.1 Frequency histogram**

<table>
<thead>
<tr>
<th>ENDPOINT_NUMBER</th>
<th>ENDPOINT_VALUE</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>102</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>104</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>311</td>
<td>26</td>
<td>1</td>
</tr>
</tbody>
</table>

Count (1) = 4 − 0 = 4  
Count (2) = 16 − 4= 12  
Count (3) = 17 − 16 = 1  
Count (4) = 22 − 17 = 5  
Count (5) = 25 − 22 = 3  
Count (6) = 26 − 25 = 1
3.1.3.2 Height-balanced histogram

In height-balanced histogram, the column values are divided into bands so that each band contains approximately the same number of rows. If the number of distinct values of the Join column is very large, greater than 254, the height-balanced histogram will be created. The following procedure is used to generate a height-balanced histogram, where the column size is set to 5 which is less than the number of distinct values available in the CUST_ID column. If the column size is set to auto and the number of distinct values in CUST_ID column is <254, frequency histogram will be automatically generated.

BEGIN
   DBMS_STATS.GATHER_TABLE_STATS (‘SYSTEM’, ‘SHIPMENT’, METHOD_OPT => 'FOR ALL COLUMNS SIZE 5');
END;

Table 3.2 Height balanced histogram

<table>
<thead>
<tr>
<th>ENDPOINT_NUMBER</th>
<th>ENDPOINT_VALUE(EP)</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>103</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>311</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
\text{count}(i) = \frac{\text{num}_{\text{rows}} \times \text{diff}_{ep}(i)}{\text{max}_{ep}} \tag{3.1}
\]

where,
\[
\text{max}_{ep} = \max (\text{ENDPOINT})
\]
\[
\text{diff}_{ep} (i) = \text{current} (\text{ENDPOINT VALUE}) - \text{previous} (\text{ENDPOINT VALUE})
\]
To generate height balanced histogram as shown in Table 3.2 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 column values, or the row for one column value can spill into several buckets. The column value that spills into several buckets is called the popular values. ENDPOINT_NUMBER and ENDPOINT_VALUE are from DBA_HISTOGRAMS, count is calculated from a formula (3.1). ENDPOINT_VALUE is zero for the first value sampled and indicates that this is the ENDPOINT$^{th}$ sample.

![Figure 3.1 Height balanced histogram](image-url)
As shown in Figure 3.1 the ENDPOINT_NUMBER=101 is sampled thrice, it is reported only once in the histogram, but the ENDPOINT_VALUE (EP) is increased by 3. If the ENDPOINT_NUMBER is sampled k times, then EP will be incremented by k. Values that are sampled more than once is called the popular values. Therefore, 101 is the most popular value.

3.2 PROPOSED FRAME WORK

The Figure 3.2 shows the framework 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 be the source tables to be joined. If |R| + |S| > M, then the entire input tables 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, it is necessary to determine the tuples which contribute more to the query output and bring such tuple from R and S that 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. A window of size W is assigned to restrict the range of tuples allowed to participate in the join operation. For example, if the window size W is 3, the top 3 join attribute values is then read from the histogram and given as input to the read operator.

The H in Figure 3.2 represents the histogram on the Table S using the histogram H, the read operator just reads the top 3 tuples which has the maximum join result from R into the memory to perform join operation. Now the matching tuples from S has to be brought into memory to perform the join operation. To find the matching tuples the Histojoin performs two methods Binary Search and Bit Array method which will add overhead to the join operation as it requires sorting and hashing respectively.

The MRR algorithm uses a simple concept of Least Common Multiplier (LCM) 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 the factor of the calculated LCM value is read from S. If there is an insufficient memory to hold the tuples of S, 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 in-memory 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, the join operation can be terminated when sufficient matching tuples are found, as further continuing the join operation will not yield any resultant tuples.

### 3.3 STEPS IN MRR JOIN ALGORITHM

The Maximized Result Rate join algorithm guarantees to produce maximum join results during the earlier stage of the join operation without much delay. 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 most of the real-time applications. Consider a primary-to-foreign key join between R(A) and S(B, A) on the join attribute A, where R is the smaller relation. The Histogram is collected for the relation S on attribute A as shown in Figure 3.3.

![Figure 3.3 Join operations on R and S](image-url)
If the distinct value of attribute A is less than 254, frequency histogram (FH) will then be generated, where each bucket value is the join attribute value and the frequency of the bucket is the frequency of the join attribute value. The histogram is sorted in decreasing order of the frequency of the bucket as shown in Figure 3.3.

Consider a join memory size of M tuples. If there is a memory constraint, then both the join tables R and S cannot be accommodated in the available memory most of the time. Hence, it is important to determine which tuples of R and S can be joined first to produce maximized join results. To achieve the maximized join results during the early stage of join, the tuples which have the maximum frequency of occurs in the table S is joined first. However, how many numbers of such tuples can be brought to the memory to join is another major question. Hence, a window of size W is fixed. For example, if W is 3, then the top three tuples which have the highest frequency in the histogram is read from the table R into the sliding window. Smaller values are selected as sliding window size W; this is to give more space for the matching tuples from S to be accommodated in memory. If the memory is still free after accommodating the tuples in W and the corresponding matching tuples from S, the window size can be increased. But when the user needs the result in no time, it is better to select a smaller sliding window size, because the size of the sliding window determines the number of comparisons required to produce the join resultant tuples. 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 fixed to a larger value. Thus the sliding window size W can be customized to the user needs.

Let \( R_1 \ldots R_W \) be the tuples of table R available in the sliding window which has the maximum join tuples in table S. To bring the corresponding matching tuples of table S to memory a range check is
performed between each tuple in window W and tuples in table S. In histojoin, range check is implemented in two methods. The first method sorts the ranges and performs a binary search to find the matching tuples. The second approach is for integer values called Bit Array method. This method requires hashing the input value and checking if the bit is set in the bit array. To perform the Binary Search method, sorting is required. To perform Bit Array method hashing and sorting are required. This may delay the production of resultant tuple as both hashing and sorting are blocking operations, which will not produce the resultant join tuples until the sorting or the hashing operation is completed.

MRR join algorithm proposes a simplified range check method using the concept of the Least Common Multiplier. The LCM of tuples in sliding window W (R₁... Rₜ) is calculated; the tuples which are the factors of the calculated LCM are retrieved from table S (let S₁, S₂ .... Sₙ be the matching tuples from S) into the memory to perform join, if the condition |R₁ ...
Rₜ| + |S₁, S₂...Sₙ| <= M (Memory) holds, else the matching tuples from S is read into 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 could not be accommodated in the memory in the previous phase are brought to the memory to perform a join operation. The number of comparisons required to produce the phase1 join results are |R₁, R₂..., Rₜ| * |S₁, S₂...Sₙ|. However, the MRR join algorithm terminates the join operation once the required matching tuples are found, this is achieved with the help of the histogram available on the join attribute column. Hence, the number of comparisons will be less than |R₁, R₂, ..., Rₜ| * |S₁, S₂..., Sₙ|. This feature enhances the performance of the MRR join algorithm when compared to the histojoin and it is capable of producing the maximum resultant tuples with lower delay than the histojoin.
Case 1 : No Memory Overflow

(a) Join stage I

(b) Join stage II

Case 2 : Memory Overflow

(c) Join stage I

(d) Join stage II

Figure 3.4 (e) Join stage III

Figure 3.4 Steps of MRR join algorithm
In the case of height balanced histogram, the values that are sampled more than once is the popular values. In Table 3.2 VALUE=101 is sampled twice, but it is reported only once in the histogram, but the EP is increased by 3, not by one. The EP will be increased by k times if it is sampled k times. Hence, the values which are sampled more number of times are selected as the top values from the histogram. The tuples which have top values considered to produce more join results. The other procedures are the same as it is done with the Frequency histogram.

Figure 3.3 shows the relation of R and S and histogram on table S on join attribute t1. The histogram is arranged in the descending order of the frequency. Figure 3.4 (a), shows a Window W of size 3 and a memory M of size 10 tuples. Based on the size of the window, the top three join attributes values are selected from the histogram i.e., tuples with join attribute values 6, 2 and 8. The matching tuples from table R which has the maximum frequency in the histogram is brought to the sliding window W to perform the join operation. The tuples from table S which matches with the set of tuples in W has to be identified and brought to the memory to initiate the join operation. To identify the tuples which may generate the resultant tuples when joined with the tuples in W, range check is performed. The range check in the MRR join algorithm is done using the concept of LCM.

During the stage 2 the next top 3 tuples are read from the histogram (tuples with join attribute 3, 7 and 5) as shown in Figure 3.4 (b) and the same process is repeated as previous. If the size of the memory is limited as shown in Figure 3.4 (c) - 3.4 (e) (Memory size = 6 tuples) there is no enough space to accommodate all the matching tuples in the memory, therefore the join happens in three stages. When the first stage is completed, the memory is cleared and the remaining matching tuples are brought to the memory to join with the tuples in the window W. During the join operation, the number of
matches for each join attribute can be determined with the help of the histogram and the join operation can be terminated when the required matching tuples are found. For example, in Figure 3.4 (c), the join operation for the tuple with join attribute value 2 is terminated without comparing the tuple with all the tuples of table S in memory as all the available matches for the tuples with join attribute 2 is found in the 3rd comparison and so the further comparisons will not yield any join query result.

3.4 PSEUDO-CODE FOR MRR JOINALGORITHM

**Input:** Outer table (R_A), Inner table (R_B), Distinct join keys in R_B (K), Histogram (which contains the frequency of the join keys in R_B arranged in increasing order of the frequency of the join key), Least Common Multiplier (LCM), window size (W), Join memory size (M).

**Output:** Join result for R_A \( \bowtie \) R_B

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

Let R and S be the source tables to be joined on join attribute A. If |R| + |S| > M, then the entire input tables 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, the tuples with a high frequency of occurrence are selected to join first. To determine the frequency of the join attribute $A$, the histogram is exploited, which is available in the database statistics. For example, if the window size $W$ is $k$ then top $k$ tuples $(r_1, r_2, ..., r_k)$ with join attribute $A$ of relation $R$ which has the highest frequency of occurrences in table $S$ is brought to the memory to join. Now the tuples in relation $S$ which matches the $k$ tuples in memory has to be determined. Histojoin (Bryce Cutt 2009) uses Binary search or Bit array method to find the matching tuples in relation $S$. However, MRR join algorithm uses a simple concept of Least Common Multiplier ($LCM$) to retrieve the matching tuples from $S$. LCM is calculated for the join attribute values of $k$ tuples in the window $W$ and the tuples whose join attribute value is factor of $LCM$ are read from $S$ and brought to the memory to perform the join operation. If there is an insufficient space in $M$ to hold the matching tuples of $S$, then only the part of the matching tuples from $S$ which can be accommodated in $M$ are read to perform the join operation. If there are $l$ matching tuples in $S$ for $k$ tuples of $R$ in memory and when $l > (M - K)$ then only $(M - K)$ tuples are read into memory to perform a join operation. After the tuples are brought to memory, the simple nested loop join algorithm is used to join the tuples. The rate at which the algorithm produces the join result is given by,

$$Rate \text{ of Outer Tuple Generated} = \sum_{i=1}^{k} f(a_i) \quad (3.2)$$

Where, $f(a_i)$ is the frequency of the tuple $a_i$ in relation $S$.

The frequency $f(a_i)$ is calculated from the histogram $H$ which is of the form $\{ (a_i, b_i) | a_i \in R.A \text{ and } b_i = \text{count}_s(a_i) \text{ and } b_i > b_{i+1} \}$ where $\text{count}_s(a_i)$ indicates the frequency of attribute $a_i$ in table $S$. Once the join process is completed the memory is flushed out. The remaining $l - (M - k)$ tuples are read and the above process is repeated to obtain the resultant tuples.
During the join process when the sufficient matching tuples are found the join operation is terminated, without further comparison, as further comparisons will not yield any resultant tuples. The cost of joining the relation $R$ and $S$ using MRR Join algorithm is given as follows,

$$\text{Read}(R) + LCM(r_1, r_2, \ldots, r_k) + \text{Read}(S) + \sum_{i=1}^{\frac{M}{k}} k \times \text{Min}(i, M - k) \times \max(l - (M - k), 0) \tag{3.3}$$

where,

- Read (R) - Time taken to scan relation R
- $LCM (r_1, r_2, \ldots, r_k)$ - Time taken to calculate the least common multiplier
- Read(S) - Time taken to scan S
- M - Size of the Memory
- l - Number of matching tuples in S for k tuples in memory.
- H - Histogram
- K - Number of tuples in Window W

As the parameter $W$ increases the number of comparisons required to produce the join results also increases. Thus the resultant rate of the algorithm decreases. The experiment can be repeated for different values of $W$ to determine the optimal value.

### 3.5 PERFORMANCE ANALYSIS

The experiment was conducted on an Intel® corei5 2.50GHz processor, with 4GB real memory, running windows7 and Java 1.7.0_09. The MRR join algorithm was analysed with two data sets, the synthetic and
real-time dataset generated from TPC-H. The detailed explanation of the
dataset used is available in section 1.7.

3.5.1 Real-Time Dataset

The number of comparisons required by HistoJoin and MRR join
algorithm is given in Figure 3.5. The graph shows that the proposed MRR
join algorithm requires 2-3% more comparison than the HistoJoin. However,
the MRR join algorithm does not use any of the blocking operations like
hashing or sorting. Therefore, the resultant tuples are generated as soon as the
join operation commences. The graph in Figure 3.6 shows the delay in
generating the resultant tuples. It shows that HistoJoin takes 20-25% slack
time in producing the query results as it uses the Bit Array and Binary Search
method.

Figure 3.5 Performance analysis on the number of comparisons
required
3.5.2 Synthetic Dataset

The MRR join algorithm is also compared with the traditional join algorithms, as all the recent join algorithms developed are an enhanced version of the traditional join algorithms. As shown in Figure 3.7 the number of comparison 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.

As shown in Figure 3.8 the resultant tuples produced by the MRR join algorithm is high during the initial stages of the join. The join phase in the Figure 3.8 indicates the different stages in the MRR join. The algorithm reads the tuples from both the sources and once the memory is full, it stops the reading phase and starts the production phase. During the production phase the resultant tuples are produced. The tuples which have already generated the resultant tuples are cleared from the memory, when the memory
is free to accommodate new tuples the algorithm continues with reading phase. The output tuples are generated only during the production phase. The graph shown in Figure 3.8 shows the resultant rates of the tuples during each production phase. The MRR join algorithm produces maximum resultant tuples during the early stages of the join.

**Figure 3.7 Performance of MRR join Vs traditional join algorithm**

**Figure 3.8 Rate of the resultant tuples produced by MRR join**
Figure 3.9 shows the rate of the resultant tuples produced by the traditional join algorithm (namely, Nested Loop Join). It is clear from Figure 3.9 that the rate of the resultant tuples is varied during the each stage of the join operation. But MRR join algorithm promises a maximum result rate during the early stage of the join operation.

**Figure 3.9 Rate of the resultant tuples produced by traditional join methods**

**Figure 3.10 Time taken to produce the first resultant tuple**
Figure 3.10 compares the time taken by the join algorithms to produce the first tuple. Figure 3.10 shows the pre-work required by each algorithm before it could produce the first resultant tuple. The 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 results in quadratic performance. MRR join produces 60% more tuples than the traditional join methods during the early stages of the join. It also produces 20% early join results than the sort merge join and 60% early than the hash join.

3.6 CONCLUSION

The proposed MRR join algorithm can be used to produce earlier and maximized join results when limited memory is available to perform the join operation. This is achieved by exploiting the distribution of the data in the join attribute column, which is available in the histogram. The rows which will produce the maximum join results are identified with the help of histogram and are joined during the earlier stages of the join operation. The recent adaptive join algorithm that produces the results earlier uses the concept of hashing and sorting. Even though they claim to produce the resultant tuples early, the hashing technique used in these algorithms induce a delay in production of resultant tuples. This delay is due to the construction of hash table. Further, the hash table has to be accommodated in the memory which leads to memory overhead.

The proposed MRR join algorithm does not require hashing or sorting which in turn results in a reduction of the memory and I/O overhead. It applies a concept of LCM to find the matching tuples to retrieve to the memory to perform the join operation. The algorithm result shows that it requires 2%-3% more comparison than the Histojoin, but the delay in
producing the first tuples are 20-25% less than the Histojoin. In MRR join algorithm the join operation can be terminated when the required matching tuples are found. This can further reduce the number of comparisons and the delay in producing the join results, which in turn reduces the time and I/O overhead.