CHAPTER 5

PROPOSED JOIN ALGORITHM USING PATH MATRIX

Improving the performance of the join query processing in a relational database management system continues to be a challenging area of database research as it is the one of the costliest operation and it critically determines the query performance. The efficiency of the execution of repetitive join queries has a paramount effect on the performance of the database system and thus deserves more optimization efforts. Repetitive queries refer to those queries that are likely to be executed repeatedly in the future. Examples of repetitive queries include those that are used to generate periodic reports, perform routine maintenance and summarize data for analysis. They constitute a large part of daily activities of the database system and deserve more optimization effort. In such situations the join operations are improved by constructing a special data structure that will facilitate faster access, namely, T-tree, Data Space Partitioning, KD-Trees, BC-Trees and Quadree. Join operations can also be improved by maintaining a list of matching tuples for commonly joined relations. Data structures like Join index and materialized view pre-compute the join result and the entire tuples of the join tables are stored in materialized view and only the join tuple identifier is stored in the case of the join index.

In all the existing join algorithms which maintain the data structure to facilitate the join operation some pre-work is involved to populate this data structure which may result in a delay in the join processing. As shown in Figure 5.1, creating the join index and materialized view reduces the query
response time. However, as the response time of the query decreases, the space used to maintain the data structure and the maintenance overhead increases. The materialized views (MVs) and indexes both significantly speeds up the query processing in the database systems, but consumes disk space and more importantly needs to be maintained when an update operation occurs. The above limitations can be eliminated by using the proposed Path Matrix join algorithm.

**Figure 5.1 Performance of join index and materialized view**

The proposed Path Matrix join algorithm uses a simple data structure called Path Matrix, which improves the performance of the join operation in the context of complex queries. In some database applications certain join queries will be executed frequently. In such applications, when the path matrix join algorithm is used, the computation cost of performing the join operation can be reduced considerably. The proposed method does not
build intermediate join tables, but works directly on the table and builds the data structure as it performs the join operation. Path matrix join algorithm is designed to avoid complete table scans and maximize the rate of the join tuples generation. The relevant tuples from the source table can be retrieved using fewer I/O’s than would be required by a full scan of the source table. Path matrix is a general method for processing equi-join in relational databases. It is especially efficient if the smaller relation fits completely in the main memory. Further, the number of disk I/O required by the Path matrix join algorithm is very low as, each relation is read only once to produce the join resultant tuples.

5.1 FUNCTIONALITY OF THE PROPOSED JOIN ALGORITHM

- The table with the foreign key and primary key is taken as the outer table and the inner table respectively. A Foreign key may be either referring a key in the primary key table or it may take NULL value. In the proposed method, a data structure called Path Matrix is constructed which includes the join attribute value and its ROW_ID. For every join attribute value present in the outer table the corresponding matching tuple is determined in the inner table and the ROW_ID of the matched tuple is recorded in the path matrix. Multiple rows in the referencing table/outer table may refer to the same row in the parent table/inner table. But the ROW_ID of that row is recorded only once in the path matrix. This difference makes the path matrix join algorithm more space efficient than the join index. Since the foreign key table is taken as the outer table, only the join attributes values present in the outer table which can produce the join resultant tuples are included in the path matrix.
The proposed method does not require any pre-work to construct the path matrix. The path matrix is constructed as the join process takes place.

The path matrix contains the ROW_ID of the tuples along with the join attribute value. This ROW_ID is the fastest and easiest way to access the rows of the table.

Path Matrix is not stored permanently in the memory. Instead, it is a temporary data structure which is constructed during the join process and it can be discarded as the join completes. Hence, no memory is required to permanently store the path matrix. Since no pre-work is done to construct the matrix, there is no loss in deleting the path matrix after the completion of the join process. Therefore the update operations like insert, delete and update of the table will not affect the path matrix.

A new Random Record join algorithm is proposed to find the matching tuples in the inner table. Once the matching tuples are found, the ROW_ID of the matching tuples are recorded in the path matrix.

5.2 BACKGROUND

For each row in the database, the ROW_ID pseudo-column returns the address of the row. It is a 64 bit string. ROW_ID values have several important uses, namely,

- They are the fastest way to access a single row.
- They can show how the rows in a table are stored.
- They are unique identifiers for rows in a table.
A ROW_ID is assigned to a row upon insertion and is immutable unless the row is deleted and re-inserted. The following query displays the ROW_ID for each row in a table named mydept.

```
SQL> SELECT ROWNUM, ROWID, DEPTNO FROM MYDEPT;
```

<table>
<thead>
<tr>
<th>ROWNUM</th>
<th>ROWID</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAAJWTAAWAAB+BiAAA</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>AAAJWTAAWAAB+BiAAB</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>AAAJWTAAWAAB+BiAAC</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>AAAJWTAAWAAB+BiAAD</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>AAAJWTAAWAAB+BjAAA</td>
<td>10</td>
</tr>
</tbody>
</table>

The ROW_ID is also called as TID’s (Tuple Id). It is an 18-character representation of the location of data in the database, with each character represented in a base-64 format consisting of A-Z, a-z, 0-9, + and /.

The ROW_ID format consists of the following four components,

- Database Object Number
- Relative Data file Number
- Data Block Number
- Row Number
5.3 FRAMEWORK FOR THE PROPOSED PATH MATRIX JOIN ALGORITHM

In the proposed Path Matrix join algorithm, a data structure called Path Matrix is constructed which is used to improve the performance of the join operation. The block diagram of the proposed Path Matrix join algorithm is given in the Figure 5.2. As shown in Figure 5.2, the table with the foreign key and primary key is designated as outer and inner table respectively. In step 1, the first row of the outer table is compared with the inner table. The proposed Random Record join algorithm is used to find the matching row in the inner table. As the table with foreign key is taken as the outer table, each row of the outer table will have exactly one matching row in the inner table, but the location of that exact matching row is unknown. If N is the number of rows in the inner table, each row of the outer table, picks up a random record to find a match. This random record is picked by generating the random number.

In some situations first pick may be the best pick, in that case all the subsequent records with the same join key attribute values tends to follow the same random numbers which led to the matching row. Once the matched row is found in the inner table the join resultant tuples are produced immediately and the ROW_ID of the matched tuple are recorded in the path matrix as given in step 2, 3 and 4 of Figure 5.2 respectively. For the second row and the other rows of the outer table, first the path matrix is checked for the join attribute and ROW_ID value pair as in step 5. If the path matrix contains the entry for the join attribute value, the ROW_ID is used to retrieve the row and the join resultant tuples are produced immediately as shown in step 6. If the path matrix doesn’t contain the ROW_ID of the required join attribute value, then using the proposed Random Record join algorithm, the matching tuples for the row is found and the ROW_ID is updated in the path matrix by repeating the step 7, 2, 3 and 4 of Figure 5.2 respectively.
For the first row of the outer table no records are available in the path matrix.

Perform a full table scan.

Return the ROW_ID of the matching tuple.

ROW_ID and the join attribute value pair is updated in the path matrix.

For the second row of the outer table scan the path matrix to locate the join attribute value and ROW_ID value pair.

After determining the ROW_ID access the inner table to generate the join result tuples.

If path matrix doesn’t contain the join attribute and ROW_ID value pair, the ROW_ID is determined using the proposed random record join algorithm.

Figure 5.2 Block diagram for path matrix
### Table 5.1 Item I

<table>
<thead>
<tr>
<th>Item_ID</th>
<th>Item_Name</th>
<th>Price</th>
<th>ROW_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>Laptop</td>
<td>400$</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>LED lamp</td>
<td>12$</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Modem</td>
<td>50$</td>
<td>3</td>
</tr>
<tr>
<td>60</td>
<td>Boots</td>
<td>15$</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>Lock</td>
<td>1$</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>Led tv</td>
<td>1000$</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>Oven</td>
<td>100$</td>
<td>7</td>
</tr>
</tbody>
</table>

### Table 5.2 Customer C

<table>
<thead>
<tr>
<th>Cust_id</th>
<th>Cust_name</th>
<th>Item_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>David</td>
<td>10</td>
</tr>
<tr>
<td>200</td>
<td>Joseph</td>
<td>10</td>
</tr>
<tr>
<td>300</td>
<td>Mary</td>
<td>10</td>
</tr>
<tr>
<td>400</td>
<td>Peter</td>
<td>20</td>
</tr>
<tr>
<td>500</td>
<td>Solomon</td>
<td>20</td>
</tr>
<tr>
<td>600</td>
<td>Mathew</td>
<td>30</td>
</tr>
<tr>
<td>700</td>
<td>Ruth</td>
<td>30</td>
</tr>
</tbody>
</table>

### Table 5.3 Path Matrix

<table>
<thead>
<tr>
<th>Join Attribute</th>
<th>ROW_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
</tr>
</tbody>
</table>
Table 5.1 and 5.2 represent the customer and Item table respectively. Now a simple data structure called path matrix has to be constructed to facilitate the join operation. To construct the path matrix, a new join algorithm called a Random Record join algorithm is proposed. Random record join algorithm avoids the full table scan by retrieving tuples at random from the source relation. The performance of the proposed Random Record join algorithm depends on the quality of the random number generated.

![Graph showing the performance of the proposed random record join algorithm]

**Figure 5.3 Performance of the proposed random record join algorithm**

The algorithm can exhibit best case, average case or worst case behavior. Best case is when the first retrieved record matches with the join attribute value. The proposed Random Record join algorithm exhibits worst case behavior when all the rows of the inner table are retrieved to find the matching tuples, now in this case the algorithm behaves as NLJ. The graph in Figure 5.3 shows that the proposed Random Record join algorithm exhibits average case behavior only once during the 50 iterations of the algorithm and no worst case behavior is reported the same. This experiment was conducted with 1000 tuples in a table and the proposed random record join algorithm didn’t perform a full table scan to find the matching row, instead it accessed record at random to generate the join results. Thus, by using the proposed
Random Record join algorithm, the number of I/O performed is less than the other memory join algorithms. Once the matching rows are found using the Random Record join algorithm, the join attribute value and ROW_ID pair is recorded in the path matrix.

The table customer C with foreign key Item_id as shown in Table 5.2 is taken as the outer table. The table Item I with primary key Item_id as shown in table 5.1 is taken as the inner table. Each row of the outer table will have exactly one matching row in the inner table. When an NLJ algorithm is performed on the above mentioned tables, all the rows of the outer table will be matched with every row of the inner table. However, in the Random Record join algorithm a random record is picked to see whether that row is matched with the row from the outer table. If it does not match, another random record is picked until a match is produced. Once the match is found the ROW_ID of the matched row is remembered and updated in a path matrix. For the purpose of simplicity, the ROW_ID of each row is represented as integer as shown in the Table 5.1. The second row of Table 5.2 [C] also has the same join key attribute as that of the first row; this second row learns from the previous row that the matching ROW_ID is 5. Hence, it does not pick the random records, instead it first checks the path matrix for join attribute and ROW_ID value pair and generates the join result with the obtained ROW_ID. If the join attribute and the ROW_ID value pair is missing in the path matrix, it used the proposed random record join algorithm to determine the ROW_ID.

The following illustration shows a simple example of how the first occurrence of the join attribute generates a sequence of random numbers to find a matching row and how the second occurrence of the same join attribute uses the previous learnt path from the path matrix and finds the matching row. Each time the number of comparisons of the algorithm will be different as the quality of the algorithm depends on the random number generated. Worst case
never occurs in this algorithm as the ROW_ID’s of the matched rows are remembered and reused. However, random record join algorithm suffers from the worst case if unlucky numbers are got from the random number generator.

\[ C[1]=10 \ P[C[1]] \rightarrow [6, 7, 1, 4, 3] \] (the random path taken by join attribute 10)

\[ J[1] \rightarrow [3] \] (Join result set for join attribute 10)

\[ C[2]=10 \ P[C[I]] \rightarrow [3] \] (as the path for the join attribute 10 is already explored and the matching row is found in the ROW_ID 3, the same path is used.)


\[ C[3]=10 \ P[C[I]] \rightarrow [3] \] (as the path for the join attribute 10 is already explored and the matching row is found in the ROW_ID 3, the same path is used.)


\[ C[4]=20 \ P[C[I]] \rightarrow [1, 4, 7, 3, 2, 6] \] (random path taken by join attribute 20)


\[ C[5]=20 \ P[C[I]] \rightarrow [6] \] (as the path for the join attribute 20 is already explored and the matching row is found in the ROW_ID 6, the same path is used.)


\[ C[6]=30 \ P[C[1]] \rightarrow [1, 7, 2] \] (the random path taken by join attribute 30)

C[7]=30  P[C[I]] → [2] (as the path for the join attribute 30 is already explored and the matching row is found in the ROW_ID 2, the same path is used.)

J[7] → [2] (Join result set for join attribute 20)

Where P [] contains the random path taken by the join attribute to find the matching row in the inner table. The random number indicates the ROW_ID of the rows in the inner table. J [] list indicates the join result set for each join attribute.

5.4  PSEUDO CODE FOR PATH MATRIX JOIN ALGORITHM

**Input:**  Tables to be joined RA and RB

**Output:**  Join result set, which contains the row_id of inner table which has a matching join attributes with the outer table.

**Initialization:**

Set Ro  ← Outer table with reference key
RI  ← Inner table with primary key
P[] ← An Array list which, stores the best path taken by the join attribute.
Pathexplored ← An array list which stores the join attribute for which the path already exists
J[] ← An Array list which stores the result set for each join attribute in the Ro

Begin
for i:=1 to no. of rows in (Ro) do
    index:=pathexplored.lastIndexOf(RO[i])
    if(index!=-1)
        J[i]=P[index].get[P[index].size()-1]; //use the path stored in P[i]
and find the matching row_id and add to result set.

else

repeat

value:=Ro[i];
j←Random(RI); // returns random row_id from RI
A ← RI[j];
P[i].add(j); // Store the random path taken by C[i]

While (A!=value)

Pathexplored.add(C[i]);
J[i]=j;
End if;

Random (RI)

for(j=0;j<RI;j++)

{ 

Int RandomInt = randomGenerator.nextInt(count);

if(RI[i]==Ro[i])

Break;

//Matching Found

else

Repeat Random();
}

5.5 PERFORMANCE ANALYSIS

Assume R and S are the two tables to be joined on join attribute A with cardinalities N_R and N_S respectively. Attribute A in R is a set of n values \( \{a_1, a_2, \ldots, a_n\} \) Attribute A in S is a multi-set of m elements from \( \{a_1, a_2, \ldots, a_n\} \) each of which may have \( (x_1, x_2, \ldots, x_n) \) copies. In this proposed join algorithm the relation with foreign key S is taken as the outer table and the relation with primary key R is identified as inner relation. For each tuple \( a_i \) in outer relation S there will be exactly only one matching tuple in the inner
relation R. For each tuple \( a_i \) in \( S \) for \( i \leq N_s \) the matching tuple in set R has to be found. To find the matching tuple, the proposed Random Record join algorithm is used, in which a random record \( a_k \) is picked from the inner table R and checked for a match. This process will be repeated until a matching record is retrieved. Once the matching tuple is found the \( ROW_ID \) of the matched tuple \( a_k \) is recorded in the path matrix along with its join attribute value. \( ROW_ID \) is the pseudo column which contains the physical address of the row. Path matrix \( P \) is set of order pair of the form \( \{(x,y), x R y \mid x \in Distinct(S.A), y = ROW_ID(a_k)\} \)

For all the values of \( a \) for \( i \geq 2 \), the path matrix is checked for an entry \( a_i \), if an entry exists in the path matrix for attribute \( a_i \), the \( ROW_ID \) of the tuple is retrieved from the path matrix and used to produce the join result immediately, without scanning the inner table. If an entry for tuple \( a_i \) is not available in path matrix, the \( ROW_ID \) of the tuple \( a_i \) is then found using the Random Record join algorithm. This Path Matrix is a temporary data structure constructed only during the join process and it will be discarded as soon as the join tuples are produced. No pre-work is required to populate the path matrix, as it is constructed as the part of join process. If the join query is a repeated join query, the path matrix may then be considered to be stored in the memory for reuse. This decision is made based on the frequency of executed join query which is available in the optimizer statistics. This proposed Path Matrix join algorithm does not occupy extra memory as the path matrix is discarded after the join process. The number of operations required by the proposed path matrix algorithm is given by,

\[
\sum_{i=1}^{count(Distinct(S))} random(a_k, R) + write(P) + \sum_{i=1}^{N_s - count(Distinct(S))} read(p)
\]  

(5.1)

where,

\( random (a_k, R) \) - time taken to pick a random tuple \( a_k \) from R
write \( (p) \) - time taken to update the path matrix

read \( (P) \) - time taken to find the ROW_ID of the join attribute value pair in the path matrix

The path matrix is constructed using the Random Record join algorithm. In Random Record join algorithm a “search” is successful if every row of the outer table \( (S_i) \) discovers a matching row in the inner table \( R_i \). The success rate of the random record join algorithm is defined as the fraction of the number of rows visited to find the matching by the total number of rows in the inner table. The probability that the random record join algorithm finds the matching row after \( k \) iteration is given by \( P(A) \).

\[
P(A) = 1 - \left(\frac{1}{2}\right)^k \quad (5.2)
\]

The search delay is the duration of time between the start of a search for the matching row and the time when the matching row is found. The overhead of the random record join algorithm is the space required to store the ROW_ID’s of the rows visited.

5.5.1 Success Rate of Random Record Join Algorithm

The success rate is the probability that a search is successful in finding the matching row in the inner table. Since the foreign key table is taken as the outer table, every row in the foreign key will definitely have a matching row in the inner table. Therefore, the random walk algorithm will not terminate unsuccessfully.

5.5.2 Expected Delay

The expected delay equals the time taken to find the matching row from the initiation of a search. The delay is given in terms of the number of
unsuccessful searches between the start of the search and the discovery of the matching row. Let the delay of the search be equal to D.

\[
D = \sum_{i=1}^{N_S} 1 - \left( \frac{1}{n} \right) \tag{5.3}
\]

where,

- \( n \) - Number of rows in the inner table.
- \( N_S \) - Number of rows in outer table \( S \)
- \( \frac{1}{n} \) - The probability that a row selected at random contains the join attribute value \( k \)

Since the primary key table is the inner table, the join attribute value is unique. The probability that a row with the join attribute value \( k \) is given as \( \frac{1}{n} \) has been selected for the present research, as there will be only one row with value \( k \) among \( n \) rows.

5.4 PERFORMANCE EVALUATION

The proposed path matrix join algorithm is analyzed based on two datasets, synthetic dataset and real-time dataset generated from TPC-H benchmark. The detailed explanation on the dataset is given in section 1.7. The performance of the Path Matrix join algorithm is evaluated by the assumption that the inner table is entirely accommodated in the main memory. If memory is a constraint and the inner table is too large to fit in the main memory, multiple disk reads are then required to populate the path matrix. This will degrade the performance of the algorithm. The performance of the path matrix join algorithm depends on the performance of a Random Record join algorithm which in-turn depends on the random number generated. Path matrix join algorithm is commonly used to perform one-to-many join with primary key and foreign key relationship.
5.6.1 Real-Time Dataset

The number of comparisons required by HHJ, Join Index, Materialized View and Path Matrix join algorithm are given in Figure 5.4. This graph specifies the number of comparisons performed during the join operation. In the case of join index the number of comparisons indicates the comparison required to locate the ROW_ID of the join attribute value in the path matrix. The number of comparisons required by the path matrix is less than that of the join index since, the path matrix contains only the attribute which contributes to the join query result. The attribute which cannot produce join resultant set are not included in the path matrix. This reduces the memory overhead and also prevents unnecessary comparisons. The number of Comparison of the materialized views are less than that of all the existing methods because the join is pre-computed and stored in materialized view. During the join query execution the results are obtained from the materialized view without accessing the base tables. The comparison is necessitated only when the query includes some select conditions apart from the join condition.

![Figure 5.4](image-url)

**Figure 5.4** Performance analysis of the number of comparisons required to produce the join resultant tuples
The Figure 5.5 shows the rate of the resultant tuples generated by HHJ, Join Index, materialized View and Path Matrix Join algorithm. Initially, the tuples generated by the path matrix is less than the HHJ, Materialized view and Join index. However, as the time increases the rate of the resultant tuples generated by the proposed Path Matrix join algorithm increases. This is because, during the initial stages, the Path Matrix are constructed, during the construction stage, the inner relation has to be accessed to populate the path matrix.

![Figure 5.5 Performance analysis on rate of resultant tuples](image)

**Figure 5.5** Performance analysis on rate of resultant tuples

![Figure 5.6 Percentage of space overhead](image)

**Figure 5.6** Percentage of space overhead
Once the path matrix is fully populated, the query throughput increases. In the case of Join Index, as the size of the table increases, the size of the index also increases which leads to poor query throughput. However, the size of the database has no effect on the materialized view as the join is pre-computed. Anyway, the overhead of creating materialized view is higher than that of the other join techniques. The Figure 5.6 shows the overhead involved in the implementation of Hybrid Hash Join, Join Index, Materialized View and Path Matrix.

5.6.2 Synthetic Dataset

![Path Matrix Vs Traditional Join Algorithm](image)

**Figure 5.7 Path matrix Vs traditional join algorithm**

The proposed path matrix join algorithm is tested with the synthetic dataset and analyzed with the traditional join algorithms and the results of the evaluation are given in the Figure 5.7-5.8.
Figure 5.7 shows the results of the comparison of the path matrix algorithm with the traditional join algorithms using the real-time dataset. When compared to the other join algorithms the path matrix join algorithm takes few comparisons to produce the join results. The behavior (output) of the algorithm can vary if the algorithm is run multiple times on the same input. Hence, the result is dependent on the quality of the random numbers generated in the Random Record join algorithm. The outcome of the algorithm and running time depends on random choices (besides input).

**Figure 5.8 Comparison of I/O reads required by various join algorithms**

Figure 5.8 Shows the I/O reads required by the various join algorithms. As the number of rows in the database increases, the I/O read required by the path matrix join algorithm decreases as the ROW_ID of the required tuples are stored in the path matrix and are used to retrieve the matching rows from the table instead of scanning the full table to find the matching row.
5.7 CONCLUSION

The path matrix join algorithm is usually simple and easy to implement, the algorithm is fast, and it produces optimum output with very high probability. In applications where the join output is required without any delay and much pre-work path matrix join algorithm can be used.

This algorithm is capable of producing the join resultant tuples immediately as the path matrix is constructed. This method doesn’t require any pre-computations. Path matrix requires less space when compared to join index and other join algorithms as only the ROW_ID of rows which are capable of producing the join resultant tuples are stored in the path matrix. The other rows which cannot produce the join resultant tuples are not considered to be stored in the path matrix. As the Path Matrix is constructed during the join process and are discarded after the join query execution, it doesn’t require any space overhead as required by the join index and materialized view. Similarly path matrix doesn’t require maintenance overhead like update, insert and delete as involved in join index and materialized view.