New Bucket Join Algorithm for Faster Join Query Results

Hemalatha Gunasekaran[1] and ThanushkodiKeppana Gowder[2]
[1]Department Of Information Technology, Ibri College of Applied Sciences, Oman
[2]Akshaya College of Engineering, Anna University, India

Abstract: *Join is the most expensive and the frequent operation in database. Significant numbers of join queries are executed in the interactive applications. In interactive applications the first few thousand results need to be produced without any delay. The current join algorithms are mainly based on hash join or sort merge join which is less suitable for interactive applications because some pre-work is required by these algorithms before it could produce the join results. The nested loop join technique produces the results without any delay, but it needs more comparisons to produce the join results as it carries the tuples which will not yield any join results till the end of the join operation. In this paper we present a new join algorithm called bucket join which will over comes the limitations of hash based and sort based algorithms. In this new join algorithm the tuples are divided into buckets without any pre-work. The matched tuples and the tuples which will not produce the join results are eliminated during each phase thus the no. of comparison required to produce the join results are considerable low when compared to the other join algorithms. Thus, the bucket join algorithm can replace the other early join algorithms in any situation where a fast initial response time is required without any penalty in the memory usage and I/O operations.*

Keywords: *Bucket join, hash join, query results, nested loop join, sort merge join.*

Received February 25, 2013; accepted June 9, 2014, published online August 9, 2015

1. Introduction

In relational database management system information's are organized in to collections of tables. In database after normalization, the information's are broken down logically into smaller, more manageable tables. To retrieve a data, two or more tables have to be joined more frequently. The example of a join query is show in the Figure 1. Thus, join operation becomes the most frequent operation in the normalized database. Additionally, joins are one of the most expensive operations that a relational database system performs [1]. Joining two tables will consume a significant amount of the system's CPU cycles, disk band-width, and buffer memory. To improve the performance of the system an efficient join algorithm is required.

Select O_ID, P_ID, P_NAME, QTY
From Orders Join Inventory
On orders. product =Inventory. Product;

An increasing number of join queries are being executed by the interactive users and applications. In all the interactive applications the time to produce the first few results are very crucial. The join algorithms developed recently are mainly developed for data integration applications where the join algorithm should handle network latency, delays and source blocking. Many non-blocking join algorithms have been developed like Ripple Join [4, 7], PMJ [3], Symmetric Hash Join (SHJ) [10], extended version of SHJ called XJoin [10], extended version of XJoin called MJoin [2, 11].

Order O

O_ID	P_ID	C_ID	Qty
100	PM1123	10002	100
101	PM1123	20345	100
102	PM2212	56345	100

Inventor I

P_ID	P_Name	Stock
PM1123	Card Reader	100
PM1123	Flash Drivers	100
PM2212	HDMI Cable	100

Join O, I (O. Product-ID= I. Product-ID)

O_ID	P_ID	Product-Name	Qty
100	PM1123	Card Reader	100
101	PM1123	Card Reader	3
102	PM2212	HDMI cable	100

Figure 1. Example of the join relational operation.

But these algorithms are not optimized for the more predictable inputs in centralized database join processing and consequently, some optimizations to reduce the total execution time and CPU usage is not considered. Following are the family of algorithms which are designed for the predictable inputs in centralized databases. Nested loop join, in which the each row of the outer query is compared with each row of the inner query. Nested loop join is more suitable for smaller relations. If the cardinality of the relation in nested loop join is n and m then the complexity of the algorithm is $O(n^2)$. Sort merge join out performs the nested loop join and it performs better if the join attribute column is already sorted. Both the relation has to be scanned only once to produce the join result. The complexity of the sort merge join includes the sorting cost of both the relation *m* and *n* i.e., $O(n \log n)+O(m \log m)$. Hash join is faster than sort merge join, but puts considerable load on memory for sorting the hash-table. It offers advantages over the other traditional join algorithms for unsorted, non-indexed join input. Grace hash join, hybrid hash join and adaptive hash join are the modified version of hash join. In this paper we propose a bucket based join algorithm which will produce the first row join result without much delay. It consists of two phases: Matching phase and splitting phase. During the matching phase the first row of the outer table is compared with the inner table as a nested loop join algorithm. During this phase the unmatched rows between the matched rows are distributed in to different buckets, this process is called splitting phase. The matching and splitting phase happens in overlapped manner. Each bucket is associated with a header, which indicates the range of tuples available in the bucket. For the second row of the outer table the bucket header are compared to find out the bucket which contains the required tuple. The other buckets whose range value does not match with the joining tuple are not considered for the matching phase. The matched tuples in the inner table are discarded once the matching is formed. For each row of the outer table the matching and the splitting phase happens in interleaved manner. During each phase the number of buckets and the number of tuples in the buckets varies.

The organization of the rest of this paper is as follows. Section 2 explains the other join algorithms. Section 3 explains the block diagram of the proposed join technique called bucket join. Section 4 explains the Bucket join algorithm. Section 5 compares the performance of traditional join algorithms with the proposed join algorithm.

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 and which are followed by the new optimized 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.

In sort merge join algorithm, 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 is processed.

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.

Grace hash join [5] it has two pass, the relations are hashed into separate bucket which resided on disk. Each bucket is small enough to fit into memory. In second pass, a bucket from one relation is brought into main memory and hash table is constructed from it. Then, for each record in the second relation, its key is hashed and compared to every key which hashed to same bucket in the first relation.

Hybrid hash join [8] like other hash-based algorithm, uses hashing to improve the speed of matching tuples. That is, hashing is used to partition the two input relations such that a hash table for each partition of the smaller input relation can fit in main memory. Corresponding partitions of the two input relations are then joined by building an in-memory hash table for the tuples from smaller input relation, and then probing the hash table with the tuples from the corresponding partition of the larger input relation.

The shin's join algorithm [9] uses divide and conquer strategy; it repeatedly divides the source and target relations by a maximum of five functionally different hash coders and filters out unnecessary tuples whenever possible. After completing a division process, the algorithm checks whether or not the source tuples and the target tuples in a pair of source and target buckets have an identical join attribute. If so, the source and target tuples in the pair of the buckets are then merged in order to produce tuples for the resulting relation. Otherwise, the address of the current pair of source and target bucket is saved and the source and target tuples in the pair of buckets may be further divided by another functionally different hash coder. If a bucket is empty and the corresponding bucket in the pair is not empty, the tuples in the corresponding bucket are not necessary; thus they are discarded. The algorithm continues dividing the tuples in a pair of buckets, merging the tuples, or eliminating unnecessary tuples until every tuple in the buckets of created hash tables is either merged or eliminated.

Early, hash join [6] early hash join is based on symmetric hash join. It uses one hash table for each input. It consist of two phase reading and flushing. This algorithm dynamically customizes its performance to trade-off between early production of results and minimal total execution time.

G-join [9] g-join replaces the three traditional types of algorithms with a single one. Like merge join, this new join algorithm exploits sorted inputs. Like hash join, it exploits different input size for unsorted inputs. It matches the performance of the best traditional algorithm in all situations. If both join inputs are sorted, the g-join performs as well as merge join. If only one input is sorted, it performs as well as the better of merge join and hash join. If both the inputs are unsorted, it performs as hash join, including hybrid hash join. If both inputs are very large, it performs as well as hash join with recursive partitioning or merge join and external merge sort with multiple merge level.

3. Proposed Join Algorithm

3.1. Frame Work of the proposed Join Algorithm

Figure 2 shows the frame work of new bucket join algorithm. Let R_A (Outer relation) and R_B(Inner relation) are the source relations with n and m as the no. of rows with one-to-many relationship in the join attribute. For the first row of the outer table $R_A(1)$, the inner table is fully scanned to find the matching tuples. If the matching tuples are found in the inner table at the index k_i where i can vary from 1, ..., m. The unmatched rows between the matched indexes are distributed into different buckets as shown in Figure 3. Each bucket is associated with header which contains the range of join attribute available in the bucket. For the other rows of the outer table $R_A(i=2, ..., m)$ the bucket header is scanned to locate the bucket which contains the required matching tuple. The bucket which contains the required tuple is highlighted in the Figure 3. Once the bucket is found the matching phase is again repeated and buckets are updated. This process is continued until there are no more rows in the outer table.

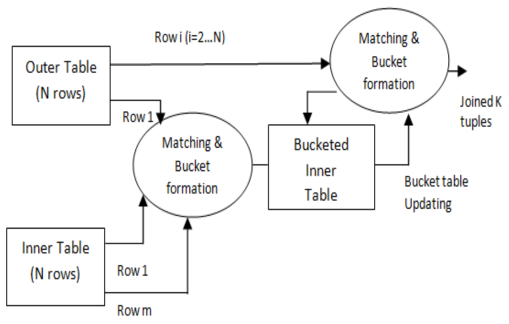


Figure 2. Frame work of new bucket join algorithm.

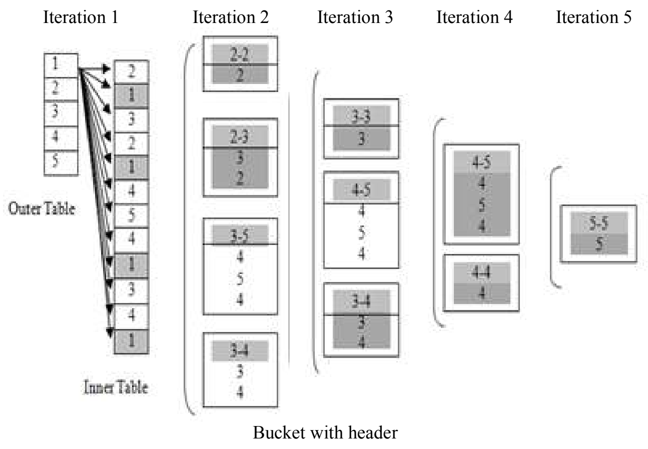


Figure 3. Matching and bucket formation phase.

3.2. Bucket Join Algorithm

In Algorithm 1 for the first row of the outer table the inner table is fully scanned to find out the matching rows. As seen in Figure 4 (step1) there are three matches for the first row of the outer table. The other unmatched rows between the three matched rows are distributed into different buckets. As seen in Figure 4 (step1) there are three buckets, each bucket is associated with a range value (min and max) which will indicate the range of tuples in each bucket. For the second row of the outer table there is no need to scan the inner table fully, instead the three bucket ranges are compared to locate the bucket which contain the matching row. Since the second row of the outer table's join attribute value falls in the range of all the three buckets, all the three buckets has to be scanned to find the matching tuple. In each bucket the unmatched rows between the matched rows are distributed into different buckets. As seen in Figure 4 (Step 2) the number of buckets increases to 4. For the third row of the outer table all four bucket ranges are scanned to find the possible bucket which will contain the matching row. As given in Figure 4 (step 2) bucket 2, bucket 3 and bucket 4 ranges do not match with the join attribute value of the outer table. Therefore there is no need in scanning the bucket 2, bucket 3 and bucket 4. Only bucket1 has to been scanned to find the matching tuple. The unmatched rows between the two matched rows in bucket 1 are distributed in to different buckets. Thus, the number of buckets reduces to 3. This procedure is iteratively repeated for all the rows of the outer table. For each step the number of buckets and the number of tuples in each bucket varies. During each phase of joining the number of tuples considered is reducing which will create a considerable reduction in the join cost.

Algorithm 1: Bucket join algorithm.

Input: R_A table with primary key, taken as the outer table. R_B table with foreign key, taken as inner table.
Output: $R_A \infty R_B$ result set Join

N_A: No. of rows in the outer table
N_B: No. of rows in the inner table e_f_t: Entering the loop for the first time, by default the value is true.
I, j, l, k, Z, i1 are the looping variables.

```
For each i in N_A do            // for each row of the outer table
    If (e_f_t == true) then
        for each j in N_B do    // compare all the rows of the inner
                                   table
            If R_A[i] == R_B[j] then
                R++;    // number of matches this Determines the
                           bucket size
                O[r] = j;
            end if;
        end for
        E_f_t=false
    end If;
    else
        for each i=1 in r do    // for the second row of the outer
                                   table it
        check the range of  the bucket, instead of full table scan
        If (R_A[i] >= min [i1] && R_A[i] <=max[i1])
            for each j in size[i] do
                If (R_A[i]==List[i1][j])
                    R++;
                    O[r]=j;
                End If;
            End for;
        End If;
```

```
 End for;
Sliding_window (array, matching);
z=0;
r=0;
j=0;
end for;
sliding_window(array,no_of_matches)
// Split the table rows into buckets and sets the range of  the bucket
 for k in r do
      y=o[k]
     for l in y do
          List[k][z]==R_b[l]
          z++;
     end for;
          x=y+1;
     min[k]=min(List[k])
     max[k]=max(List[k])
     size[k]=z;
end for;
```

Assume R and S are the two relations to be joined on join attribute A with cardinalities N_R and N_S respectively such that $N_R < N_S$. Attribute A in R is a set of n values$\{a_1, a_2, ..., a_n\}$. Attribute A in S is a multi-set of m elements of the form $\{a_1, a_2, ..., a_n\}$, each of which may have $\{x_1, x_2, ..., x_n\}$ copies, such that $\{x_1+ x_2+ ...+ x_n=N_s\}$In other terms, let a_k be all tuples which has join attribute value k such that the frequency $f(a_k)=Count(Tuples(a_k))$. The number of tuples with join attribute value j in $R(S)$ is denoted by $r_j(s_j)$. Thus,

$$|R| \sum_{j=1}^{N} r_j \text{ and} |S| \sum_{J=1}^{N} S_j .$$

For the tuple a_1 of R a full table scan of S is done. During the scan based on the position of the matching tuple a_1, table S is partitioned into B buckets and the header of the buckets are updated. The bucket header contains two parameter min and max, which indicates the range of tuples available in each bucket. In this method, the total number of comparisons is $1 + 2 (n\text{-}2)$ in worst case and $1+n\text{-}2$ in best case. In the above implementation, the worst case occurs when the elements are sorted in descending order and the best case occurs when the elements are sorted in ascending order. Therefore, the number of operations required to find the matching for row a_1 is given by:

$$Read(a_1)+ Read(a_1)+C_{Bucket\ Formation}+C_{Header\ Updation} \tag{1}$$

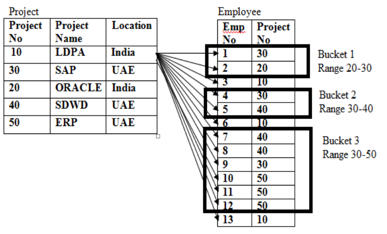
Where $Read(a_1)$: Reading the first tuple a_1 of, $Read(a_1)$: Full table scan of table S, $C_{Bucket\ Formation}$: Cost to form the buckets, and $C_{Header\ Updation}$: Cost to update the header.

For all the tuples a_i of R *for* $i\geq 2$, the bucket headers are scanned to locate the bucket which has matching tuples a_i. The number of buckets formed for each tuple a_i of relation R depends on the position of a_i in S. In other terms, it depends on the frequency of the tuple a_i in S, which canbe determined using histograms. Number of buckets N_B will be in the range $1\leq N_B$ $1\leq f(a_i)+1$. Number of tuples in each bucket B_r for $r\leq N_B$ depends on the position of attribute a_i in relation S. If $i_1, i_2, ..., i_m$ are the positions of matches found for tuple a_i in S then buckets $B_1, B_2, ...,B_r$ will contains $(i_2\text{-} i_1)$, $(i_3\text{-}i_2)$, ..., $(i_m\text{-} i_{m-1})$ number of tuples respectively. For all the tuplesa_i of R for $i\geq 2$, the bucket headers are scanned to locate the bucket which has matching tuples a_i. Only the buckets with the *min* and *max* values fall in the range of the join attribute value A of a_i is scanned to produce the resultant tuple. During the scanning process, the tuples in the buckets are further split to different buckets and the headers of the buckets are updated. Therefore, the number of operations required to find matching for the row a_i, for $i\geq 2$ is given by the following formula:

$$\sum_{i=2}^{N_R} B(a_i - 1) + \sum_{i=1}^{b} coun(B_i) + C_{Bucket\ Formation} + C_{Header\ updation} \tag{2}$$

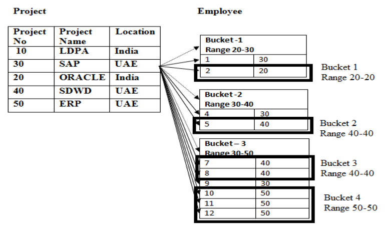
Where $B(a_i\text{-}1)$: No. of buckets after matching of tuple a_i-1, and $Count(B_i)$: Number of tuples in each bucket B_i.

Step 1:



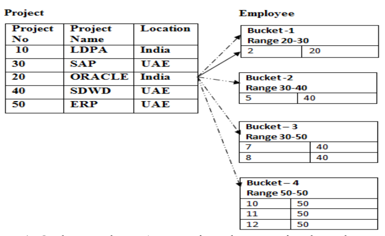
a) Full table scan.

Step 2:



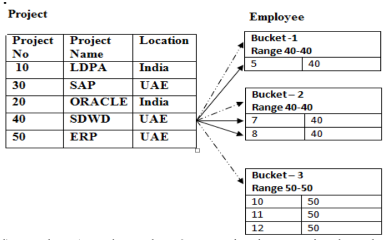
b) Bucket headers are scanned.

Step 3:



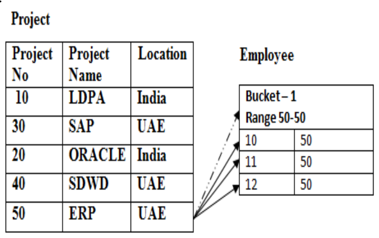
c) Only Bucket -1 contains the required tuple.

Step 4:



d) Bucket 1 and Bucket 2 contain the required tuple.

Step 5:



e) Bucket -1 contain the required tuple.

Figure 4. Steps in new bucket join algorithm.

4. Experimental Validation

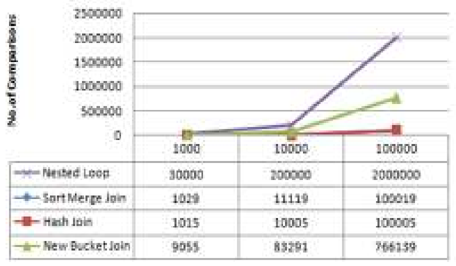
We ran the experiments on an Intel® corei5 2.50GHz processor, with 4GB real memory, running windows7 and Java 1.7.0_09. We have written a PL/SQL procedure to populate two tables Project (Project No, Project Name, Location) with 1000 rows and Employee (Emp No, Project No.) with 1,00,000 rows respectively. Project No. from Project table is the primary key and the Project No 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 tables Project and Employee are joined using the new join algorithm and during each stage the matched rows are eliminated or filtered out. This reduces in the no. of rows considered for each stage to produce the resultant set.

5. Results

The performance of the bucket join algorithm depends on the distribution of the values in the join attribute. The distribution of the data values affects the number of the buckets and the size of the bucket. This creates an impact on the number of the comparisons done during the join operation. Bucket join algorithm degrades when the join attribute values of the inner relation are non-uniformly distributed. In this case the algorithm will run with many buckets which will increase the number of comparison done. If the join attribute value is uniformly distributed and if the range of the tuples in the bucket is also distributed uniformly, the number of comparison required will be considerable less and it out performs all other traditional join algorithms.

This algorithm was executed for different numbers of tuples in the inner relation (1000, 10000, 100000) the number of comparison resulted during each iteration is projected in the Figure 5.

As seen in the graph hash join and sort merge join are better than the bucket join but they require some latency time to produce the first matching tuple. Hash join required to build the hash table for the outer relation and inner relation before it could produce the matching tuple. The time or the delay taken by the hash join to produce the matched tuples is given in Figure 6.



	1000	10000	100000
Nested Loop	10000	200000	2000000
Sort Merge Join	1029	11119	100019
Hash Join	1015	10005	100005
New Bucket Join	9055	83291	766139

Figure 5. One to many join: Number of comparisons.

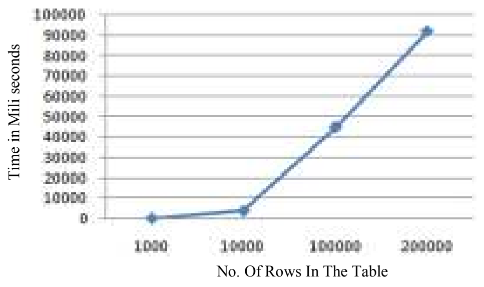


Figure 6. Time taken to build the hash table.

Similarly the sort merge join has to sort the join attribute columns of the inner table and the outer table before it produces the resultant set. The Figure 7 shows the delay taken by the sort merge join in producing the resultant set. In the nested loop and sort merge join the tuples are carried out till the end of the join operation even though they have no matching tuples. The number of rows considered during the join process remains the same in the nested loop join and the sort merge join. In the case of bucket join during each iteration the matched rows are removed and not considered for the next matching rows. The number of tuples considered during each iteration is given in the Figure 8. This graph is drawn with a sample size of 2500 rows in the inner table. The graph contains only the few iterations. As the graph shows the number tuples compared in each iteration is decreasing when compared to the first iteration.

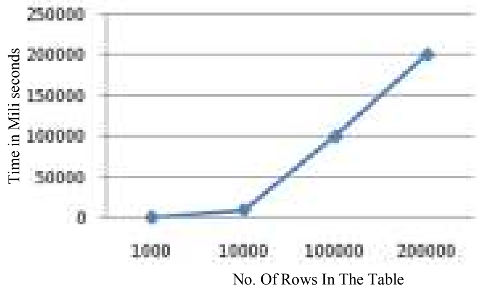


Figure 7. Time taken to sort the table.

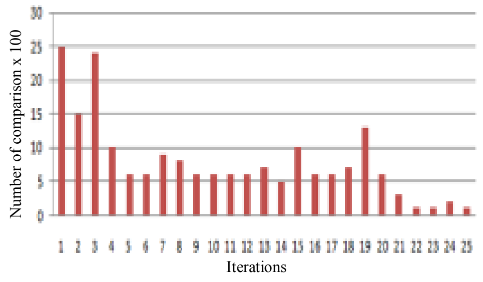


Figure 8. Number of tuples compared in each iteration.

The algorithm is also compared in terms of CPU usage, I/O reads and writes performed per sec. Figure 9 shows the CPU usage for the different algorithms. New bucket join save approximately 17% of CPU time when compared to sort merge join and nested loop join and 10% when compared to hash join. Hash join and sort merge join required 10% more I/O reads per sec than the new bucket join algorithm and the nested loop join as show in the Figure 10 Similarly the I/O writes are more for sort merge join and hash join when compared to the nested loop join and new bucket join as show in Figure 11.

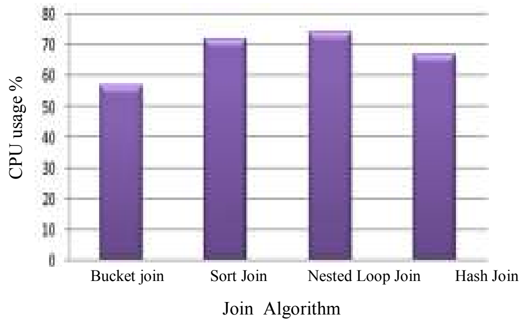


Figure 9. One to Many Join with 1, 00,000 tuples in inner table.

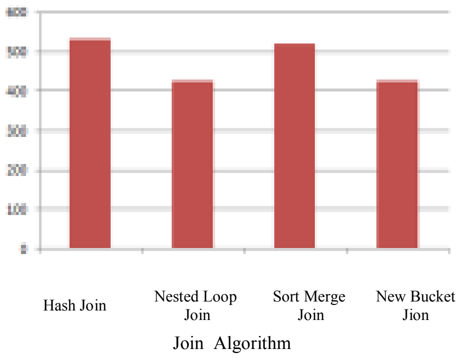


Figure 10. One to Many Join with 1, 00,000 tuples in inner table.

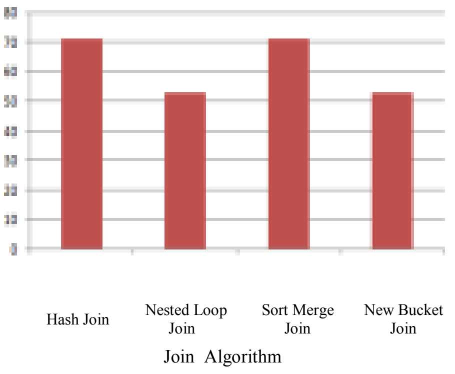


Figure 11. One to Many Join with 1, 00,000 tuples in inner table.

6. Conclusions

Bucket join algorithm best suits for interactive applications with rapid responds time and the minimal CPU and I/O operations when compared to the other types of hash join and sort merge join. The no. of comparison done in the bucket join is slightly more than the comparison required by the hash join and sort merge but it produces the initial result faster than the hash and sort merge join algorithms. Bucket join algorithm is significantly faster for one-to-many joins. The performance of the bucket join is affected by the distribution of the join attribute value. If the join attribute values are distributed uniformly, the tuples in the bucket will also be uniformly distributed. The range of the bucket will not be wide, this will reduce the no. of comparisons during the matching phase. Without any pre work and memory over head the bucket join can be used to produce the join results. For the future considerations bucket join can be extended to perform many-to-many join.

References

[1] Aljanaby A., Abuelrub E., and Odeh M., "A Survey of Distributed Query Optimization," *the International Arab Journal of Information Technology*, vol. 2, no. 1, pp. 48-57, 2005.

[2] Bornea M. A,vassalos V, Kotidis Y., and Deligiannakis A., "Adaptive Join Operators for Result Rate Optimization on Streaming Inputs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 8, pp. 1110-1125, 2010.

[3] Dittrich P., Seegar B., Taylor S., and Wigmaker P., "Progressive Merge Join: A Generic and Non-Blocking Sortbased Join Algorithm," *in Proceedings of the* 28th *International Conference on Very Large Data Bases*, Hong Kong, China, pp. 299-310, 2002.

[4] Haas J. and Hellerstein M., "Ripple Joins for Online Aggregation," *in Proceedings of SIGMOD*, New York, USA, pp. 287-298, 1999.

[5] Kitsuregawa M., Nakayama M., and Takagi M.., "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method," *in Proceedings of the 15th International Conference on Very Large Data*, San Francisco, USA, pp. 257- 266, 1989.

[6] Lawerence R., "Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results," *in Proceedings of the 31st VLDB Conference*, Trondheim, Norway, pp. 841-852, 2005.

[7] Luo G., "A Scalable Hash Ripple Join Algorithm," *in Proceedings of ACM SIGMOD*, Wisconsin, USA, pp. 252-262, 2002.

[8] Mokbel F., Lu M., and Aref G., "Hash-Merge Join: A Nonblocking Join Algorithm for Producing Fast and Early Join Results," *in Proceedings of the 20th International Conference on Data Engineering ICDE*, West Lafayette, USA, pp. 251-263, 2004.

[9] Shin K. and Meltzer C., "New Join Algorithm," *in Proceedings of ACM SIGMOD*, USA, pp. 13-20, 1994.

[10] Urhan T. and Franklin M., "XJoin: A Reactively Scheduled Pipelined Join Operator," *IEEE Data Engineering Bulletin*, vol. 23, no. 2, pp. 7-18, 2000.

[11] Viglas D., Naughton F., and Burger J., "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," *in Proceedings of the 29th International Conference on Very large data Conference*, Berlin, Germany, pp. 285-296, 2003.

Hemalatha Gunasekaran received her BE degree in computer science and engineering from Bharathiyar University in 2003 and ME degree in computer science and engineering from Karunya University in 2005. She has completed her PhD in communication and information technology in 2014 from Anna University, India. Currently, she is working as lecturer in Ibri College of Applied Sciences, oman. Her area of research includes: Query Optimization, Tuning and Big Data.

ThanushkodiKeppana Gowder received his BE degree in electrical and electronics engineering from College of Engineering, Guindy in 1972, and MSc (Engg.) degree from PSG College of Technology, Coimbatore in 1974, and PhD degree in power electronics from Bharathiyar University in 1991. Presently he is the Director of Akshaya College of Engineering and Technology, Coimbatore and he is a former Syndicate Member, Anna University of Technology, Coimbatore. His research interests are: Power electronics drives, electrical machines, power systems, and soft computing techniques, computer networks, image processing and virtual instrumentation.