CHAPTER 7

RE-INDEXING THE R*-TREE

Database systems are dynamic in nature. When number of events takes place in the database, the performance deteriorates. The overhead to keep the system and performance parameters at their optimal level at the occurrence of every event are very cost prohibitive. Hence, routine maintenance activities have to be carried out off-line. Some of such routine off-line maintenance activities are

a) Defragmentation of storage units,
b) Backups,
c) Collecting statistical information and
d) Re-indexing.

The above said activities are collectively called as post implementation performance tuning activities. Post implementation performance tuning is not directly addressed in the research literature. Commercial database software and utilities emphasize more on performance tuning.

7.1 Preamble

One of the most important routine database maintenance activities that are required for keeping the databases running at peak performance is re-indexing. When insertion, deletion and updating are performed dynamically, indexes can become fragmented. The parameter that primarily reflects the deterioration in the performance of the R*-tree during the dynamic behavior is the minimum node fill ratio. Minimum node fill ratio gives the minimum of the percentage of entries among all nodes in the R*-tree. The objective of this work is to propose a re-indexing technique for R*-tree that would result in increased minimum node fill ratio after re-indexing.

The basic algorithms and the properties of R*-tree guarantee a minimum node fill ratio of 50%, that is, every node will be at least half filled at any given point in time.
irrespective of any kind of activity that take place with the tree. The performance of the R*-tree is directly proportional to this ratio. Hence, any algorithm that is devised for R*-tree must ensure the improvement of this ratio. But the dynamic nature of the tree affects this parameter after implementation.

Algorithms are available in the literature to keep the minimum node fill ratio level at its optimal level, which is very near to 100%. But these algorithms are for trees that are static and do not allow insertion, deletion and updating after implementation.

Increasing this percentage is a challenge and few attempts were made in the research literature to address this issue. But a great emphasis is given in literature belonging to the commercial databases [267-270]. The methods that are generally used by the commercial databases can be grouped under two categories.

   a) Reconstruction of the Index with Random Reinsertions and
   b) Reconstruction of the Index with Ordered Reinsertions.

In the first method a new indexing structure is constructed by inserting the object in the currently fragmented structure in some random order into the new structure. In the second method the same is done after arranging the object in some order.

7.2 Proposed Re-indexing Algorithm

One of the important categories to which R*-tree belongs to is the recursive data structure. In this work, a new method is proposed that inspects the nodes at the each level of the tree and regroups them recursively thus removing the unnecessary nodes and packing the nodes to their capacity. Necessary adjustments are made at the intermediate levels of the tree according to the regrouping made at the lower levels. This process is termed as recursive re-grouping. Figure 7.1 illustrates recursive re-grouping. The process of re-grouping starts at the penultimate level. For each node at this level, the children nodes are combined and accommodated into lesser number of children nodes. For further optimizations the adjacent nodes are also explored. This also results in the grouping of parents and subsequently shrinking of the R*-tree. The parents
of the groups are then readjusted for their co-ordinates. This process is continued in the upper layers till the root is reached. The result of this process may be any one of the following:

![A R*-tree before re-indexing](image)

Figure 7.1a A R*-tree before re-indexing

![Re-indexed R*-tree in Figure 7.1a](image)

Figure 7.1b Re-indexed R*-tree in Figure 7.1a
a. The re-indexed R*-tree has same number of levels as that of the original R*-tree but has reduced number of nodes as illustrated in Figure 7.1 or

b. The re-indexed R*-tree has reduced number of levels than that of the original R*-tree as well as reduced number of nodes as illustrated in Figure 7.2.
Algorithm \texttt{NodeRegroup}(int \textit{CurrentLevel})

/* Algorithm to re-index a R*-tree */

1. \texttt{begin}
2. \quad \texttt{if}(\textit{CurrentLevel} \leq 1) \texttt{then}
3. \quad \texttt{for} every node in the \textit{CurrentLevel} of the R*-tree \texttt{do}
4. \quad \texttt{begin}
5. \quad \quad \text{Redistribute the children among minimum number of nodes;}
6. \quad \quad \text{if combination of adjacent nodes improve fill ratio} \texttt{then}
7. \quad \quad \texttt{begin}
8. \quad \quad \quad \text{Combine adjacent nodes;}
9. \quad \quad \quad \text{Redistribute the children among minimum number of nodes;}
10. \quad \quad \texttt{end;}
11. \quad \texttt{if} nodes at \textit{CurrentLevel} - 1 can be eliminated \texttt{then}
12. \quad \texttt{begin}
13. \quad \quad \text{Eliminate nodes to shrink the R*-tree;}
14. \quad \quad \text{Redistribute the children among minimum number of nodes;}
15. \quad \texttt{end;}
16. \quad \text{Adjust the coordinates of MBRs;}
17. \quad \texttt{call} \texttt{NodeRegroup}(\textit{CurrentLevel} - 1);
18. \texttt{end;}
19. \texttt{end;}

Figure 7.3 Algorithm to re-index R*-tree

In the first case the number of nodes that is searched for a query in the worst case would be lesser than that of the original tree. In the second case along with the advantage of the first case, the number of nodes that have to be accessed along a path is also reduced. The number of nodes in a level of a R*-tree, \( N_l \) is \( N_l = \frac{N}{f_{i+1}} \) where \( f \) is the
average fan out of the R*-tree. The expression shows that the number of nodes grow exponentially as the level of the R*-tree increase. For example, if the average fan out of the R*-tree is 4, then the number of nodes in level 1 is 1, the number of nodes in level 2 is 4, the number of nodes in level 3 is 16, the number of nodes in level 4 is 64. Hence, reducing the number of levels would bring down the number of nodes exponentially.

Figure 7.3 gives the algorithm for Re-indexing the R*-tree. The Algorithm is initially called with CurrentLevel as the penultimate level of the R*-tree. Redistribution uses the algorithms of R*-tree for seed picking, splitting and axis choosing.

7.3 Experimental Results and Discussion

Experiments were conducted to prove the effectiveness of the re-indexing proposed. The details of the experimental set up are given below:

The experiment was conducted using a 2-dimension data set of 250000 data points over the space $[0, 1]^2$. The maximum extent of the MBRs in each dimension was kept at 0.01 units. The lengths and the locations of the MBRs followed random distribution.

Two R*-trees were constructed, one with node size as 8 and the other with node size as 16. Insertions, deletions and modifications were made randomly. The total number of insertions that were performed on the tree was 250000. Apart from this 50000 deletions and 30000 modifications were also performed. The trees had 200000 data MBRs after all the insertions, deletions and modifications were over. The trees were then re-indexed using three methods namely, reconstruction using ordered insertion, reconstruction using random insertions and recursive regrouping. The following parameters were used in the evaluations:

a) Tree Height (TH),
b) Total Number of Nodes in the Tree (TNodes),
c) Total Number of Entries in the Tree (TEntries) and
d) Space Utilization (SU)
The computations of the parameters are as follows:

\[ TH = \text{count of the levels in the tree} \]

\[ TNodes = \text{count of the nodes in the tree} \]

\[ TEntries = \sum_{\forall \text{node } \in \text{tree}} (\text{number of entries in the node}) \]

\[ SU = \frac{TEntries}{(b\times TNodes)} \times 100 , \text{where } b \text{ is the node size} \]

The first comparison was made on the heights of the R*-tree. Using recursive regrouping, the number of levels was reduced by one, which is a significant improvement. The results are given in Figure 7.4. The reduction in the levels mean that the R*-tree is much smaller in height and for every path of the tree traversed for a query; the number of nodes accessed is reduced by one.

The second comparison was made on the total number of nodes in the R*-tree. The results are presented in Figure 7.5. While re-indexing by random insertion and ordered insertion reduced the number of nodes by 8.69% and 19.84% respectively, re-indexing by recursive regrouping produced 29.40% lesser number of nodes for a node size of 8 MBRs. While re-indexing by random insertion and ordered insertion reduced the number of nodes by 10.64% and 10.84% respectively, re-indexing by recursive regrouping produced 13.12% lesser number of nodes for a node size of 16 MBRs. Reduction in the number of nodes signifies that the minimum node fill ratio had improved and hence the nodes are packed heavily than before. This result may assist in converting the packed R*-tree and Hilbert packed R*-tree structures to dynamic structures.

The third comparison was made on the total number of entries in R*-tree. The result is graphically presented in Figure 7.6. Irrespective of the method that is applied to
re-index the R*-tree, the number of nodes at the leaf level will remain the same. The reduction happens at the intermediate levels of the tree namely, the directory nodes. The reduction in the number of directory entries implies that the number of paths that are traversed to answer a query will be less. While re-indexing by random insertion and ordered insertion reduced the number of entries by 1.67% and 3.81% respectively, re-indexing by recursive regrouping produced 6.59% lesser number of entries for a node size of 8 MBRs. While re-indexing by random insertion and ordered insertion reduced the number of entries by 1.02% and 0.09% respectively, re-indexing by recursive regrouping produced 2.26% lesser number of entries for a node size of 16 MBRs. Re-indexing using recursive regrouping begets lesser number of entries when compared with other methods of re-indexing. This effect combined with the reduction in the height of the tree, gives a good improvement in the overall performance.

The final comparison was made on the space utilization and the results are given in Figure 7.7. Again re-indexing using recursive regrouping performs better than other methods. While re-indexing by random insertion and ordered insertion improved the space utilization by 5% and 13% respectively, re-indexing by recursive regrouping improved the space utilization by 21% for a node size of 8 MBRs. While re-indexing by random insertion and ordered insertion improved the space utilization by 7% and 15% respectively, re-indexing by recursive regrouping improved the space utilization by 25% for a node size of 16 MBRs. The space utilization is an important parameter in the sense; it gives the measure of useful input-output operations that are performed to answer a query. Moreover, it also shows how optimally the space was utilized in the peripheral storage medium.

Legend for the graphs:
- BR: Before Re-indexing
- RI: After Re-indexing with Random Insertions
- OI: After Re-indexing with Ordered Reinsertions
- RR: After Re-indexing with Recursive Regrouping
Re-indexing methods

Figure 7.4 Comparison of the heights of the R*-tree before and after re-indexing

Figure 7.5 Comparison of the total number of nodes in the R*-tree before and after re-indexing
Re-indexing methods

- Node size: 8
- Node size: 16

Figure 7.6 Comparison of the total number of entries in the R*-tree before and after re-indexing

Re-indexing methods

- Node size: 8
- Node size: 16

Figure 7.7 Comparison of the space utilization percentage of the R*-tree before and after re-indexing
7.4 Summary

This chapter addresses the issue of re-indexing R*-tree. A new method, that recursively regroups the nodes has been developed and compared with the existing methods. Four parameters were used for comparison. The proposed method shows significant improvements in all the parameters when compared with other methods for re-indexing R*-tree.