CHAPTER 6

EFFICIENT PHYSICAL ORGANIZATION OF R*-TREE USING NODE CLUSTERING

R*-tree is a multidimensional indexing structure that forms basis for all the multidimensional indexing structures based on data partitioning. A number of attempts have been made in the past to improve the performance of R*-tree by manipulating the tree parameters and the data parameters. But few attempts had been made to use external parameters such as disk parameters to enhance the performance. This work proposes a new method to improve the performance of R*-tree by efficiently clustering the nodes into input-output units of the hard disk within the constraint that the independence between the logical and physical organization of the R*-tree should be preserved. Moreover, to preserve the structural and functional properties of R*-tree at any point in the process of clustering, a concept called controlled duplication has been introduced. Extensive experiments were conducted and the results are tabulated. The improvements are significant and open more avenues for exploration. Even though, a few works are reported in the literature on the issues regarding the storage medium [262-266], the cache [257, 259, 260, 262] and the main memory [258, 259, 261] for multidimensional data handling, this work is one of the first of its kind.

6.1 Disk Organization

A hard disk is a direct access storage device with large capacity. It is ubiquitous and part of every day computing system. A sector is the smallest addressable storage unit in a hard disk. This means, any storage object in the system takes at least one sector to get stored. This also implies that the data transfer between hard disk and the main memory is done sector by sector. But in present day computers transfer of data is done at a much higher volume and the term cluster is used to mention the storage unit. A cluster is a group of pre-defined number of sectors and is constant for a system. In this
chapter 'input-output unit' and 'storage unit' are used instead of cluster to avoid the confusion between R*-tree clusters and disk clusters. A track in a surface has many clusters in it. All the $i^{th}$ tracks in all the surfaces form the $i^{th}$ cylinder of the hard disk. The organization of a hard disk is given in Figure 6.1. With one read/write head for each surface of every platter, all the tracks of a cylinder can be read in one go after positioning the head at the required cylinder. The time taken to move the head from one cylinder to another cylinder is called as the seek time. Bringing the required cluster under the read/write head requires the rotation of the disk and the time taken for it is called latency time. Read/write time refers to the actual time taken to transfer data between a storage unit and main memory. Each storage unit in the hard disk can now be identified with hierarchical address $\text{cylinder#}.\text{surface#}.\text{cluster#}$.

Figure 6.1 Hard disk organization
6.1.1 The Hard Disk, Operating System and Database Nexus

In a computing system the hardware, operating system and the database software are completely independent of each other. The designers of each of these make their internal structure and functionality transparent and only provide interfaces through which others can avail the services. Each sub system optimizes its performance within itself. But when combined together to form a larger system the final throughput may not be the optimal. In such cases optimality could be achieved by trying to manipulate the parameters of the other correlated sub system in some way. In other words, the possibility of inter sub system optimization should be explored. This work makes one such attempt to improve the performance of the R*-tree that belongs to the database system by manipulating the parameters of hard disk system that is managed by the operating system. This attempt has greater importance in the context of distributed and heterogeneous database systems where the requirement of the maintenance of independence between storage model and logical model is imperative.

6.1.2 Need for Efficient Physical Organization

As discussed earlier, the research literature has abundant ramification of R*-tree. Every diversified method tries to improve the performance of the R*-tree by repeatedly manipulating the parameters of the logical model of the R*-tree. But ultimately the R*-tree is stored in the secondary storage medium which has its own performance parameters. The following issues are seldom addressed in the research literature.

a) physical organization,

b) mapping between logical organization and physical organization and

c) preserving the independence between the two organizations of the indexing tree structures.

However, the commercial database software developers and users place high emphasis on these issues. The importance of these issues is discussed in the context of performance tuning and is well documented in commercial database literature. It is evident from these documents that the commercial software are still working on B-trees and not on other tree based indexing structures. But of late they are adopting indexing
structures such as R*-tree, quad-tree, etc., instead of B-trees for indexing multidimensional data. Hence, it becomes imperative to come out with a model for efficient physical organization of R*-tree and its sequel.

In short, the objective of this work is to come out with a method that would improve the performance of an R*-tree by manipulating the data transfer unit of the storage medium and would still preserve the independence between the logical model and the storage model of the R*-tree. Clustering has been chosen as the method to achieve the objective. Attempting to cluster the R*-tree nodes into the data transfer units is not straightforward and gives rise to a lot of technical and implementation issues. A new controlled duplication method has been proposed to handle these issues. The details are given in the subsequent sections.

6.2 Objectives of Node Clustering

Let \( s_1, s_2, \ldots, s_n \) be the \( n \) steps required to access the smallest addressable input-output unit of the storage organization. Let \( s_1 \) be the first step and \( s_n \) be the last step. Let \( t_i \) be the average time taken to execute the \( i^{th} \) step. Let \( t_1, t_2, \ldots, t_n \).

Let \( R \) be the given R*-tree, \( L \) be the number of levels of the tree, \( N_j \) be the number of nodes in \( j^{th} \) level and \( p \) be the number of input-output units occupied by a node. If the number of bytes in a node is less than the number of bytes in an input-output unit, then \( p \) is 1. The time taken to retrieve any node from the storage organization is given by

\[
T = p \sum_{i=1}^{n} c_i t_i
\]  

where \( c_i \) is a constant for the \( i^{th} \) step. Let \( Tot \) be the time taken to access all the nodes of the tree. Then, \( Tot \) is given by

\[
Tot = \sum_{i=1}^{L} (N_i \ast T)
\]

The above formula is also applicable to any sub tree of the given tree. Now, the objective of retrieval is to minimize \( Tot \), which is possible if some method can be

100
devised that reduces $T$. The following discussion forms the basis of the methodology proposed in this chapter for improved physical organization of the R*-tree that minimizes $Tot$.

Let $x_1, x_2, ..., x_z$ be $z$ nodes of $R$ to be retrieved. Let $j < n$. If the first $j$ steps are common for the retrieval of $x_1, x_2, ..., x_z$, then the time taken to retrieve the nodes is

$$\left(1 \sum_{k=1}^{j} c_k t_k\right) + \left(z \sum_{q=j+1}^{n} c_q t_q\right)$$

Since $t_1, t_2, ..., t_n$, the above formula always gives a lesser value than

$$z \sum_{k=1}^{n} c_k t_k$$

Obviously,

$$z_1 \sum c_1 t_1 + (z_2 \sum c_2 t_2) + ... + (z_{n-1} \sum c_{n-1} t_{n-1}) + (z \sum c_n t_n) \leq z \sum_{k=1}^{n} c_k t_k$$  \hspace{1cm} (6.3)$$

where $z_1, z_2, ..., z_{n-1}$ are less than $z$ and $z_0, l \leq n-1$ is the number of nodes that require the step $c$ to be performed. The above equation considers the fact that at least one step, that is step $n$ in the equation, is not common for all the nodes that are required to be retrieved.

From the above discussion it is clear that, if a suitable method can be found out that group those nodes that would reduce the number of retrieval steps, a higher performance of the system could be achieved. Each group of nodes is called as a *cluster*. The result of clustering is a *cluster graph*. The *successors* of a cluster in a cluster graph are the collection of all the clusters that have the children of all the nodes in that cluster. The *predecessors* of a cluster in a cluster graph are the collection of all the clusters that have the parents of all the nodes in that cluster. A *cluster tree* is a cluster graph whose clusters do not have more than one parent. A *cluster access cycle* is defined as the set of steps to access a complete cluster and *cluster access time* is the time taken to complete a cluster access cycle.

By equation 6.3, it is expected that the cluster access time will be less than the total time taken if the nodes are accessed separately. A cluster is accessed whenever a
node in that cluster is required. Hence, during clustering, the prime objectives would be to cluster the nodes in such a way:

a) to reduce the number of times each cluster is accessed during a database operation and

b) that the initial steps are not repeated for every node of the tree.

Figure 6.2 A skeletal R*-tree

Clusters: (HDI) (BJE) (KAL) (FMC) (NGO)

Figure 6.3 Clustering of R*-tree in Figure 6.2 based on in-order traversal of nodes

Consider the R*-tree given in Figure 6.2. Figure 6.3 gives one of the possible clustering and the corresponding cluster graph that is based on in-order traversal of nodes. Figure 6.4 gives another possible clustering and the possible cluster graph. This
clustering results in a tree structure. Comparing Figure 6.3 and Figure 6.4, it is evident that if clustering results in a cluster tree then every cluster is accessed only once during searches. This is due to fact that any vertex has only one path to any of its descendants in a tree. On the other hand, if clustering results in a graph then any node with multiple predecessors has to be retrieved more than once during search. Hence, it is desirable that clustering ends up in a cluster tree rather than a cluster graph. An implication that is worth mentioning would be that there is a chance for the formation of a cluster graph if there is more than one tree in a cluster, but not always.

Clusters: (ABC) (DHI) (EJK) (FCM) (NGO)

Figure 6.4 Clustering of R*-tree in Figure 6.2 that maintains a tree structure

6.3 Implementation Issues of Clustering

With the discussions of the previous section in the background, this section proposes a new method to efficiently organize the R*-tree in the permanent storage device that would minimize the access time of the nodes of the R*-tree.

Let an arbitrary cluster contain a node $n_a$, and its children, $n_{al}$, $n_{a2}$, ..., $n_{ab}$. If there exist some nodes that are not accessed after accessing $n_a$, their sub trees too will not be accessed and also the clusters containing them. All $n_{al}$, $n_{a2}$, ..., $n_{ab}$ would not be accessed only if the process ends up in a dead space in $n_a$. Furthermore once a cluster is accessed, every node in the cluster is processed and would not be required at a future time. This implies that this cluster need not be retrieved once again. If instead of one of the children of $n_a$, a grandchild of $n_a$ is in the cluster, then possibility of a cluster graph arise that results in unnecessary retrieval as in the case of Figure 6.3. It is also desirable to form clusters using nodes that have a high probability of retrieval immediately after
In an R*-tree only a node’s children have the higher probability of retrieval immediately after its retrieval. Clustering the parents and children will result in eliminating a few initial steps to retrieve these multiple nodes. Obviously, clustering only one parent and all its children or only all the children of a parent in a cluster results in another R*-tree as shown in Figure 6.4. In such a clustering, if the cluster size matches the input-output unit size the implementation is straight forward. But the freedom to choose the input-output unit size is not with database administrators. This gives rise to very important implementation issue that has four cases.

a) Only one node of the R*-tree accommodated in one input-output unit of the hard disk,

b) Exactly \( M+1 \) nodes of the R*-tree accommodated in one input-output unit of the hard disk,

c) Less than \( M+1 \) nodes of the R*-tree accommodated in one input-output unit of the hard disk and

d) More than \( M+1 \) nodes of the R*-tree accommodated in one input-output unit of the hard disk.

Figure 6.5 gives pictorial overview of the four cases for \( M = 4 \). Among the four cases, case 1 is trivial and would result in a typical R*-tree. Figure 6.5b demonstrates case 1 for the two sub trees of an R*-tree given in Figure 6.5a. Case 2 does not create any significant implementation issue as discussed earlier and is demonstrated in Figure 6.5c. Case 3 requires some innovation during implementation. Case 4 may be treated as a special case of case 3 or combination of case 2 and case 3. Figure 6.5e demonstrates this case. Hence, the implementation issues of case 3 are discussed in detail here.

The discussions in the previous sections suggest that, optimal performance is obtained only when

a) clustering should end up in a structure similar to an R*-tree and

b) \( M \) should only be as big as allowing \( M+1 \) nodes into an input-output unit.
Figure 6.5a Two sub trees of the same parent

Figure 6.5b Clustering of one R*-tree node in one disk input-output unit

Figure 6.5c Clustering of $M+1$ R*-tree nodes in one disk input-output unit

Figure 6.5d Clustering of less than $M+1$ R*-tree nodes in one disk input-output unit
Figure 6.5e Clustering of more than $M+1$ R*-tree nodes in one disk input-output unit

Figure 6.5f Elimination of node created during controlled duplication

With the independence of database software and the operating system in place, satisfying both is not possible. Hence, to achieve this controlled duplication of nodes are allowed during growing and shrinking phases of the R*-tree. Controlled duplication necessarily preserves the R*-tree structure both inside clusters and across clusters. Consider a cluster size of three nodes as in Figure 6.5d. The cluster size is less than $M+1$, for $M = 4$. In the clustered R*-tree, nodes A and F would be duplicated and available in two clusters as shown. These duplications are controlled in the sense; the duplicates remain in the tree only as long as the necessities for them to be split arise. Consider the case of node A. If one of the nodes among B, C, D and E overflow, then A would also overflow. In such a case a new node has to be created and the MBRs in the overflowing node A, must be distributed among them. Now, the duplicate of A in an appropriate cluster may be renamed and used as a new node instead of creating a new one. This results in the elimination of one of the duplicates of node A that was
**Procedure Cluster_Insert**
/* Algorithm to insert a node into a cluster */
1.  begin
2.         if Cluster Overflows then Create a new cluster;
3.         if duplication of any parent required then
4.         begin
5.             Duplicate parent;
6.             Distribute children among duplicate parents;
7.         end;
8.         Update the parents of nodes of affected clusters ; Realign tree;
9.  end;

**Procedure Node_Insert**
/* Algorithm to insert a MBR into a node */
1.  begin
2.         if Node does not Overflow then Update Parents till Root
3.         if Node Overflows then
4.         begin
5.             Create new node and Distribute MBRs;
6.             Adjust Parents till Root; call Cluster_Insert;
7.         end;
8.  end;

**Algorithm INSERT_MBR**
/* Algorithm to insert a MBR in a clustered R*-tree */
1.  begin
2.         Select Leaf Node to Insert MBR;
3.         Insert MBR; call Node_Insert;
4.  end;

Figure 6.6 Insertion algorithm
**Procedure Cluster_Delete**

/* Algorithm to delete a node from a cluster */

1. begin
2. if Cluster does not underflow then return;
3. if Cluster underflows then
4. Merge adjacent clusters without affecting cluster or R*-tree properties and realign tree;
5. end;

**Procedure Node_Delete**

/* Algorithm to delete a MBR from a node */

1. begin
2. if Node does not Underflow then Update Parents till Root;
3. if Node Underflows then
4. begin
5. Merge adjacent nodes and Redistribute MBRs;
6. Adjust Parents till Root;
7. call Cluster_Delete;
8. end;
9. end;

**Algorithm DELETE_MBR**

/* Algorithm to delete a MBR from a R*-tree */

1. begin
2. Select the leaf Node where the MBR to be deleted is present;
3. Delete MBR;
4. call Node_Delete;
5. end;

Figure 6.7 Deletion algorithm
previously created. The new scenario that appears when A is split due to the
overflowing of E is given in Figure 6.5f. In the figure, E splits into E and E' while A
splits into A and A'. The figure also shows how the realignment of nodes takes place
when the tree grows upward by one level.

During realignment, if the number of levels in the tree is odd, the clusters that
hold the leaves will not have their parents in them. These clusters only become forests
of leaves. The clusters that hold the intermediate nodes become forests of trees with
two levels. On the other hand, if the number of levels is even, every cluster contains
forests of trees with two levels. The insertion and deletion now becomes a two step
process. The initial processing is at the node level and the subsequent processing is at
the cluster level. Skeletal insertion algorithms are given in Figure 6.6. Skeletal deletion
algorithms are given in Figure 6.7.

6.4 Experimental Results and Discussion

The performance of an R*-tree depends on the number of input-output
operations that are done from and to hard disk during insertion, deletion, updating and
searching. The number of input-output operations can be significantly reduced by
efficiently clustering the nodes in an R*-tree.

For the experiments, an input-output unit with eight sectors was chosen, that is,
4K bytes. A node size in the R*-tree was fixed at eight MBRs. Six different 2-
dimensional MBR sets with 25K, 50K, 75K, 100K, 125K and 150K MBRs in each set
was generated for the experiments. Uniform location distribution and uniform length
distribution were followed for both axes of the MBRs. The MBRs were generated in a
unit space \([0, 1]^2\). The maximum length of the MBRs was fixed at 0.2 units. Since the
values were stored in character mode, each MBR entry took 60 bytes in the disk. Apart
from MBR entries, each node also had other necessary entries for the management of
the R*-tree. Clustered R*-trees were constructed for every set of MBRs by changing the
number of nodes in a cluster from 1 to 32. If a node takes more than one input-output
unit, they were chosen in a way that would minimize the total time taken to access all
the input-output units together instead of one by one, i.e., the seek time and rotation
time are constant for all the input-output units in the cluster. The parameters that were
used for comparisons were

a) Total number of nodes accommodated in each disk input-output unit (DIP),

b) Space utilization percentage (SUP),

c) Total Number of clusters formed (TC) and

d) Speedup (SU).

The computations of the parameters are as follows:

\[ \text{DIP} = \frac{\text{Size of disk IO unit}}{\text{Size of a cluster}} \]

\[ \text{SUP} = \frac{\sum_{\text{cluster}} \text{size of space occupied by nodes in bytes}}{\sum_{\text{cluster}} \text{size of a cluster in bytes}} \times 100 \]

\[ \text{TC} = \text{Count of the clusters formed for the R*-tree.} \]

\[ \text{SU} = \frac{(T_{wc} - T_{ac})}{T_{wc}} \times 100 \]

where \( T_{wc} \) is the time taken for traversing the R*-tree without clustering and
\( T_{ac} \) is the time taken for traversing the R*-tree after clustering.

The number of input-output units taken for various cluster sizes is given in
Figure 6.8. When the cluster size is one, a trivial R*-tree is constructed. In such a case
the maximum space taken by a node/cluster in an input-output unit is 510 bytes. The
remaining space goes waste. By clustering nodes into input-output units more space
utilization is obtained. This is shown in Figure 6.9. As more and more R*-tree nodes are
packed into input-output units better space utilization is achieved. The number of
clusters formed for varying number of nodes in a cluster for various data sets are given
in Figure 6.10. It has been well established in the previous sections that time taken to
retrieve a cluster is much smaller than the sum of times taken to retrieve each node of a
cluster individually. The graph in Figure 6.10 gives the amount of time that could be
saved by clustering. The speedup achieved after clustering for traversing the tree is
given in Figure 6.11.
Figure 6.8 Number of disk input-output units taken for various cluster sizes

Figure 6.9 Space utilization for various cluster sizes
Figure 6.10 Number of clusters formed for various cluster sizes for six different data sets

Figure 6.11 Speedup after clustering in six different data sets
6.5 Summary

R*-tree is a multidimensional data partitioning indexing structure that has become the basis of all the indexing techniques in the future that used data partitioning. While a lot of research had gone into the improvement of this structure from data characteristics, and tree characteristics view points, few attempts have been made from physical storage view point. This work basically attempted at improving the performance of R*-trees from physical storage view point and has come out successfully. The methods and algorithms proposed here are applicable for every ramification of R*-tree.

All along this work, the basic and very important constraint of maintaining the independence between logical and physical organization of the R*-tree was given due focus and every result provided is within this important constraint. This independence gives the designers and administrators of databases complete freedom to design the logical solutions without any hindrance from the physical design of the system and vice-versa.