A State-of-Art in R-Tree Variants for Spatial Indexing

Lakshmi Balasubramanian
Department of Computer Science and Engineering
Pondicherry Engineering College
Puducherry, India.

M. Sugumaran
Department of Computer Science and Engineering
Pondicherry Engineering College
Puducherry, India.

ABSTRACT
Nowadays, indexing has become essential for fast retrieval of results. Spatial databases are used in many applications which demand faster retrieval of data. These data are multi-dimensional. Designing index structure for spatial databases is current area of research. R-Tree is the most widely used index structure for multi-dimensional data. Many variants of R-Tree has evolved with each performing better in some aspect like query retrieval, insertion cost, application specific and so on. In this work, state-of-art of variants in R-Tree is presented. This paper provides an idea of the present development in spatial indexing and paves way for the researchers to explore and analyze the difficulties and trade-offs in the work. The R-Tree variants are classified according to the way they are different from the original R-Tree either in the process of construction or whether it is a hybrid of R-Tree and some other structure or whether it is an extension of R-Tree to support many other applications.

General Terms
Database Management, Spatial Databases, Spatial Indexing and Queries

Keywords
R-Tree, Spatial Index, Spatio-Temporal Index, R-Tree variants

1. INTRODUCTION
Database management is one of the important research inventions in the field of computer science. Databases store the information in some form so that the retrieval of data on necessity is easy. Databases store data like details of employee, students, patients and the like; stock and inventories of various fields; history of some events to evaluate the increase or decrease in performance etc. Nowadays databases store data which are more than one dimension also. Examples of these data include multimedia data, spatial data, spatio-temporal data etc. The applications that make use of these data enforce storage of large volume of data which may range even millions to billions. So, efficient handling of these data is necessary. Indexing helps retrieving data faster than the actual. Indexing spatial data is an important area of research in recent times. Spatial indexing benefits applications [26] like Geographical Information System, Computer Aided Designing and Multimedia etc. Many spatial index structures like quad tree, k-d tree, R-Tree and grid files etc., have been proposed in the literature. R-Tree is found to be the most widely used spatial index structure for its efficiency in performance and simplicity in implementation. In this index structure, the objects are represented as Minimum Bounding Rectangles (MBRs). R-Tree consists of root, non-leaf and leaf nodes. The nodes in R-Tree have a maximum and minimum capacity. When a node reaches the maximum capacity, the node is split such that the number of objects in each node lies between the minimum and maximum limit. Different variants of R-Tree have been proposed in the literature to give better performance in case of some queries or applications. The benchmark queries in spatial domain include range, nearest neighbor, join, topological and nearest surrounder queries. There are many variants in each of these queries.

Different variants concentrate on optimal insertion, construction, splitting, range query retrieval, topological query retrieval or queries related to direction and so on. The variation can be a slight modification of the index structure or combination of other index structures with the R-Tree such that the advantages from both are exploited.

R-Trees can also be extended to support extra features like support for storage of details about moving injects. Thus the R-Tree index structure can be modified to provide temporal support. This enables it to be considered as a spatio-temporal index too. Applications [26] such as Radio Frequency Identification, Mobile Computing, Geographic Positioning System, Traffic Control Systems, Sensor Based Networks and Location Based Services etc., benefit from this index structure. The spatio-temporal domain enforces need for a wider variety of queries. The queries can be related to any point of time (past, present or future) or any time interval.

The indices are designed such that the overhead due to its maintenance is less and query retrieval becomes easier. The index saves the number of disk access and time in retrieving results for a query. The index should be such that it is beneficial for most variety of queries. In spatial domain the variety of queries is more than that of the normal databases. They include range queries and their variants, nearest neighbor query and their variants, join queries, nearest surrounder queries etc. The spatio-temporal domain increases this variety to a greater extent. They enforce queries at any time instant. The queries can be from past, present or future positions of the object. The time can be a time instant or time interval. There also arises a new variety of query called the trajectory query which describes the path taken by the object.

In this paper, a survey on the different R-Tree variants is presented. The organization of the paper is as follows. Section 2 explains the basic R-Tree index structure and the related algorithms. Section 3 presents the state of art in the R-Tree variants and section 4 gives the conclusion.

2. R-TREE INDEX STRUCTURE
The most widely used index structure for indexing spatial data is the R-Tree. The efficiency of R-Tree is due to its simplicity. The properties and basics of R-Tree are explained below.

2.1 Index Structure
R-Tree [2] is a height balanced tree similar to B-Tree. Each node corresponds to a disk page in the disk. The structure is

designed such that a search requires visiting only a small number of nodes. The index is completely dynamic: the inserts and deletes can be intermixed with the searches.

The leaf nodes contain the unique identifier of the object, MBR of the object, and pointer to the actual data. MBR is the smallest rectangle that covers the object. The non-leaf nodes contain the MBR which covers all its children and pointer to the children. A set of properties [2] must be satisfied for the structure to be a valid R-Tree. The properties are:

- If *M* is the maximum limit that a node can hold, then lower limit *m* should be such that $m \leq \frac{M}{2}$.
- Each leaf node should contain between *m* to *M* index records.
- Each index node should have between *m* to *M* children unless it is the root.
- The MBR should be the smallest rectangle that covers the total area covered by its children in the case of index nodes.
- The root node should have at least two children unless it is a leaf.
- All leaves must appear on the same level.

The data can be inserted, deleted and updated in the R-Tree. When the node reaches its maximum limit, the node is said to overflow and hence it is split such that the resulting structure satisfies the above properties. There exist many node splitting algorithms in the literature. Linear, quadratic [2], R* [4], branch grafting [26], 2-3 splitting [7] are some of them. The following section briefs the process of searching, insertion, deletion and updating of data to the node and splitting the node.

2.2 Searching, Inserting, Deleting, Updating and Splitting

Inserting records to the R-Tree should be such that properties of the R-Tree are satisfied before and after the insertion. When a record is to be inserted and the tree has only one node, then insert the record to that node. If there is more than one node, then traverse the tree from top to bottom until a non leaf node which can accommodate the entry with least enlargement of MBR is obtained. Then insert the object and propagate the changes upto the root. The above case is fine if the number of entries lies between minimum and maximum limit. If on insertion the numbers of entries exceed the maximum limit, then the nodes are split such that each node has entries between minimum and maximum limit.

Searching records which lie within a given window is the most often used query. The tree is always traversed from top to bottom. The non-leaf records which overlap the given window are chosen during traversal for further exploration. This is done at every level until the leaf nodes are reached. The leaf nodes which overlap with the given window are the qualified records.

Deleting a record also should satisfy the properties before and after deletion. The particular record to be deleted is searched using the search algorithm. After the record is found, the record is deleted. The changes are then propagated upto the root. There are chances for underflow (i.e.) the number of records becomes less than that of the minimum capacity. During underflow, the nodes are merged to satisfy the properties. Updating a record involves deletion of the old values and insertion of new values.

3. R-TREE VARIANTS

Variations of R-Tree evolve in the motive of increasing the performance in some aspect or to provide support to additional applications. In this work, the variants are categorized based on process change, hybridization and extension. The classification is given in Fig. 1. Each variant is described briefly and its purpose is mentioned. The following subsections presents on each category of classification.

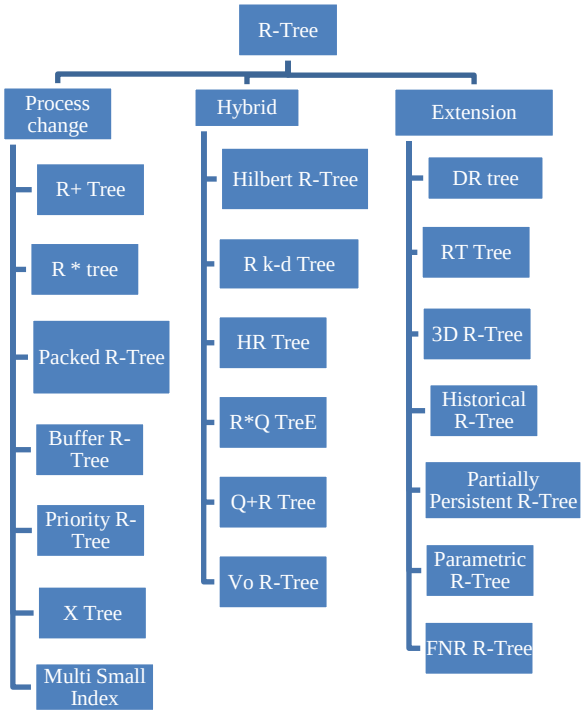

Fig 1: Classification of R-Tree Variants

3.1 Variations Based on Process Change

Process change may refer to the change in the process during any step of construction. The variants include R+ Tree [3], R* Tree [4], Packed R-Tree [6], Buffer R-Tree [8], Priority R-Tree [24], X-Tree [10] and Multi Small Index [33]. Each differs from one another in their construction.

3.1.1 R^+-Tree

The R^+-Tree [3] variant was introduced in the motivation to increase search performance especially the point queries. It reduces the search cost by reducing the number of disk accesses needed to get the result. Overlap and coverage area are the two important concerns for search performance. The main concept of this variant is to have zero overlap. If any data rectangle at the lower level overlaps with another rectangle, it is decomposed into a collection of non overlapping sub-rectangles whose union makes up the original rectangle. The insertion algorithm should add the rectangle to more than one node by dividing into sub-rectangles during overlap. The deletion algorithm also needs to search the entire sub rectangles which make the original rectangle and delete all of them. Searching is made easy as the multi-path search due to overlap is avoided here. Splitting should be propagated both upward and downward for the stability of the tree.

3.1.2 R-Tree*

R-Tree concentrates mainly on minimization of area covered by it. There were many other aspects to be concentrated for best performance. The aspects are minimization of overlap,

area covered by the directory nodes (dead space should be minimum), margin (square gives the minimum margin for the given area) and maximizing storage utilization. The variant R* [4] is built by incorporating a few changes in concepts to the basic R-Tree. The changes are 1) while choosing sub-tree, if the node is nonleaf and points to the leaf then minimum overlap (area is considered only during tie) enlargement is considered. If the non leaf node does not point to leaf then the node which needs least area enlargement is chosen. This increases the performance of small window queries on non uniform data. 2) Splitting is based on the goodness value which is determined by area, margin and overlap. The axis for split is chosen by margin value, in that axis, the distribution which has minimum overlap (area during ties) is chosen and split accordingly. 3) Since splitting is a costly process, splits are reduced by forced reinsertion. Whenever a node overflows, some entries (30%) of the overflowing node are reinserted into the tree. This improves storage utilization and shape of the nodes is more likely to be quadratic. R*-Tree [27] was found to outperform R-Tree for all queries (especially queries with small window). It provides a good performance for join queries too.

3.1.3 Packed R-Tree

Packing [6], a bulk loading technique is used to populate the R-Tree. This technique is useful for static R-Trees. The packing algorithm first sorts the rectangles by x-coordinate of their centers. Then the rectangles is packed into groups such that all except the last group contains *c* rectangles. Calculate the MBR of each group and assign them to a node. The process is continued recursively. Hibert Packed R-Tree [7] and STR R-Tree [11] were derived from this basic approach. Hilbert Packed R-Tree uses the Hilbert value of the centroids of the rectangles for sorting. In STR R-Tree, the rectangles are first sorted by x co-ordinate of their centers and vertical slices are created. Then within each slice, the rectangles are sorted by their y co-ordinate and then packed into the node. STR R-Tree is found to have less overlap compared to other variants.

3.1.4 Buffer R-Tree

Buffer R-Tree uses the buffering technique by exploiting the main memory and reduces the number of disk I/O operations. Using formulae and calculations in [8] buffers of calculated size are attached to all the nodes of the calculated level. Any operation to the tree can be performed in a lazy manner. In case of insertion, it need not take place immediately. The buffer at the root collects a block of rectangles. When the buffer becomes full, the next lower buffer node is searched to hold each of the rectangle that was present in the root buffer node which was full. This process continues until the data reaches the leaf node. Once it reaches the leaf node, the insertion takes place. This variant performs well for bulk insertion, deletion, updating and queries.

3.1.5 Priority R-Tree

Priority R-Trees (PR-Tree) [24] were derived from Pseudo Priority trees. Pseudo Priority trees are similar to R-Tree in that the parent node stores the MBR enclosing all the MBRs of the children but they are not height balanced (leaves need not be in same level) and each node has degree equal to six. Each rectangle is considered as a four dimensional point (x_{min}, y_{min}, x_{max}, y_{max}). Out of the six children for each node, four are called priority leaves which store the boundary rectangles (one stores the rectangles with x_{min}, the second with y_{min} and so on). The rest of two children recursively form pseudo priority trees. To derive a PR-Tree from a pseudo-PR-Tree, the PR-Tree has to be built into stages, in a bottom-up fashion. First, the leaves are created and the construction proceeds to the root node. At stage *i*, first the pseudo-PR-Tree is constructed from the rectangles of this level. The nodes of level *i* in the PR-tree consist of all the leaves of pseudo PR tree formed at that stage (i.e) the internal nodes are discarded. The authors have provided worst case analysis for bulk loading and range query retrieval for *d* dimensions [24]. The PR-Tree provides worst-case optimality, because any other R-tree variant may require (in the worst case) the retrieval of leaves, even for queries that are not satisfied by any rectangle. The bulk-loading of a PR-Tree is slower than the bulk-loading of a packed 4D Hilbert tree. However, regarding query performance, for nicely distributed real data, PR-Trees perform similar to existing R-tree variants. In contrast, with extreme data, PR-Trees outperform all other variants, due to their guaranteed worst-case performance.

3.1.6 X-Tree

X-Tree variant [10] supports indexing high dimensional data with less performance degradation. It uses the concept of super node and minimum overlap. When a node overflows, overlap factor for the split is calculated. If it is within the threshold then the nodes are split, otherwise the maximum capacity of the node is increased double its value (super node). The basic idea is that linear traversal gives better performance than multipath traversal due to overlap. This variation is same as R-Tree if there is no super node and linear if there is only one super node.

3.1.7 Multi Small Index

Single Big Index is indexing the objects in whole data space. R-Tree is a Single Big Index (SBI). Multi Small Index (MSI) [33] is dividing the data space into many partitions and indexing each partition. MSI performs better in terms of disk accesses than SBI for small queries. The concept of MSI starts with dynamic recursive partitioning is as follows: There exists a user threshold which limits the number of objects. When the number of objects exceeds the threshold, the space is divided into four subspaces called quartiles. Then each quartile is checked for the threshold condition. If those quartiles exceed the threshold they are again divided into quartiles. The process continues until the quartiles contain objects less than the threshold. The subspaces with objects less than the threshold are called target spaces and R-Tree is constructed for each target space.

3.2 Hybrid Variations

Variants were developed by combining the concepts of different index structures which may include quad trees [1], hash index, Hilbert curve, k-d Tree, R-Tree, R*-Tree, Voronoi diagrams etc. The variants presented below are Hilbert R-Tree [7], R k-d Tree [28], R*Q Tree [20], Q+R Tree [21], HR Tree [29] and Vo R-Tree [32].

3.2.1 Hilbert R-Tree

The main idea behind this variant is that ordering of rectangles plays important role in efficiency. Space filling curves were used for this purpose and Hilbert curve was found to give the best results. The centers of the input data rectangles are considered as the Hilbert values. With these values the tree is formed. Each node has a field to store the maximum Hilbert value of its children. Searching procedure is similar to that of R-Tree. Insertion is done using the Hilbert value of the new rectangle as a key [7]. In this way after reaching the leaf the rectangle is inserted and the changes are propagated upto the root. The idea of deferred splitting is used. When a node overflows, the rectangles are inserted in the siblings. When the siblings too are full, then splitting takes place. Rather than traditional 1-2 splitting, 2-3, 3-4 and 4-5

splitting were tried and 2-3 splitting gave a trade-off between insertion speed and search speed. The Hilbert R-Tree supported around 28% saving in response time compared with the other variants.

3.2.2 R k-d Tree

This variant combines and takes advantage of both the space partitioning (SP) (kd tree) and data partitioning (DP) technique (R-Tree) and gives a fast intra node search. Fan-out is independent of dimensionality. The indexed space need not be mutually disjoint and split with overlap is allowed only when overlap free splits are found to cause costly cascading splits. The whole space is first space partitioned into different spaces. The space partition within each index node is represented by kd tree. Since kd trees can represent only non overlapping splits, kd tree structure is modified [28] to include the right side position of the left partition (*rsp*) and left side boundary of the right partition (*lsp*). When *lsp* and *rsp* are equal, then it shows no overlap, if they are different then it shows overlap. Now the k-d based representation is mapped to BR (Bounding Rectangle) representation (which supports overlap) so that the search, insert and delete algorithms of DP based techniques can be adopted.

3.2.3 HR Tree

HR Tree is based on the R-Tree and hash address sorting. Centre position of the data object is used to judge and adopt the hash table which links the parent and the child nodes together. Insertion, deletion and queries can quickly find the targeted node. CPU execution time is reduced. The hash address sorting is done by first calculating the centre of the outer most MBR, then sorting the values based on x co-ordinates and putting them in first level one dimensional hash table. Then calculate the inner layer MBR's centre which lie within the outer layer MBR. Sort based on x co-ordinates and put them in secondary one-dimensional hash table and make the pointers in first level to point them. The data objects are stored in last hash table. The HR Tree [29] is constructed by making the first level as root, the ones in the secondary level as the next level and the last level as leaf nodes.

*3.2.4 R*Q Tree*

R*Q Tree [20] is a hybrid of both the R* variant and Quad tree. Variations and improvements to this tree also have been proposed. In conventional R*Q Tree, the depth of 2 in quad-tree will divide the index space into four index spaces, and each node of this quad-tree is connected with a sub-index-space and a R-Tree. In addition, R*Q tree can also be seen as a set of a group of R-Trees. However, there are two special cases: (1) when the depth of the quad-tree is 1, the R*Q tree is R*-Tree; (2) when the depth of the R*-Tree is 1 or 0, the R*Q-tree is null. An improvement [31][35] to this structure to avoid overlap is to maintain two pointers HP and VP (horizontal and vertical pointers) [35] to maintain list of objects which cannot be included fully in any area and intersect horizontal and vertical split line respectively. The improved version enhances query performance significantly. Lazy splitting and clustering is proposed [34] to enhance query performance and reduce the index cost. This version defers split by inserting entries to the neighbor nodes when the target node is full. After the neighbors also being filled, clustering is done to reorganize the objects.

3.2.5 Q+R Tree

This tree [21] combines Quad Tree and R-Tree and efficiently handless the moving objects. Quasi static (E.g. People moving within a building) and fast moving objects are separated by topography. Quasi static objects, though moving have a smaller velocity and thus the chance of getting out of present MBR is very less. The lazy update approach [18] is applied here. Thus R-Tree is constructed for the quasi static objects. Fast moving objects are indexed by quad tree even though fast moving, chances of moving out from the present quad tree is less. Since the R-Tree and Quad Tree space overlap, pointers from the leaves of quad tree point to the MBRs of the R-Tree. The pointers can point to any level of the R-Tree. Updates in this variant can be any combinations like within the same MBR in R-Tree, between MBRs in R-Tree, from R-Tree to Quad Tree, within same quadrant in Quad Tree, between different quadrants in R-Tree, from quad Tree to R-Tree. Most of the cases fall under the first type and thus overhead will be very less.

3.2.6 Vo R-Tree

Vo R-Tree [32] is a hybrid of R-Tree and Voronoi diagrams and processes nearest neighbor queries very efficiently. Nearest neighbor query is answered in two phases in traditional R-Tree. First the R-Tree is searched hierarchically and the neighborhood of the result set is arrived. Second the R-Tree nodes intersecting with the search region of an initial answer are explored to find all the members of the result set. R-Trees are found to be efficient in the first step. But during the second step R-Tree results in searching more unnecessary nodes which do not contribute for the result set. Voronoi diagrams are efficient in exploring the nearest neighbor search region but arriving to the search region is slow in this method. Thus Vo R-Tree, the hybrid variety combines the advantage of both the methods and is very efficient in solving nearest neighbor queries. The Vo R-Tree is constructed as follows: R-Tree is constructed for the set of data points. Voronoi diagram is also constructed for the same set of data points. The leaf nodes of the R-Tree contain extra information regarding the location of voronoi neighbors and the vertices of voronoi cell of the data points.

3.3 Variations due to Extension of the Structure

The R-Tree structure is extended to store additional information so that queries can be easily solved or additional applications can be supported. The trees include DR Tree [30], RT Tree [5], 3D R-Tree [10], Historical R-Tree [16], Partially Persistent R-Tree [17], Parametric R-Tree [14] and FNR R-Tree [23].

3.3.1 DR Tree

The DR Tree [30] is based on direction relationship. The structure is slightly modified and is explained below. The centroid of the MBRs is calculated and the direction is calculated based on their positions. The root node is the outermost MBR of the query region centroid. The root node and the non leaf nodes have five children nodes *I, N, S, W* and *E* which store the internal nodes, north, south, west, east represent the nearest MBRs in the respective directions. During a k-nearest neighbor (k-NN) query, the node of the MBR containing the queried data object is obtained from the leaf node, if there exists data objects in its internal node; the distance is calculated, sorted and then put in the result set. The distance from the sibling nodes of the data object are calculated, sorted and added to the result queue. If the k-NN query is satisfied then the process is over, else the MBRs of all the queried regions are queried in the four directions of the MBR nodes until the condition is met. The DR Tree gives excellent results for k-NN queries by reducing the CPU time for retrieval.

3.3.2 RT Tree

RT-Tree [5] is an extension of traditional R-Tree storing temporal information of the objects. Indexing is done based on spatial information. The temporal extent is added when the object is inserted. As long as the object does not change its position, the temporal extent is increased at every time instant. When the object changes its position at some other time instant, a new entry is inserted with the temporal extent initialized to this time instant. When the updates are frequent, more entries are added and more space is occupied. Since the tree is based on spatial information, temporal queries need considerable computation effort.

3.3.3 3D R-Tree

In 3D R-Tree [10], the MBRs are extended in the third temporal dimensional. This method suits only when the start and end time of the objects are known in prior. When the end time is not known, it can be a predicted time in future but this may cause pror performance. However this tree was found to be better in answering spatio-temporal queries than a variant in which 2D Tree was to maintain spatial information and 1D Tree was temporal information. In order to overcome the open boundary problem in 3D R-Tree, 2+3 R-Tree [13] was proposed. In this approach, two trees, a 2D R-Tree and a 3D R-Tree are maintained. The 2D R-Tree stores the spatial information and the 3D R-Tree stores the spatio-temporal information of the objects. The process is as follows: As long as the end time of the temporal extent of a particular object is not known, it is kept in the 2D R-Tree with its spatial and start-time information. Once the end-time of the object is known, then the object is moved from the 2D R-Tree to the 3D R-Tree. Thus, information about the present state of the object can be obtained from the 2D R-Tree and the past information can be obtained from the 3D R-tree.

3.3.4 Historical R-Tree

Historical R-Trees [16] make use of the concept of overlapping trees. Whenever an update occurs, a new R-Tree is formed. Since updates are localized, most part of the R-Tree would be the same for both the versions. Maintaining separate physical R-Tree structures is not space efficient. Hence, R-Trees are maintained logically. Historiacal R-Trees can be viewed as acyclic graphs rather than separate structures. Queries can be answered by maintaining an array which points to the root of the underlying R-Trees. Once the desired R-Tree is found, then the process and performance is same as if the R-Tree is maintained physically. When the number of moving objects is large and their updates are frequent, then no common part among separate structures exists, so performance degrades. This variant outperforms 3D and 2+3 R-Trees.

3.3.5 Partially Persistent R-Tree

Partially Persisten R-Tree (PPR Tree) is based on partial persistency [12]. As updates arrive only in the order of time, it can be performed on the last recorded instance of the object. PPR Tree [17] is an acyclic graph with each node serving as a root to the subsequent ephermal R-tree. Each data record has an insertion and deletion time. Internal nodes too maintain the evolution of their directories. A leaf or internal node entry is alive at any time instance within its life time interval. Leaf node or internal node is said to be alive until it is split. To improve I/O efficiency, there exists bound on number of alive entries that can be present in a node. When an update is to be performed at time instance *t*, the leaf where the update must be performed is searched by traversing only the alive entries. When the leaf is found for insertion operation, the object is inserted with the time interval (t, now) and for deletion operation, the time interval of the previous version entry *now* is substituted by *t.*

3.3.6 Parametric R-Tree

Parametric R-Tree [17] is similar to R-Tree except that the MBR is parameterized. Both the spatial extent and the temporal extent of the object is considered. The Minimum Bounding Parametric Rectangle *r* can be defined as the parametric rectangle of *S* if it contains all the parametric rectangles of *S*. projection of *r* on space at any time instant is minimized. The parameters are x and y co-ordinate of the lower and upper boundaries as a function of *t* and the time bound for *t.* Insertion, Deletion, Splitting is similar to R-Tree but with small modifications which is described in [17]. Parametric R-Tree is found to have efficient I/O. Time Parameterised R-Tree (TPR-Tree) [15] includes the velocity bound so that the future positions of the object can be calculated from its present position and velocity. TPR* Tree [22] is an improvement of TPR Tree and performs better in answering predictive queries. VCI R-Tree [19], another variant is a velocity constrained variant in which the maximum allowed speed for the object is stored along with it. During queries, due to false positives, a post processing step is necessary. An enhancement to this variant is to periodically re organize the tree to reduce the false positives.

3.3.7 FNR Tree

Spatio-Temporal R-Trees consider free movement of objects. But in most applications, constraints exist in the underlying spatial network. Though constraint movement is a special case of free movement, severl optimizations can be done in constraint movement. FNR Tree [23] is a forest of R-Trees consisting of a 2D R-Tree which stores the spatial network and the 2D R-Tree organises set of 1D R-Trees which stores the objects. The 2D R-Tree does not change and only the 1D R-Tree only undergoes update. Each record in 1D R-Tree consists of ID line which identifies the segment of the spatial network, MBR and orientation. MON Tree [25] a variant of FNR Tree, uses 2D R-Tree instead of the 1D R-Tree for every leaf node of the 2D R-Tree which stores the spatial network.

4. CONCLUSION

R-Tree is the most widely used spatial index structure serving many applications which demand multi-dimensional data. Many variants of R-Tree have been proposed so that some aspects of indexing may be improved or done efficiently. This paper presents a complete survey of all the variants of R-Tree. The variants are classified based on how they vary from the original. They are classified based on change in process, hybrid varity or extension to the existing to support additional applications. It is seen that R-Tree can also act as a spatio-temporal index.

5. REFERENCES

[1] Raphael A. Finkel and Jon Louis Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", Journal of Acts Informtica, vol.4, no.1, pp.1-9, 1974.

[2] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984* ACM SIGMOD *International Conference on Management of Data*, pp.47-57, 1984.

[3] Timos K. Sellis, Nick Roussopoulos and Christos Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", *Proceedings of 13th International Conference on Very Large Data Bases,* pp.507-518, 1987.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp.322-331, 1990.

[5] X. Xu, J. Han and W. Lu, "RT-Tree: An Improved R-tree Index Structure for Spatiotemporal Databases", *Proceedings of the 4th International Symposium on Spatial Data Handling*, pp.1040-1049, 1990.

[6] I. Kamel and C. Faloutsos, "On Packing R-Trees", *Proceedings of the 2nd International Conference on Information and Knowledge Management*, pp. 490-499, 1993.

[7] I. Kamel and C. Faloutsos, "Hilbert R-Tree - An Improved R-tree using Fractals", *Proceedings of the 20th International Conference on Very Large Data Bases*, pp.500-509, 1994.

[8] L. Arge, "The Buffer Tree: A New Technique for Optimal I/O Algorithms", *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pp.334-345, 1995.

[9] Stefan Berchtold, Daniel A. Keim and Hans-Peter Kriegei, "The X-tree: An Index Structure for High-Dimensional Data", *Proceedings of the 22nd International Conference on Very Large Data Bases*, pp.28-39, 1996.

[10] Y. Theodoridis, M. Vazirgiannis and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications", *Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems*, pp.441-448, 1996.

[11] S. Leutenegger, J.M. Edgington and M.A. Lopez, "STR - A Simple and Efficient Algorithm for R-Tree Packing", *Proceedings of the 13th IEEE International Conference on Data Engineering*, pp.497-506, 1997.

[12] A. Kumar, V.J. Tsotras and C. Faloutsos, "Designing Access Methods for Bitemporal Databases", IEEE Transactions on Knowledge and Data Engineering, vol.10, no.1, pp.1-20, 1998.

[13] M.A. Nascimento, J.R.O. Silva and Y. Theodoridis, "Evaluation of Access Structures for Discretely Moving Points", *Proceedings of the 1st International Symposium on Spatiotemporal Database Management*, pp.171-188, 1999.

[14] Mengchu Cai and Peter Revesz, "Parametric R-Tree: An Index Structure for Moving Objects", *Proceedings of the 10th International Conference on Management of Data*, 2000.

[15] S. Saltenis, C.S. Jensen, S. Leutenegger and M. Lopez, "Indexing the Positions of Continuously Moving Objects", *Proceedings of ACM SIGMOD Conference on Management of Data*, pp.331-342, 2000.

[16] Y. Tao and D. Papadias, "Efficient Historical R-Trees", *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pp.223-232, 2001.

[17] G. Kollio et.al., "Indexing Animated Objects Using Spatiotemporal Access Methods", IEEE Transactions on Knowledge and Data Engineering, vol.13, no.5, pp.758-777, 2001.

[18] Dongseop Kwon, Sangjun Lee and Sukho Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree", *Proceedings of the 3rd International Conference on Mobile Data Management*, pp.113–120, 2002.

[19] S. Prabhakar et.al., "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects", IEEE Transactions on Computers, vol.51, no.10, pp.1124-1140, 2002.

[20] Jianhua Qiu, Xuebing Tang and Huaguo Huang, "An Index Structure based on Quad-Tree and R*-Tree—R*Q-Tree", Journal of Computer Applications, vol.23, no.8, pp.124-126, 2003.

[21] Y. Xia and S. Prabhakar, "Q+R-Tree: Efficient Indexing for Moving Object Databases", *Proceedings of 8th International Conference on Database Systems for Advanced Applications*, pp.175-182, 2003.

[22] Yufei Tao, Dimitris Papadias and Jimeng Sum, "The TPR*-tree: An Optimized Spatio-Temporal Access Method for Predictive Queries", *Proceedings of the 29th International Conference on Very Large Data Bases*, pp.790-801, 2003.

[23] E. Frentzos, "Indexing Objects Moving on Fixed Networks", *Proceedings of the 8th International Symposium on Spatial and Temporal Databases*, pp.289-305,2003.

[24] L. Arge, M. deBerg, H.J. Haverkort and K. Yi, "The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree", *Proceedings of ACM SIGMOD Conference on Management of Data*, pp.347-358, 2004.

[25] V.T. Almeida and R.H. Guting, "Indexing the Trajectories of Moving Objects in Networks", *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pp.115-118, 2004.

[26] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos and Yannis Theodoridis, R-Trees: Theory and Applications, London: Springer, 1st Edition, 2006.

[27] TAN Ning and SHI Yue-xiang, "Optimization Research of Multi-Dimensional Indexing Structure of R*-Tree", *Proceedings of the International Conference on Information Technology and Applications*, pp.612-615, 2009

[28] P. Anandha Kumar et.al., "Location Based Hybrid Indexing Structure - R k-d Tree", P*roceedings of the 1st International Conference on Integrated Intelligent Computing*, pp.140-145, 2010.

[29] Guobin Li and Jine Tang, "A New HR-Tree Index based on Hash Address", *Proceedings of the 2nd International Conference on Signal Processing Systems*, pp.35-38, 2010.

[30] Guobin Li and Jine Tang, "A New DR-tree K-Nearest Neighbor Query Algorithm based on Direction Relationship", *Proceedings of the International*

Conference on Environmental Science and Information Application Technology, pp.246-250, 2010.

[31] Guobin Li and Jine Tang, "A New Method about Spatial Data Query based on the R*Q-Tree", *Proceedings of the 6th International Conference on Neural Computing*, pp.1044-1047, 2010.

[32] Mehdi Sharifzadeh and Cyrus Shahabi, "VoR-Tree: R-Trees with Voronoi Diagrams for Efficient Processing of Spatial Nearest Neighbor Queries", *Journal Proceedings of VLDB Environment*, vol.3, no.1, pp.1231-1243, 2010.

[33] Amer Al-Badarneh and Abdullah Al-Alaj, "A Spatial Index Structure using Dynamic Recursive Space Partitioning", *Proceedings of the International Conference on Innovations in Information Technology*, pp.255-260, 2011.

[34] Pan Jin and Quanyou Song, "A Novel Index Structure R*Q-Tree based on Lazy Splitting and Clustering", *Proceedings of the International Conference on Computer Science and Automation Engineering*, pp.405-407, 2011.

[35] Quanyou Song, "An Improved Spatial Index based on the R*Q-Tree", *Proceedings of the International Conference on Business Management and Electronic Information*, pp.786-789, 2011.