WIKIPEDIA
The Free Encyclopedia

WIKIPEDIA

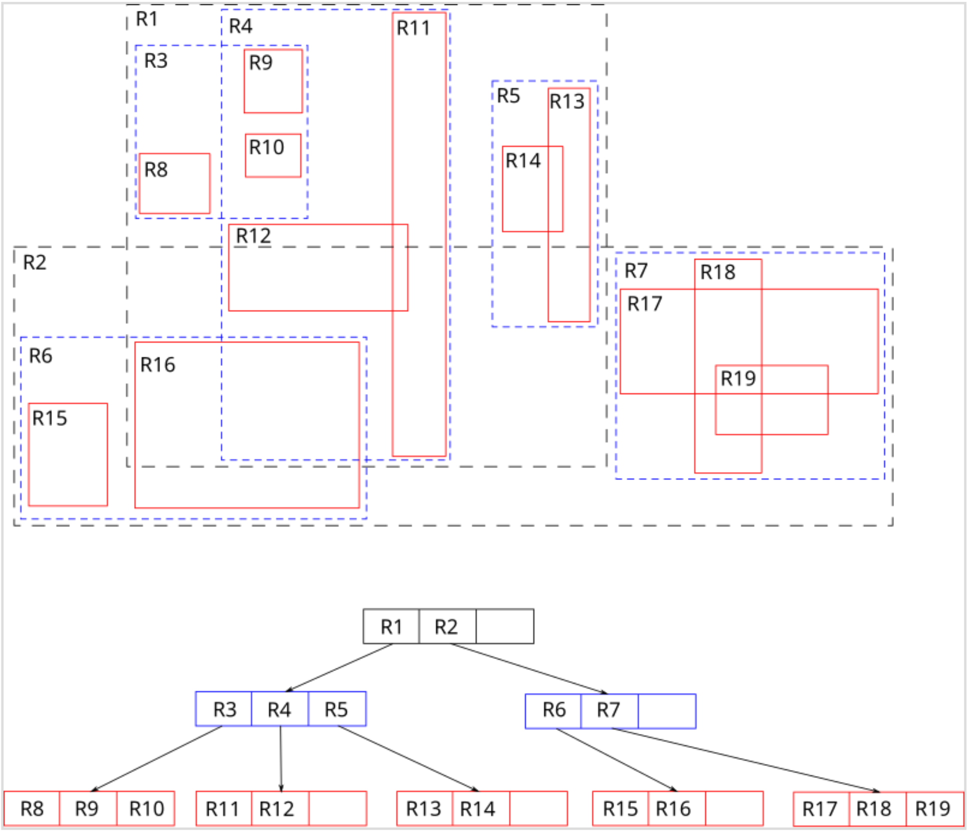
R-tree

R-tree		
Type	tree	
Invented	1984	
Invented by	Antonin Guttman	
Time complexity in big O notation		
Operation	**Average**	**Worst case**
Search	O($log_M n$)	O(n)[1]
Insert		O(n)
Space complexity		

R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. The R-tree was proposed by Antonin Guttman in 1984[2] and has found significant use in both theoretical and applied contexts.[3] A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest gas station" (although not taking roads into account). The R-tree can also accelerate nearest neighbor search[4] for various distance metrics, including great-circle distance.[5]

R-tree idea

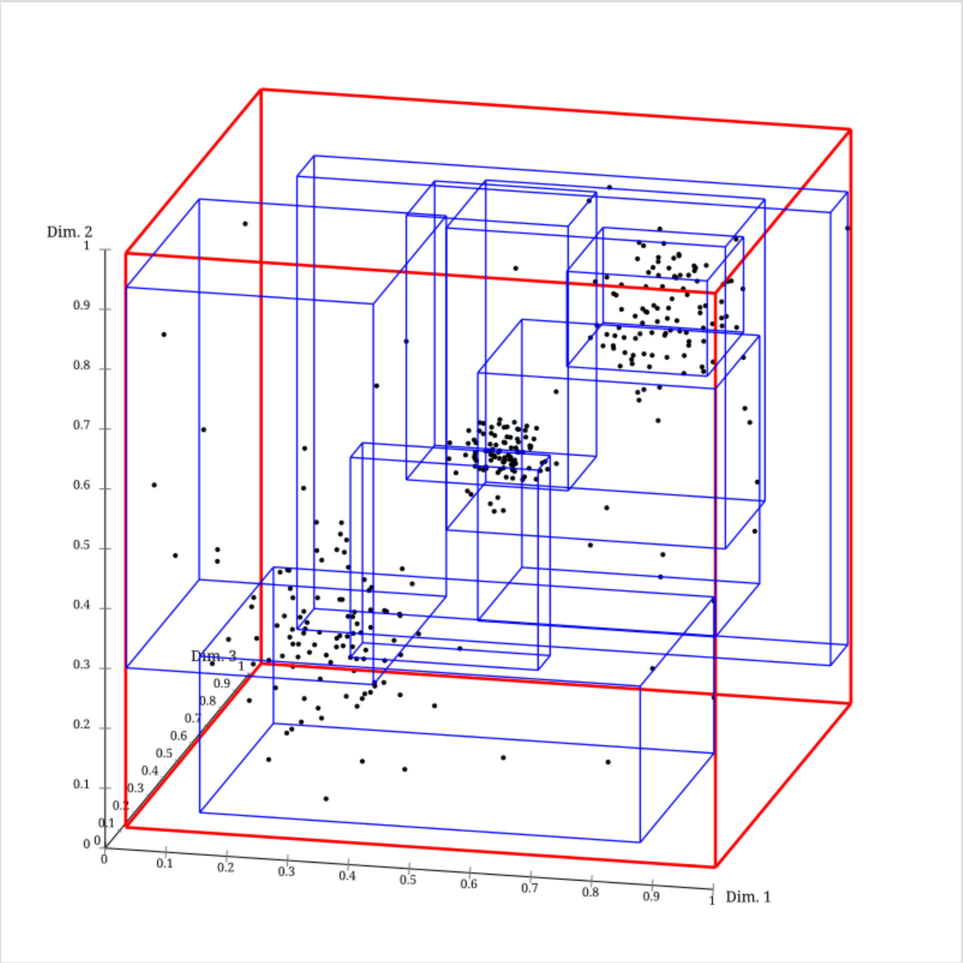
The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree; the "R" in R-tree is for rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation includes an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.



Simple example of an R-tree for 2D rectangles

Similar to the B-tree, the R-tree is also a balanced search tree (so all leaf nodes are at the same depth), organizes the data in pages, and is designed for storage on disk (as used in databases). Each page can contain a maximum number of entries, often denoted as M. It also guarantees a minimum fill (except for the root node), however best performance has been experienced with a minimum fill of 30%–40% of the maximum number of entries (B-trees guarantee 50% page fill, and B*-trees even 66%). The reason for this is the more complex balancing required for spatial data as opposed to linear data stored in B-trees.

As with most trees, the searching algorithms (e.g., intersection, containment, nearest neighbor search) are rather simple. The key idea is to use the bounding boxes to decide whether or not to search inside a subtree. In this way, most of the nodes in the tree are never read during a search. Like B-trees, R-trees are suitable for large data sets and databases, where nodes can be paged to memory when needed, and the whole tree cannot be kept in main memory. Even if data can be fit in memory (or cached), the R-trees in most practical applications will usually provide performance advantages over naive check of all objects when the number of objects is more than few hundred or so. However, for in-memory applications, there are similar alternatives that can provide slightly better performance or be simpler to implement in practice. To maintain in-memory computing for R-tree in a computer cluster where computing nodes are connected by a network, researchers have used RDMA (Remote Direct Memory Access) to implement data-intensive applications under R-tree in a distributed environment.[6] This approach is scalable for increasingly large applications and achieves high throughput and low latency performance for R-tree.



Visualization of an R*-tree for 3D points using ELKI (the cubes are directory pages)

The key difficulty of R-tree is to build an efficient tree that on one hand is balanced (so the leaf nodes are at the same height) on the other hand the rectangles do not cover too much empty space and do not overlap too much (so that during search, fewer subtrees need to be processed). For example, the original idea for inserting elements to obtain an efficient tree is to always insert into the subtree that requires least enlargement of its bounding box. Once that page is full, the data is split into two sets that should cover the minimal area each. Most of the research and improvements for R-trees aims at improving the way the tree is built and can be grouped into two objectives: building an efficient tree from scratch (known as bulk-loading) and performing changes on an existing tree (insertion and deletion).

R-trees do not guarantee good worst-case performance, but generally perform well with real-world data.[7] While more of theoretical interest, the (bulk-loaded) Priority R-tree variant of the R-tree is worst-case optimal,[8] but due to the increased complexity, has not received much attention in practical applications so far.

When data is organized in an R-tree, the neighbors within a given distance r and the k nearest neighbors (for any L^p-Norm) of all points can efficiently be computed using a spatial join.[9][10] This is beneficial for many algorithms based on such queries, for example the Local Outlier Factor. DeLi-Clu,[11] Density-Link-Clustering is a cluster analysis algorithm that uses the R-tree structure for a similar kind of spatial join to efficiently compute an OPTICS clustering.

Variants

- Priority R-tree
- R*-tree
- R+ tree
- Hilbert R-tree
- X-tree

Algorithm

Data layout

Data in R-trees is organized in pages that can have a variable number of entries (up to some pre-defined maximum, and usually above a minimum fill). Each entry within a non-leaf node stores two pieces of data: a way of identifying a child node, and the bounding box of all entries within this child node. Leaf nodes store the data required for each child, often a point or bounding box representing the child and an external identifier for the child. For point data, the leaf entries can be just the points themselves. For polygon data (that often requires the storage of large polygons) the common setup is to store only the MBR (minimum bounding rectangle) of the polygon along with a unique identifier in the tree.

Search

In range searching, the input is a search rectangle (Query box). Searching is quite similar to searching in a B+ tree. The search starts from the root node of the tree. Every internal node contains a set of rectangles and pointers to the corresponding child node and every leaf node contains the rectangles of spatial objects (the pointer to some spatial object can be there). For every rectangle in a node, it has to be decided if it overlaps the search rectangle or not. If yes, the corresponding child node has to be searched also. Searching is done like this in a recursive manner until all overlapping nodes have been traversed. When a leaf node is reached, the contained bounding boxes (rectangles) are tested against the search rectangle and their objects (if there are any) are put into the result set if they lie within the search rectangle.

For priority search such as nearest neighbor search, the query consists of a point or rectangle. The root node is inserted into the priority queue. Until the queue is empty or the desired number of results have been returned the search continues by processing the nearest entry in the queue. Tree nodes are expanded and their children reinserted. Leaf entries are returned when encountered in the queue.[12] This approach can be used with various distance metrics, including great-circle distance for geographic data.[5]

Insertion

To insert an object, the tree is traversed recursively from the root node. At each step, all rectangles in the current directory node are examined, and a candidate is chosen using a heuristic such as choosing the rectangle which requires least enlargement. The search then descends into this page, until reaching a leaf node. If the leaf node is full, it must be split before the insertion is made. Again, since an exhaustive search is too expensive, a heuristic is employed to split the node into two. Adding the newly created node to the previous level, this level can again overflow, and these overflows can propagate up to the root node; when this node also overflows, a new root node is created and the tree has increased in height.

Choosing the insertion subtree

The algorithm needs to decide in which subtree to insert. When a data object is fully contained in a single rectangle, the choice is clear. When there are multiple options or rectangles in need of enlargement, the choice can have a significant impact on the performance of the tree.

The objects are inserted into the subtree that needs the least enlargement. A Mixture heuristic is employed throughout. What happens next is it tries to minimize the overlap (in case of ties, prefer least enlargement and then least area); at the higher levels, it behaves similar to the R-tree, but on ties again preferring the subtree with smaller area. The decreased overlap of rectangles in the R*-tree is one of the key benefits over the traditional R-tree.

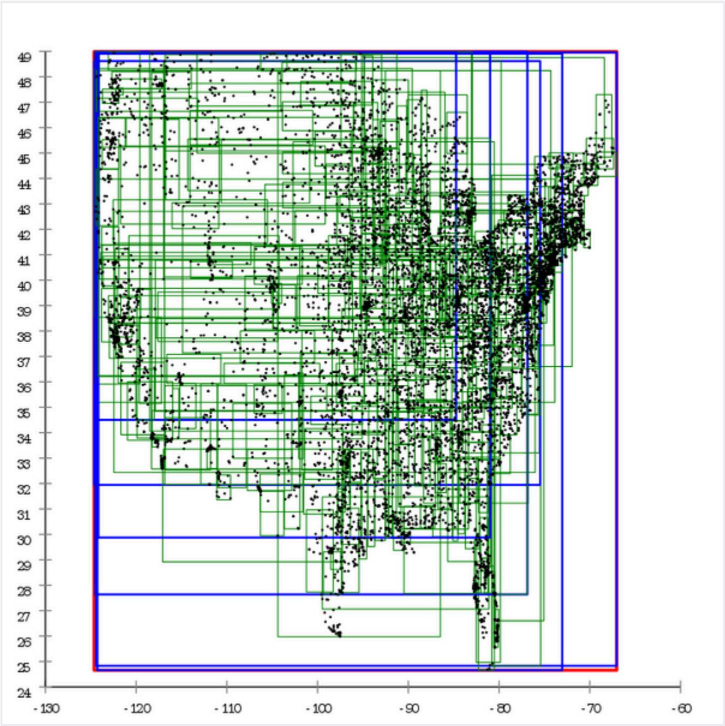
Splitting an overflowing node

Since redistributing all objects of a node into two nodes has an exponential number of options, a heuristic needs to be employed to find the best split. In the classic R-tree, Guttman proposed two such heuristics, called QuadraticSplit and LinearSplit. In quadratic split, the algorithm searches for the pair of rectangles that is the worst combination to have in the same node, and puts them as initial objects into the two new groups. It then searches for the entry which has the strongest preference for one of the groups (in terms of area increase) and assigns the object to this group until all objects are assigned (satisfying the minimum fill).

There are other splitting strategies such as Greene's Split,[13] the R*-tree splitting heuristic[14] (which again tries to minimize overlap, but also prefers quadratic pages) or the linear split algorithm proposed by Ang and Tan[15] (which however can produce very irregular rectangles, which are less performant for many real world range and window queries). In addition to having a more advanced splitting heuristic, the R*-tree also tries to avoid splitting a node by reinserting some of the node members, which is similar to the way a B-tree balances overflowing nodes. This was shown to also reduce overlap and thus increase tree performance.

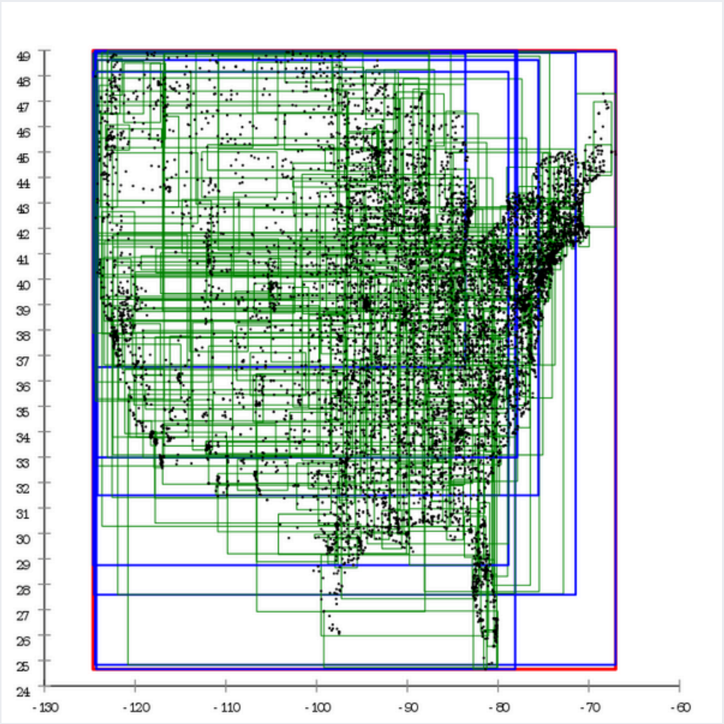
Finally, the X-tree[16] can be seen as a R*-tree variant that can also decide to not split a node, but construct a so-called super-node containing all the extra entries, when it doesn't find a good split (in particular for high-dimensional data).

Effect of different splitting heuristics on a database with US postal districts



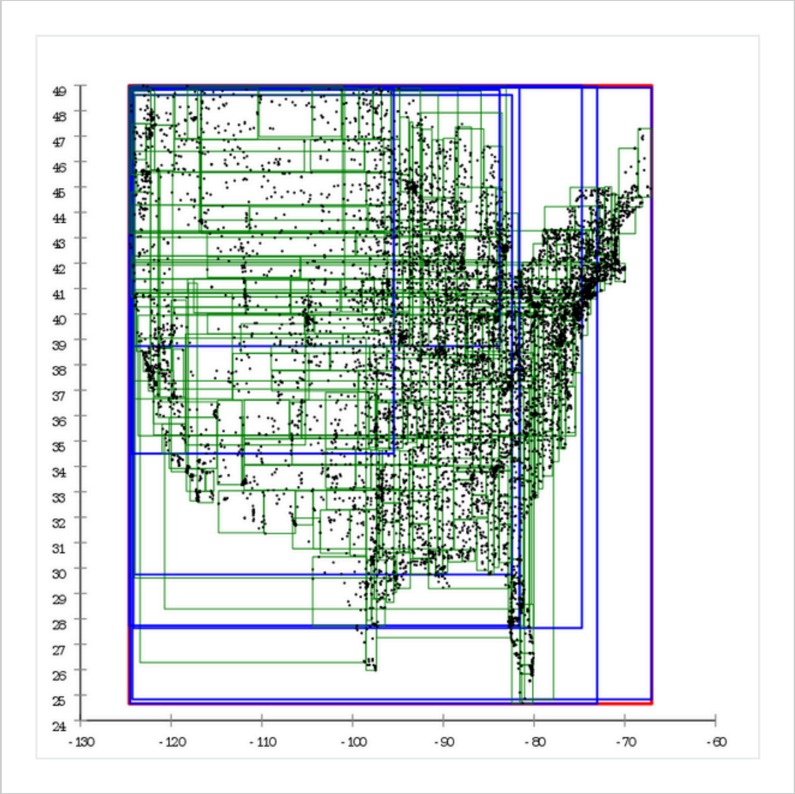
Guttman's quadratic split.[2]

Pages in this tree overlap a lot.

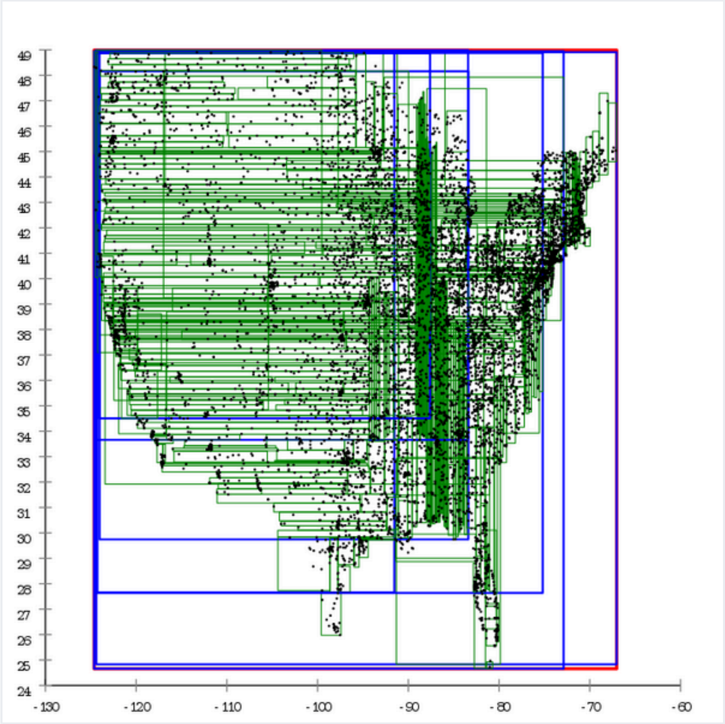


Guttman's linear split.[2]

Even worse structure, but also faster to construct.

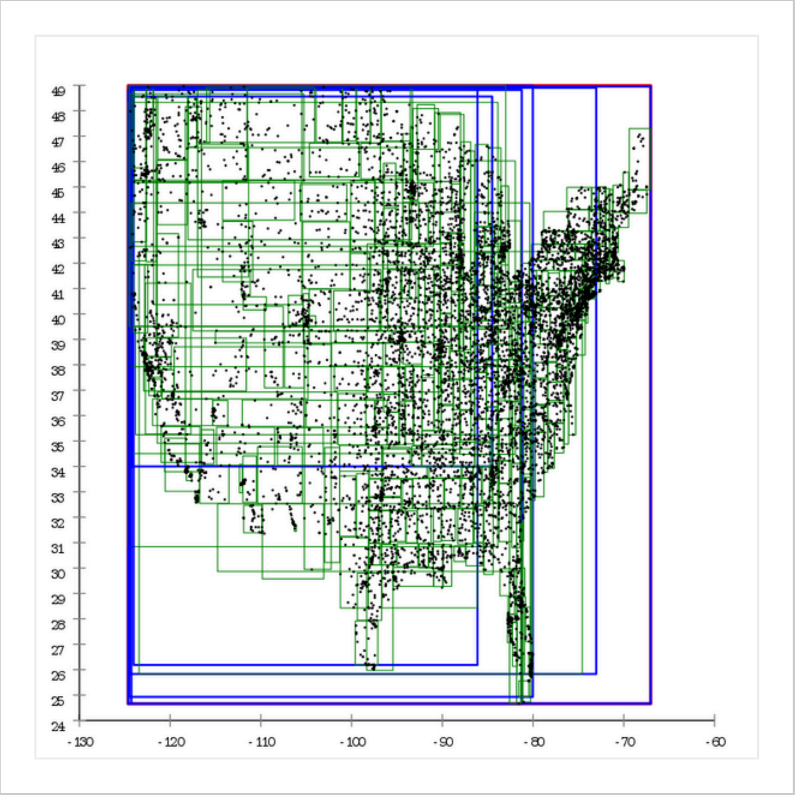


Greene's split.[13] Pages overlap much less than with Guttman's strategy.



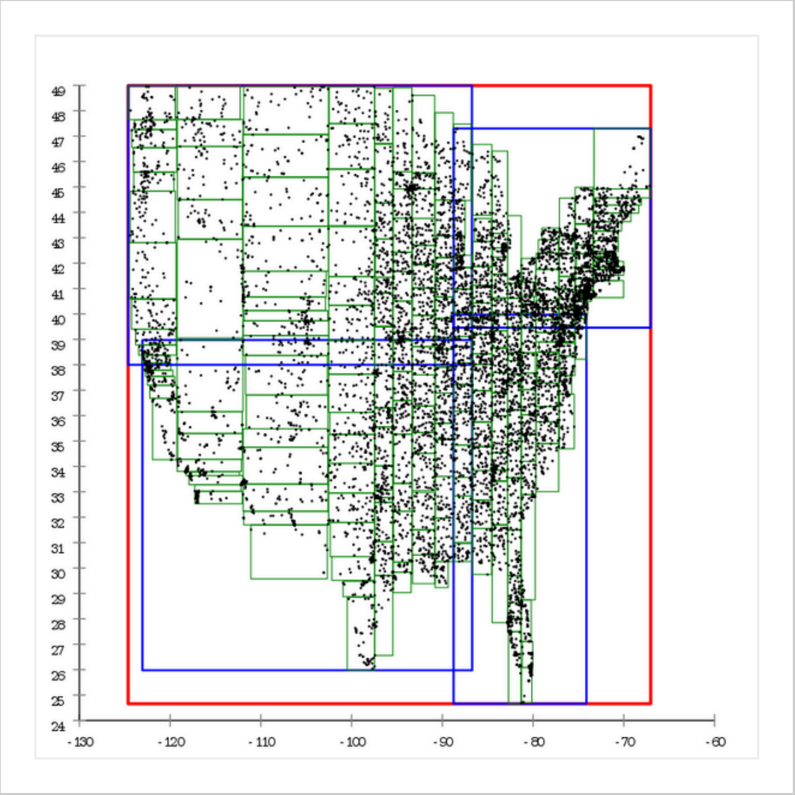
Ang-Tan linear split.[15]

This strategy produces sliced pages, which often yield bad query performance.

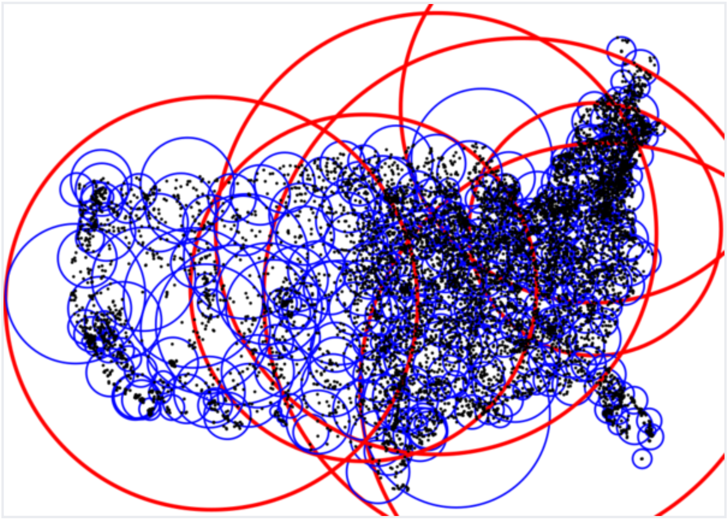


R* tree topological split.[14]

The pages overlap much less since the R*-tree tries to minimize page overlap, and the reinsertions further optimized the tree. The split strategy prefers quadratic pages, which yields better performance for common map applications.



Bulk loaded R* tree using Sort-Tile-Recursive (STR).
The leaf pages do not overlap at all, and the directory pages overlap only little. This is a very efficient tree, but it requires the data to be completely known beforehand.



M-trees are similar to the R-tree, but use nested spherical pages.
Splitting these pages is, however, much more complicated and pages usually overlap much more.

Deletion

Deleting an entry from a page may require updating the bounding rectangles of parent pages. However, when a page is underfull, it will not be balanced with its neighbors. Instead, the page will be dissolved and all its children (which may be subtrees, not only leaf objects) will be reinserted. If during this process the root node has a single element, the tree height can decrease.

Bulk-loading

- Nearest-X: Objects are sorted by their first coordinate ("X") and then split into pages of the desired size.
- Packed Hilbert R-tree: variation of Nearest-X, but sorting using the Hilbert value of the center of a rectangle instead of using the X coordinate. There is no guarantee the pages will not overlap.

- Sort-Tile-Recursive (STR):[17] Another variation of Nearest-X, that estimates the total number of leaves required as $l = \lceil \mathbf{number\ of\ objects}/\mathbf{capacity} \rceil$, the required split factor in each dimension to achieve this as $s = \lceil l^{1/d} \rceil$, then repeatedly splits each dimensions successively into s equal sized partitions using 1-dimensional sorting. The resulting pages, if they occupy more than one page, are again bulk-loaded using the same algorithm. For point data, the leaf nodes will not overlap, and "tile" the data space into approximately equal sized pages.
- Overlap Minimizing Top-down (OMT):[18] Improvement over STR using a top-down approach which minimizes overlaps between slices and improves query performance.
- Priority R-tree

See also

- Segment tree
- Interval tree – A degenerate R-tree for one dimension (usually time).
- K-d tree
- Bounding volume hierarchy
- Spatial index
- GiST

References

1. R Tree (https://www2.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf) cs.sfu.ca
2. Guttman, A. (1984). "R-Trees: A Dynamic Index Structure for Spatial Searching" (http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf) (PDF). *Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84*. p. 47. doi:10.1145/602259.602266 (https://doi.org/10.1145%2F602259.602266). ISBN 978-0897911283. S2CID 876601 (https://api.semanticscholar.org/CorpusID:876601).
3. Y. Manolopoulos; A. Nanopoulos; Y. Theodoridis (2006). *R-Trees: Theory and Applications* (https://books.google.com/books?id=1mu099DN9UwC&pg=PR5). Springer. ISBN 978-1-85233-977-7. Retrieved 8 October 2011.
4. Roussopoulos, N.; Kelley, S.; Vincent, F. D. R. (1995). "Nearest neighbor queries". *Proceedings of the 1995 ACM SIGMOD international conference on Management of data – SIGMOD '95*. p. 71. doi:10.1145/223784.223794 (https://doi.org/10.1145%2F223784.223794). ISBN 0897917316.

5. Schubert, E.; Zimek, A.; Kriegel, H. P. (2013). "Geodetic Distance Queries on R-Trees for Indexing Geographic Data". *Advances in Spatial and Temporal Databases*. Lecture Notes in Computer Science. Vol. 8098. p. 146. doi:10.1007/978-3-642-40235-7_9 (https://doi.org/10.1007%2F978-3-642-40235-7_9). ISBN 978-3-642-40234-0.
6. Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang (2022). *" An RDMA-enabled In-memory Computing Platform for R-tree on Clusters"*. ACM Transactions on Spatial Algorithms and Systems. pp. 1–26. doi:10.1145/3503513 (https://doi.org/10.1145%2F3503513).
7. Hwang, S.; Kwon, K.; Cha, S. K.; Lee, B. S. (2003). "Performance Evaluation of Main-Memory R-tree Variants" (https://archive.org/details/advancesinspatia0000sstd/page/10). *Advances in Spatial and Temporal Databases*. Lecture Notes in Computer Science. Vol. 2750. pp. 10 (https://archive.org/details/advancesinspatia0000sstd/page/10). doi:10.1007/978-3-540-45072-6_2 (https://doi.org/10.1007%2F978-3-540-45072-6_2). ISBN 978-3-540-40535-1.
8. Arge, L.; De Berg, M.; Haverkort, H. J.; Yi, K. (2004). "The Priority R-tree" (http://www.win.tue.nl/~mdberg/Papers/prtree.pdf) (PDF). *Proceedings of the 2004 ACM SIGMOD international conference on Management of data – SIGMOD '04* (http://doi.acm.org/10.1145/1007568.1007608). p. 347. doi:10.1145/1007568.1007608 (https://doi.org/10.1145%2F1007568.1007608). ISBN 978-1581138597. S2CID 6817500 (https://api.semanticscholar.org/CorpusID:6817500).
9. Brinkhoff, T.; Kriegel, H. P.; Seeger, B. (1993). "Efficient processing of spatial joins using R-trees". *ACM SIGMOD Record*. **22** (2): 237. CiteSeerX 10.1.1.72.4514 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.4514). doi:10.1145/170036.170075 (https://doi.org/10.1145%2F170036.170075). S2CID 7810650 (https://api.semanticscholar.org/CorpusID:7810650).
10. Böhm, Christian; Krebs, Florian (2003-09-01). "Supporting KDD Applications by the k-Nearest Neighbor Join". *Database and Expert Systems Applications*. Lecture Notes in Computer Science. Vol. 2736. Springer, Berlin, Heidelberg. pp. 504–516. CiteSeerX 10.1.1.71.454 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.454). doi:10.1007/978-3-540-45227-0_50 (https://doi.org/10.1007%2F978-3-540-45227-0_50). ISBN 9783540408062.
11. Achtert, Elke; Böhm, Christian; Kröger, Peer (2006). "DeLi-Clu: Boosting Robustness, Completeness, Usability, and Efficiency of Hierarchical Clustering by a Closest Pair Ranking". In Ng, Wee Keong; Kitsuregawa, Masaru; Li, Jianzhong; Chang, Kuiyu (eds.). *Advances in Knowledge Discovery and Data Mining, 10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006, Proceedings*. Lecture Notes in Computer Science. Vol. 3918. Springer. pp. 119–128. doi:10.1007/11731139_16 (https://doi.org/10.1007%2F11731139_16).

12. Kuan, J.; Lewis, P. (1997). "Fast k nearest neighbour search for R-tree family". *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat. No.97TH8237)*. p. 924. doi:10.1109/ICICS.1997.652114 (https://doi.org/10.1109%2FICICS.1997.652114). ISBN 0-7803-3676-3.
13. Greene, D. (1989). "An implementation and performance analysis of spatial data access methods". *[1989] Proceedings. Fifth International Conference on Data Engineering*. pp. 606–615. doi:10.1109/ICDE.1989.47268 (https://doi.org/10.1109%2FICDE.1989.47268). ISBN 978-0-8186-1915-1. S2CID 7957624 (https://api.semanticscholar.org/CorpusID:7957624).
14. Beckmann, N.; Kriegel, H. P.; Schneider, R.; Seeger, B. (1990). "The R*-tree: an efficient and robust access method for points and rectangles" (http://dbs.mathematik.uni-marburg.de/publications/myPapers/1990/BKSS90.pdf) (PDF). *Proceedings of the 1990 ACM SIGMOD international conference on Management of data – SIGMOD '90*. p. 322. CiteSeerX 10.1.1.129.3731 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.3731). doi:10.1145/93597.98741 (https://doi.org/10.1145%2F93597.98741). ISBN 978-0897913652. S2CID 11567855 (https://api.semanticscholar.org/CorpusID:11567855).
15. Ang, C. H.; Tan, T. C. (1997). "New linear node splitting algorithm for R-trees". In Scholl, Michel; Voisard, Agnès (eds.). *Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD '97), Berlin, Germany, July 15–18, 1997*. Lecture Notes in Computer Science. Vol. 1262. Springer. pp. 337–349. doi:10.1007/3-540-63238-7_38 (https://doi.org/10.1007%2F3-540-63238-7_38).
16. Berchtold, Stefan; Keim, Daniel A.; Kriegel, Hans-Peter (1996). "The X-Tree: An Index Structure for High-Dimensional Data" (http://www.dbs.ifi.lmu.de/Publikationen/Papers/x-tree.ps). *Proceedings of the 22nd VLDB Conference*. Mumbai, India: 28–39.
17. Leutenegger, Scott T.; Edgington, Jeffrey M.; Lopez, Mario A. (February 1997). "STR: A Simple and Efficient Algorithm for R-Tree Packing" (https://archive.org/details/nasa_techdoc_19970016975).
18. Lee, Taewon; Lee, Sukho (June 2003). "OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree" (http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-74/files/FORUM_18.pdf) (PDF).

External links

- Media related to R-tree at Wikimedia Commons