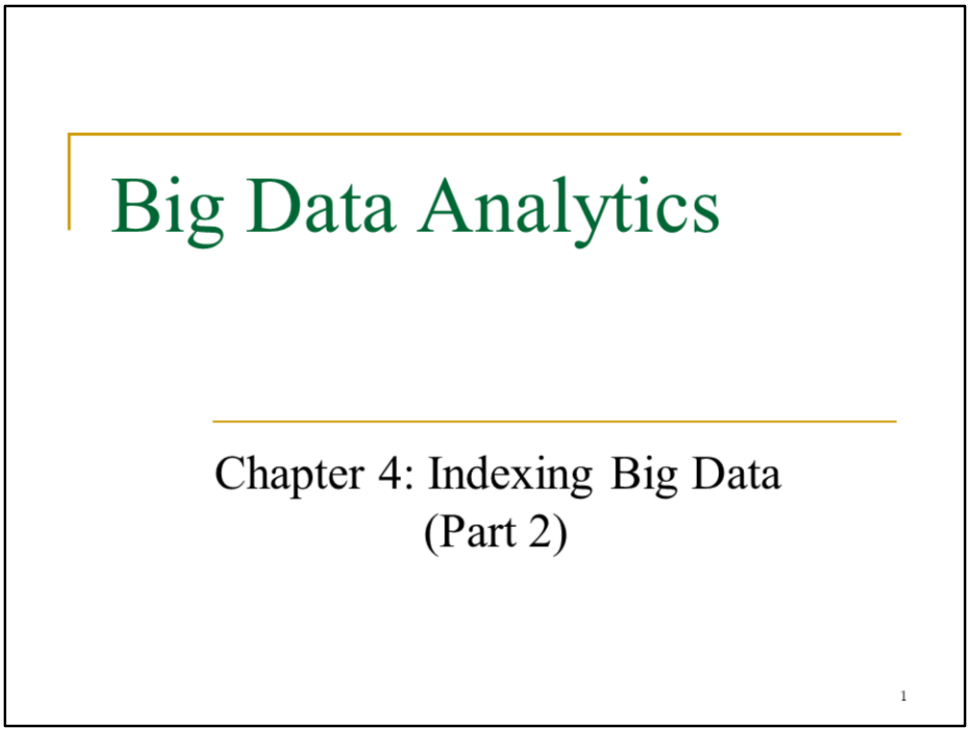


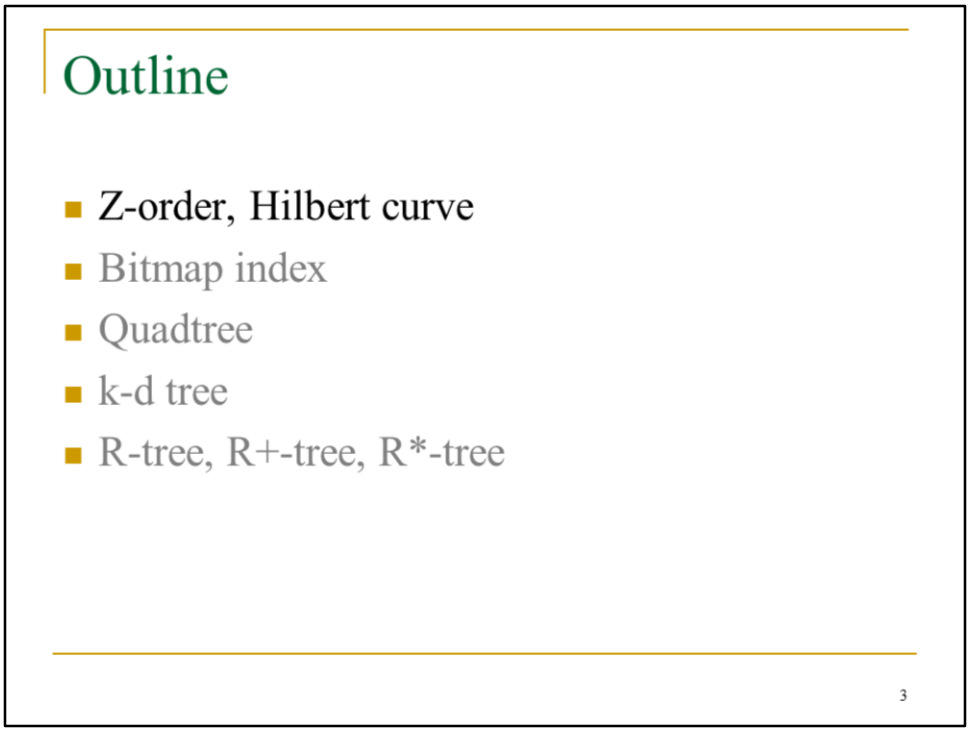
Chapter 4: Indexing Big Data (Part 2)

Objectives

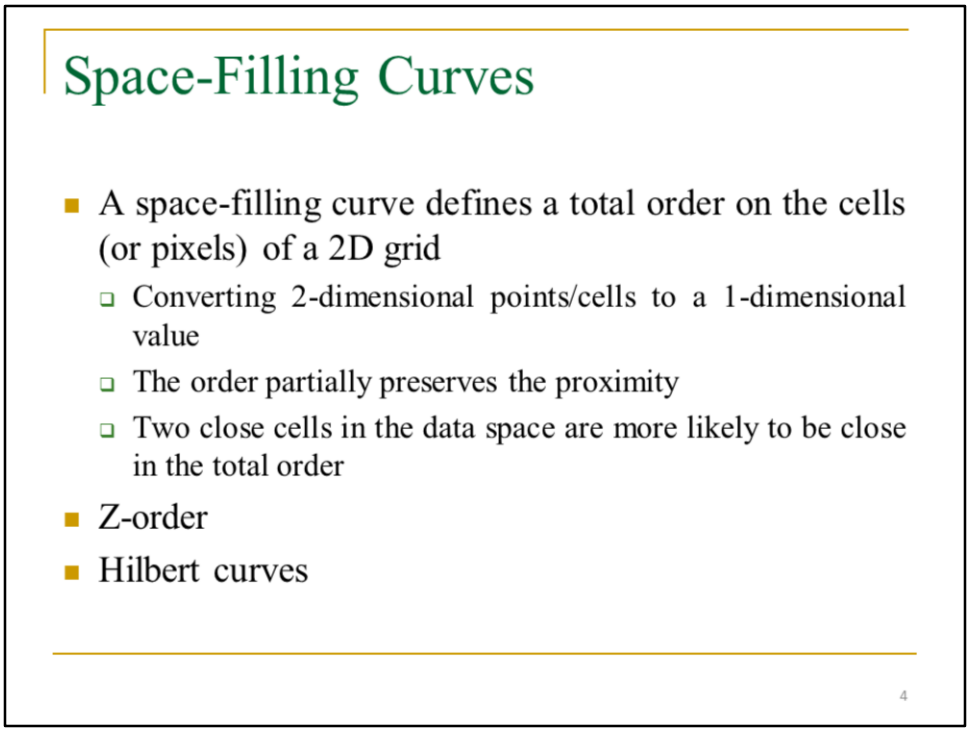
- In this chapter, you will:
  - Get familiar with multidimensional indexes:
    - Z-order, Hilbert curve
    - Bitmap index
    - Quadtree
    - k-d tree
    - R-tree, R+-tree, R*-tree

2

In this chapter, we start to discuss more multidimensional indexes over big data, including Z-order, Hilbert curve, Bitmap index, Quadtree, k-d tree, and R-tree family.



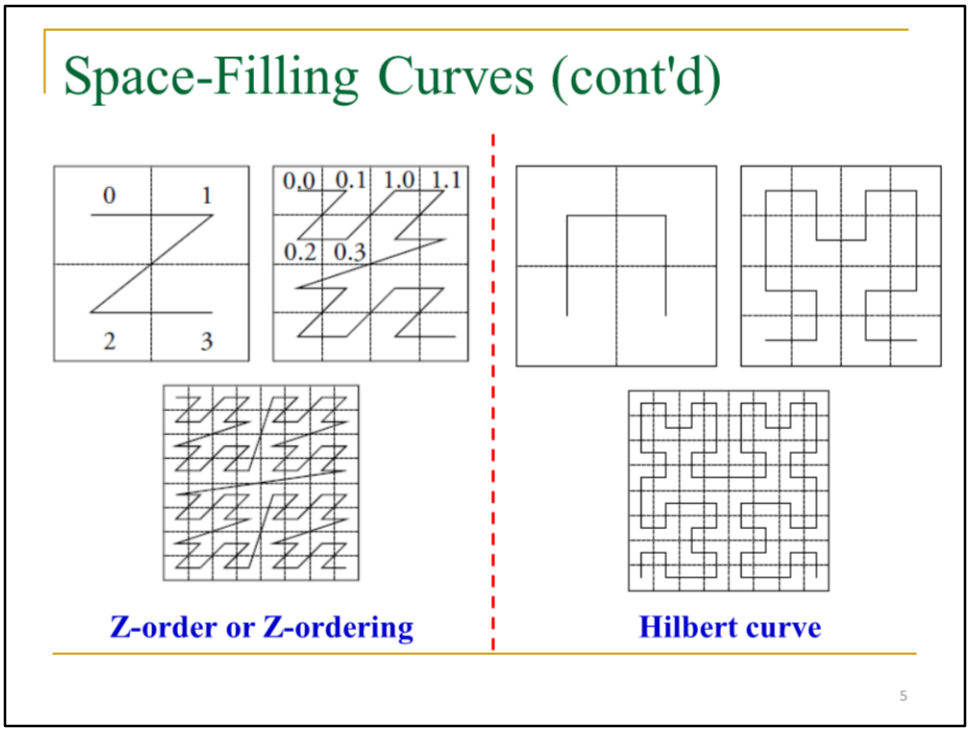
So far, we have talked about B+-tree and extensible hashing, which are usually built over 1-dimensional data. The grid index is designed for multidimensional data, however, usually for 2D or 3D data. In this chapter, we will continue to talk about more indexes for multidimensional data.



The first type of indexes for multidimensional data (typically 2D data) is called space-filling curves. The basic idea of space-filling curves is as follows. Since we have many nice 1-dimensional indexes like B+-tree and hashing indexes, we can convert multidimensional data into one dimensional data, and index the converted 1D data via B+-tree or hashing, which can be used for query processing.

To do the conversion, the space-filling curve defines a total order for cells in the space (just like the increasing or decreasing order of 1-dimensional data). If two close cells in the data space are more likely to be close in the total order, this property can be nicely used for processing queries (e.g., range query).

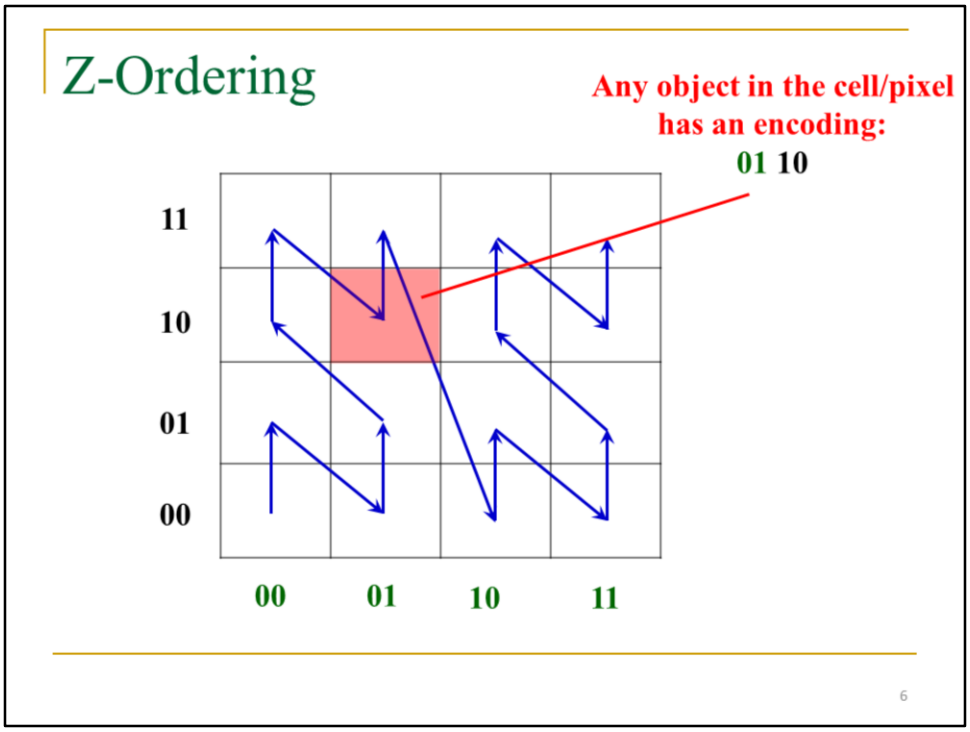
Examples of space-filling curves include Z-order and Hilbert curves.



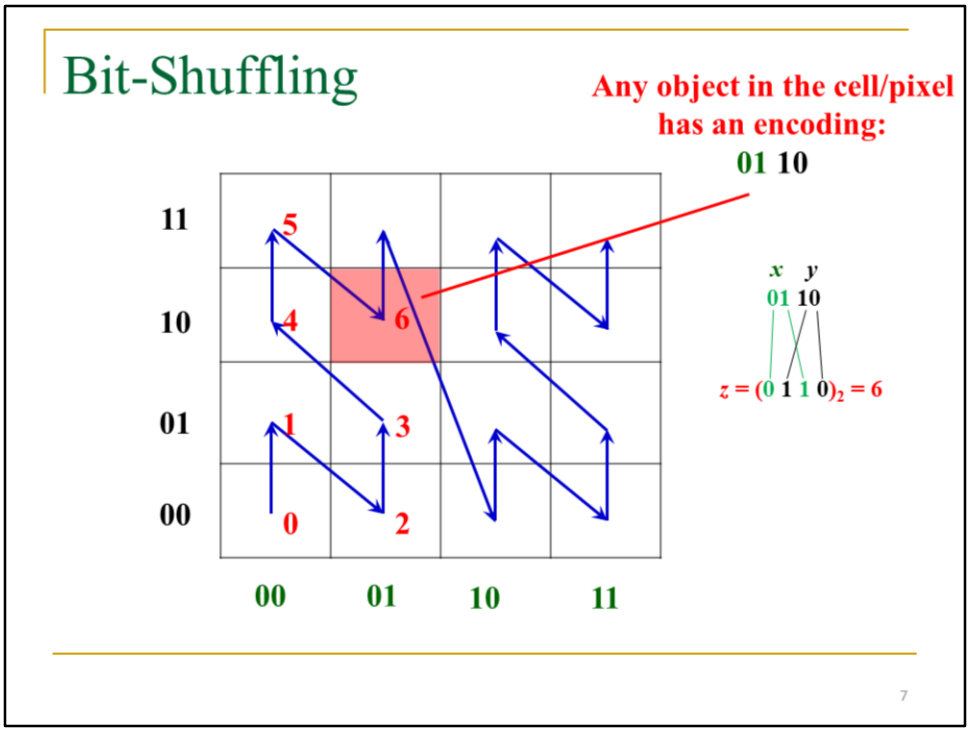
P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases - with application to GIS. *Morgan Kaufmann*, San Francisco, 2002.

Z-order or Z-ordering has a zigzag "Z" shape. For example, in the left figure, if we divide the space into 4 cells, we number them with 0, 1, 2, and 3. Similarly, we can further divide cell 0 (and other cells as well) into another 4 finer cells with numbers 0.0, 0.1, 0.2, and 0.3, which results in 4X4 cells. We can obtain even finer resolution 8X8. This way, we can assign each cell with a unique number, and define their global order.

In the right figure, we can define another space-filling curve of a different shape, called Hilbert curve, which we will discuss later.



Next, we will discuss how to convert a 2D cell into a unique number in the Z-order curve. Assume that we have a 2D data space with 4X4 cells. On each dimension, we name the intervals using 2 bits (due to totally 4 intervals), that is, 00, 01, 10, and 11. Then, for each cell, we have its 2D location (x, y). For example, the red cell in the figure has the 2D location (01, 10).

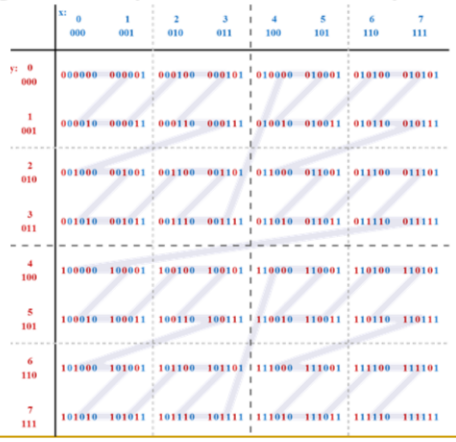


http://www.cs.cmu.edu/~christos/courses/721.S03/LECTURES-PDF/0230-z-ordering.PDF

To convert the 2D location (01, 10) of the red cell, we can use a bit-shuffling method and transform it into a 1-dimensional integer number. That is, we take the first bit of each dimension, combine them as 01, then the second bit from each dimension, combine them as 10, and so on. Finally, we obtain an interleaving binary bit string 0110, and convert it back to decimal number 6, which is exactly the order number of the red cell in the Z-order curve (called z-value).

Bit-Shuffling (cont'd)

- Interleaving the binary coordinate values yields binary z-values



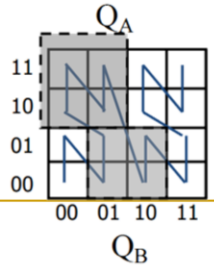
8

https://en.wikipedia.org/wiki/Z-order_curve#/media/File:Z-curve.svg

We can also have the z-value for higher resolution, for example, 8X8 cells. In this case, on each dimension, we need 3 bits to represent 8 intervals, with numbers from 000 to 111. We can apply the same bit-shuffling techniques to obtain z-values.

Indexing Over Z-Values

- Given a number of objects o with 2D coordinates $(o.x, o.y)$
  - Convert each object to a z-value, $o.z$
  - Use a B^+-tree index to organize z-values of all objects
- A range query in a 2D data space can be mapped to a set of intervals for the converted z-values



Q_A → range [4, 7]

Q_B → ranges [2,3] and [8,9]

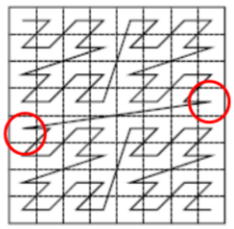
9

http://infolab.usc.edu/csci587/Fall2016/slides/session2-spatial-indexes-no-rtree.pdf

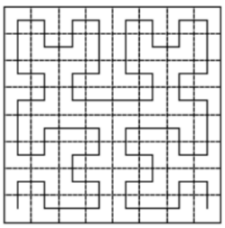
Given the converted z-values from multidimensional objects, we now can use classic 1-dimensional index such as B+-tree to organize them. For a range query in the data space, we can transform the query range to one or multiple query intervals over z-values. For example, for range query Q_A, we can transform Q_A to a query interval [4, 7], and search this interval through the B+-tree index. Similarly, for range query Q_B, we can transform it to two query intervals [2, 3] and [8, 9], and use B+-tree to search for objects with z-values in these two intervals.

Hilbert Curve

- Hilbert Curve
  - Objects that are close in 1D space are also close in 2D space
  - We want objects that are close in 2D space to be close in 1D space
- Z-order has some "jumps", whereas Hilbert curve does not



Z-order

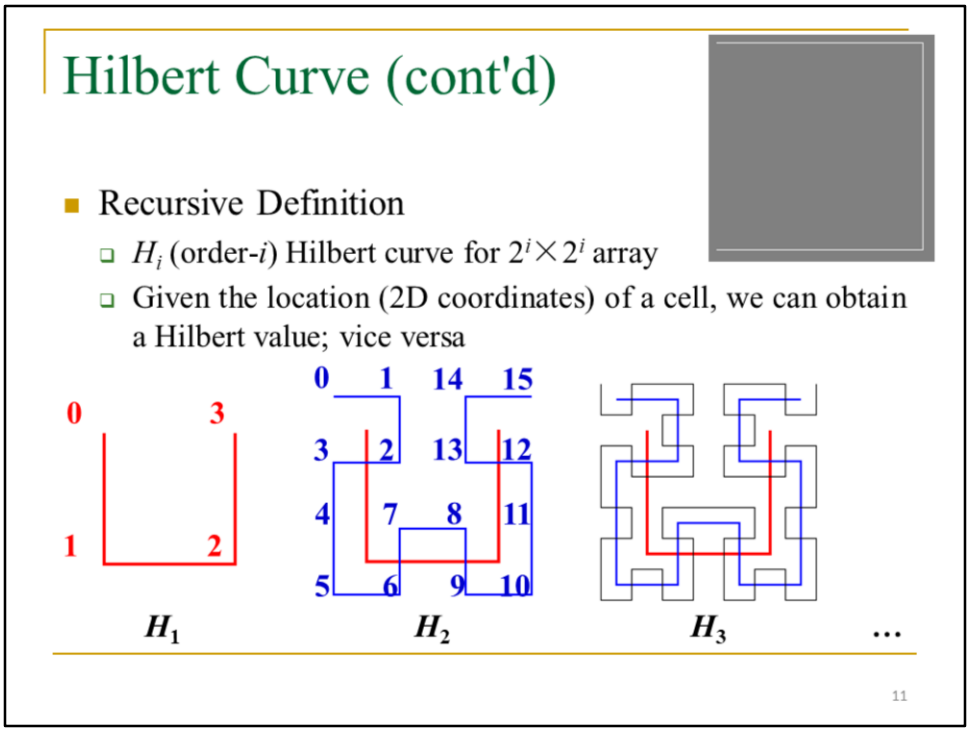


Hilbert curve

10

Hilbert curve is another type of space-filling curve. One disadvantage of Z-order curve is that two consecutive z-values in Z-order curve may correspond to two cells that are far away from each other in multidimensional space. You can see that there is a "jump" in the Z-order curve between two circled cells. This may not be good for queries.

Instead, we may want to have two close objects in the multidimensional space with close 1D converted numbers as well. Hilbert curve is a good choice, since there are no sudden "jumps" in the Hilbert curve.



H. V. Jagadish: Linear Clustering of Objects with Multiple Atributes. ACM SIGMOD Conference 1990: 332-342
https://en.wikipedia.org/wiki/Hilbert_curve

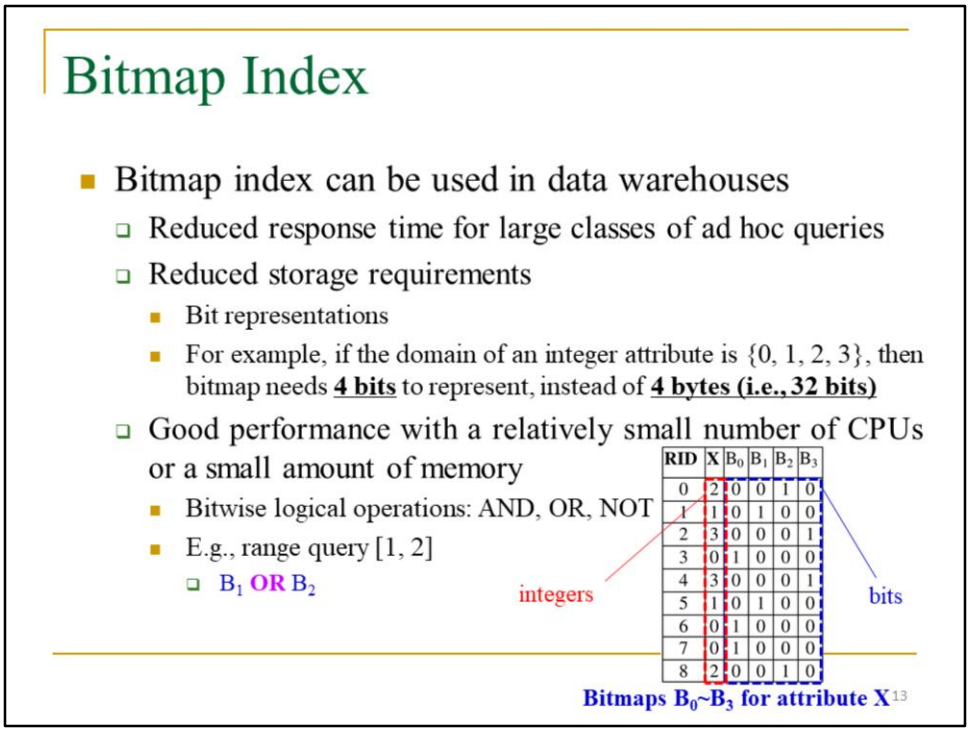
The Hilbert curve H_i is defined recursively starting from H_1 as shown in the figure. In fact, given any 2D location of a cell, we can convert it into a Hilbert value; similarly, given any Hilbert value, we can also convert it back to a vector in multidimensional data space.

Outline

- Z-order, Hilbert curve
- Bitmap index
- Quadtree
- k-d tree
- R-tree, R+-tree, R*-tree

12

Next, we will talk about the bitmap index.



https://en.wikipedia.org/wiki/Bitmap_index
https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm#CIHGAFFF
http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf
https://sdm.lbl.gov/~kewu/ps/LBNL-62756.pdf

Another index, called bitmap index, is often used for data warehouses. The basic idea is to use space-efficient bits to represent the data and apply bit operators to quickly search for query answers.

As an example, if the domain of an integer attribute contains 4 integers, from 0 to 3, then bitmap only needs 4 bits to represent each possible integer, rather than using 32 bits to represent integer variables.

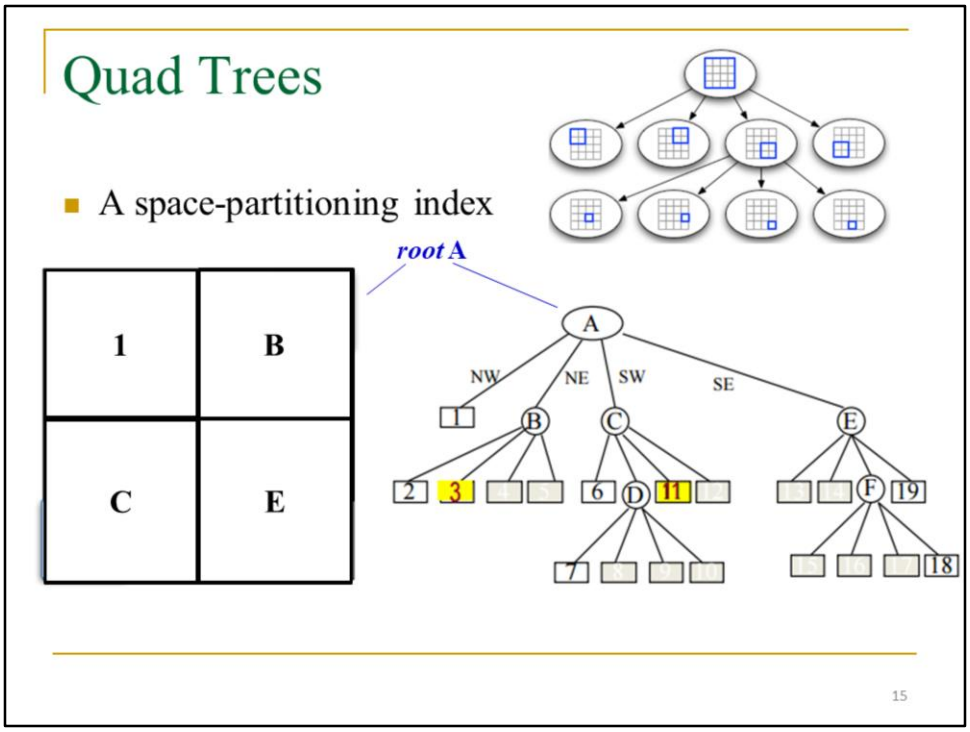
The bitmap index needs a small amount of memory and less CPU time to enable the query processing. As shown in the table, we have 9 records. The first record has integer attribute X equal to 2, therefore, the corresponding bit B_2 is set to 1 (while other bits are 0s). If we have a range query to find records with attribute X falling into interval [1, 2], we can then apply bit-OR operator between vectors B_1 and B_2 (of lengths 9), and those records with bit-OR results equal to 1 are the range query answers.

Outline

- Z-order, Hilbert curve
- Bitmap index
- Quadtree
- k-d tree
- R-tree, R+-tree, R*-tree

14

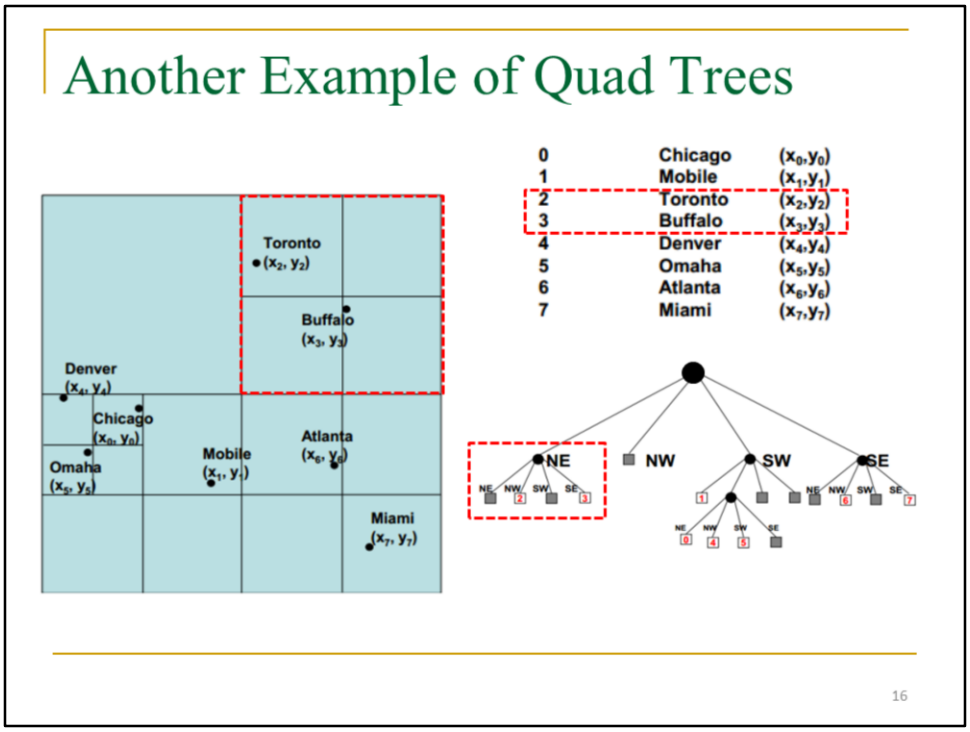
Next, we will discuss the quadtree.



http://infolab.usc.edu/csci587/Fall2016/slides/session2-spatial-indexes-no-rtree.pdf

Raphael Finkel and J.L. Bentley (1974). "Quad Trees: A Data Structure for Retrieval on Composite Keys". Acta Informatica 4 (1): 1–9. doi:10.1007/BF00288933.

The quad tree is a space-partitioning index, which recursively divides the 2D data space into 4 partitions. We show an example of the quad-tree in a 2D data space. The root of the quad tree corresponds to the entire data space. Then, we split the data space into four partitions, 1, B, C, and E, of the same size, which are also 4 children of the root. Next, since there are many objects in node B, we further split B into 4 partitions, 2, 3, 4, and 5. Similarly, we can also further partition nodes C and E, as well as nodes D and F.



Damn Cool Algorithms: Spatial indexing with Quadtrees and Hilbert Curves. http://blog.notdot. net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves

http://web.eecs.utk.edu/~cphillip/cs594_spring2014/quadtree-Allan.pdf

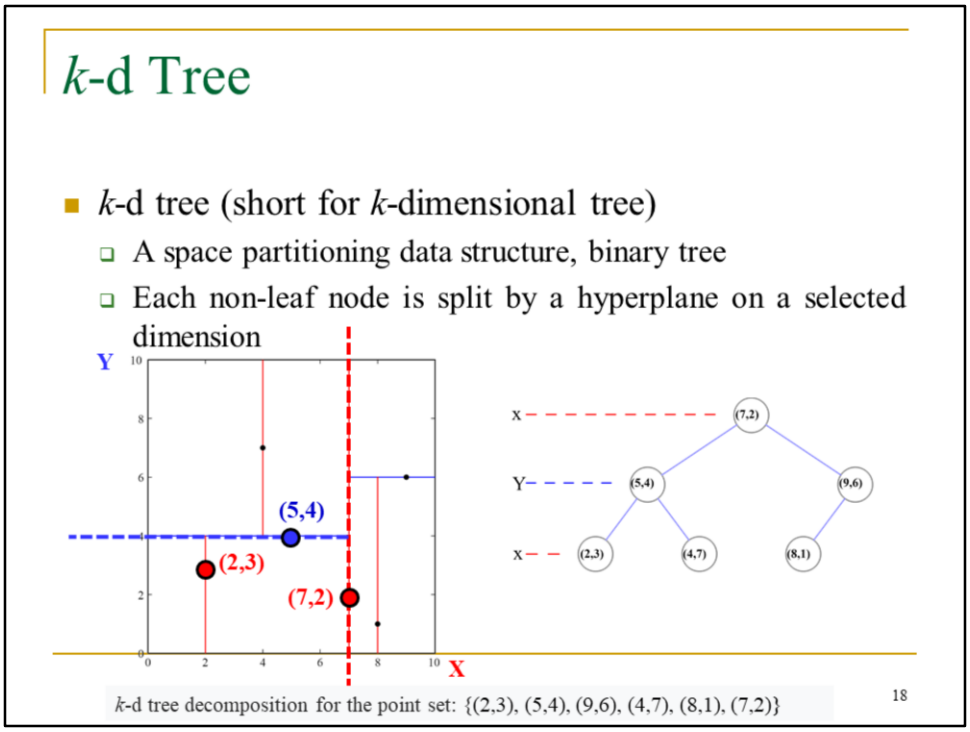
Here is another example of the quad-tree for spatial data, where cities such as Toronto have their 2D locations. We can also build a quad tree over these cities. For example, NE is a node which is further partitioned into 4 cells. Two cities, Toronto and Buffalo, fall into NW and SE cells, respectively.

Outline

- Z-order, Hilbert curve
- Bitmap index
- Quadtree
- k-d tree
- R-tree, R+-tree, R*-tree

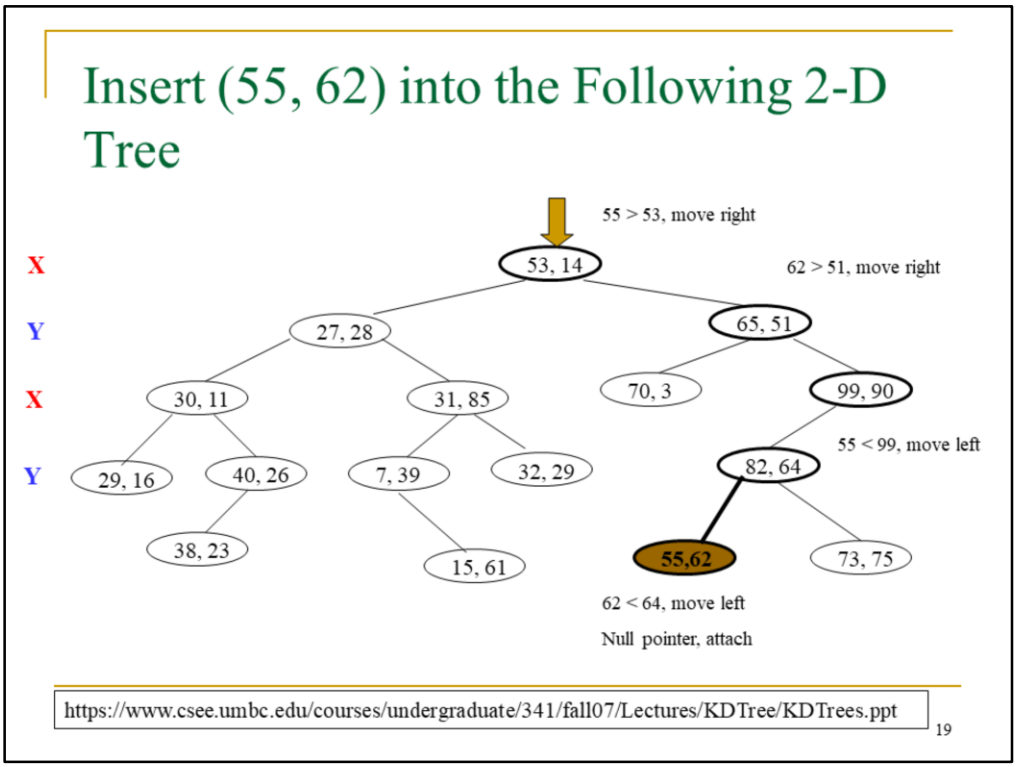
17

We next consider the k-d tree.



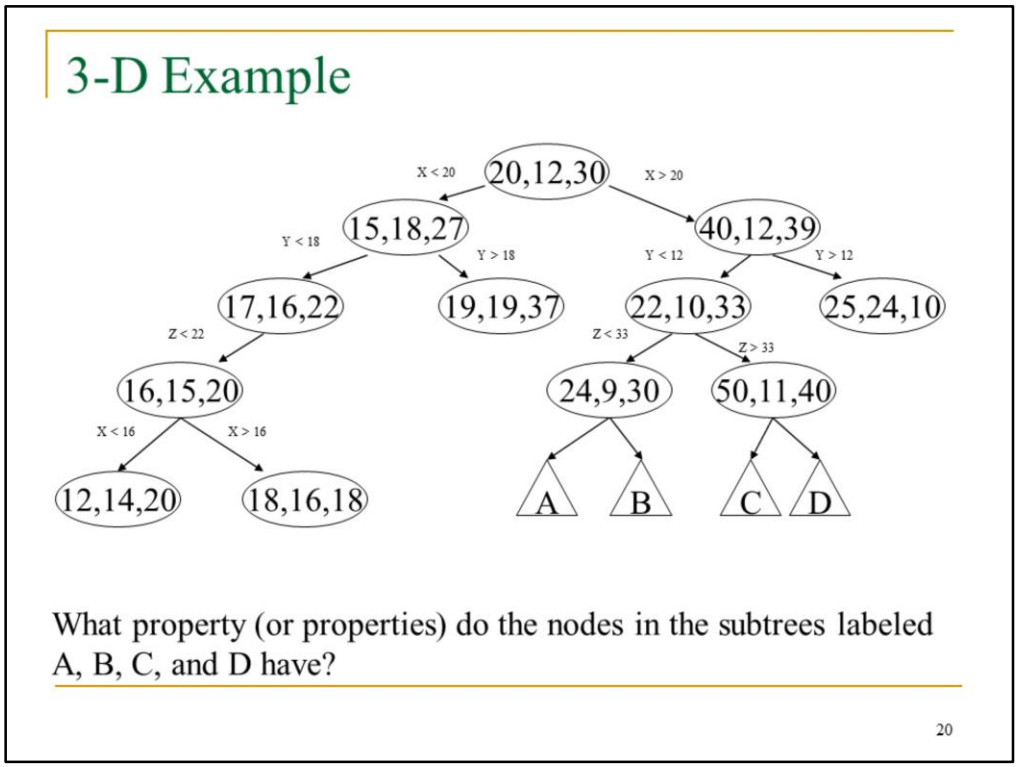
https://en.wikipedia.org/wiki/K-d_tree

Different from the quadtree that directly partitions the multidimensional data space, k-d tree partitions the space on one dimension each time. Taking the figure as an example, in a 2D X-Y space, we first partition the space on X-dimension at the mid splitting point (7, 2) so that two cells contain approximately the same number of points. Then, for each cell, we further perform the partitioning based on Y-axis, that is, the left partition divides the space at the splitting point (5, 4). Then, both left cells are further partitioned based on X-axis again. This way, we can build a k-d tree as shown on the right figure.



https://www.csee.umbc.edu/courses/undergraduate/341/fall07/Lectures/KDTree/KDTrees.ppt

Let us consider how to insert an object (55, 62) into the k-D tree. We first start from the root of the k-D tree. Since the root is partitioned according to X-axis, we compare 55 with X-axis of the splitting point (53, 14). Since 55>53, we descend to the right branch. Next, we compare 62 with Y-axis of the splitting point (65, 51). Since 62>51, we descend to the right branch. Similarly, we descend to the left branch of the splitting point (99, 90) (since 55<99), and then left branch of (82, 64) again (since 62<64). Finally, we reach the leaf node of the k-D tree, and retrieve the object (55, 62).



https://www.csee.umbc.edu/courses/undergraduate/341/fall07/Lectures/KDTree/KDTrees.ppt

How about k-D-Btree?
http://repository.cmu.edu/cgi/viewcontent.cgi?article=3451&context=compsci

The K-D-B-Tree : a search structure for large multidimensional dynamic indexes

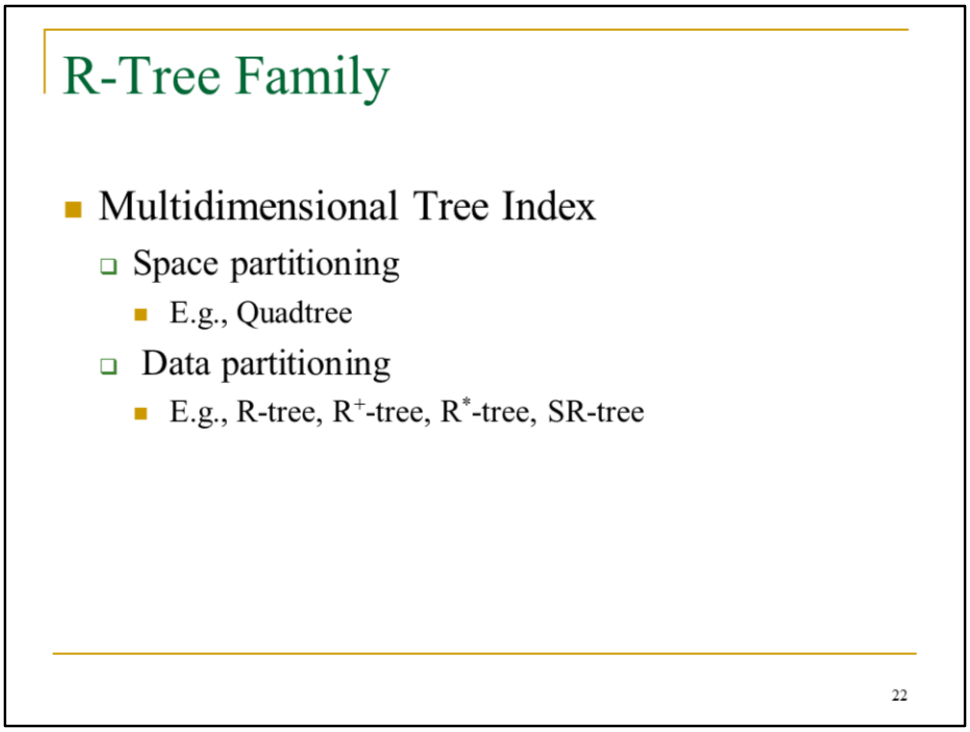
In practice, the k-D tree works for multidimensional data with dimensions 2, 3, and so on. Here is another example of 3D k-D tree.

Outline

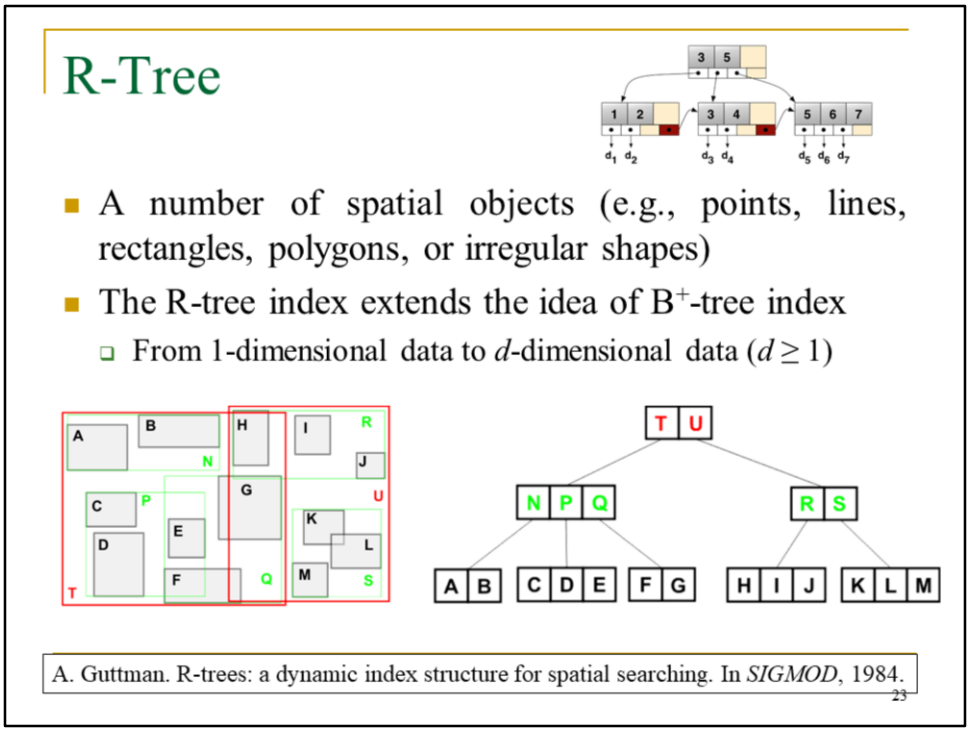
- Z-order, Hilbert curve
- Bitmap index
- Quadtree
- k-d tree
- R-tree, R+-tree, R*-tree

21

We will next consider an important family of the R-tree index and its variants, which has a critical role in facilitating query processing on multidimensional data.



Multidimensional indexes can be classified into space-partitioning index and data-partitioning index. The space partitioning index divides the data space regardless of the data distribution. For example, quadtree always divides the cell into 4 finer cells of the same size in the 2D data space, rather than different sizes based on the data distribution. In contrast, the data-partitioning index, like the R-tree family we will talk about next, will be constructed according to the data distributions.



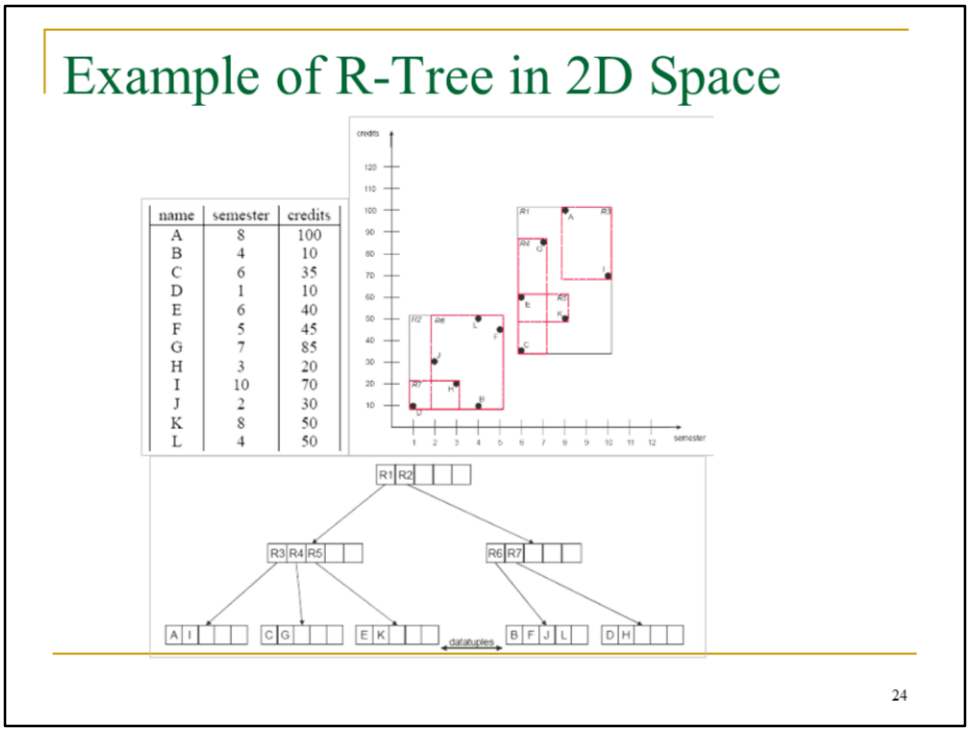
A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.
http://cglab.ca/~cdillaba/comp5409_project/R_Trees.html
https://en.wikipedia.org/wiki/B%2B_tree

In many real applications such as GIS, location-based services, and so on, we may encounter many spatial objects, in the form of points, lines (like roads), rectangles (like buildings), polygons, or other irregular shapes. The R-tree was first proposed by A. Guttman. If you check the citation count of this first R-tree paper on Google Scholar, there are about 9,718 citations by the end of 2018. The R-tree adopts the idea very similar to B+-tree, and extends from 1-dimensional index to d-dimensional index, where d>=1. That is, R-tree is a balanced tree structure that supports the dynamic updates (including insertions and deletions).

The two figures show an example of the R-tree over 2D spatial objects of rectangular shape, A~M. R-tree uses a notion called *minimum bounding rectangle* (MBR) to minimally bound spatial objects. For example, objects A and B are grouped and enveloped by an MBR N; similarly, objects C, D, and E are bounded by MBR P, and so on. Then, MBR nodes such as N, P, and Q can be further grouped by an even larger MBR T. This way, an R-tree can be built over such spatial objects.
`

Unlike one-dimensional data that have a unique order, objects in multidimensional

space do not have a global order due to multiple attributes/dimensions. Therefore, for B+-tree, intermediate nodes do not have overlapping value intervals. However, when constructing the R-tree, there are some inevitable overlaps among MBRs. The disadvantage of these overlaps is to increase the search cost. Therefore, later on we will mention that we want to minimize the overlapping areas among MBRs when constructing or updating the R-tree. `

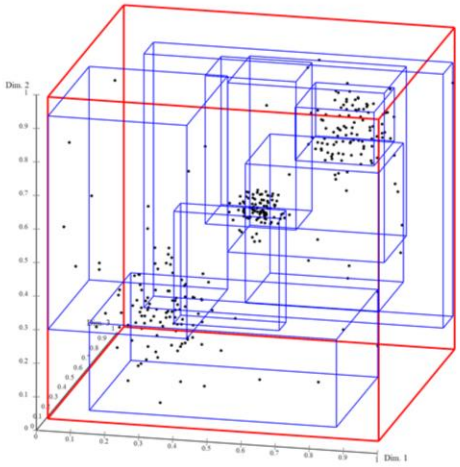


http://sysnet.ucsd.edu/~cfleizac/cse262/R-Trees.ppt

Here is another example of R-tree over data points, where each data point is a student record with attributes semester and credits. In the R-tree, each leaf node stores spatial objects (or student records), whereas each non-leaf node contains a number of node entries, each with an MBR for objects under this node entry.

R-Tree in the Multidimensional Space

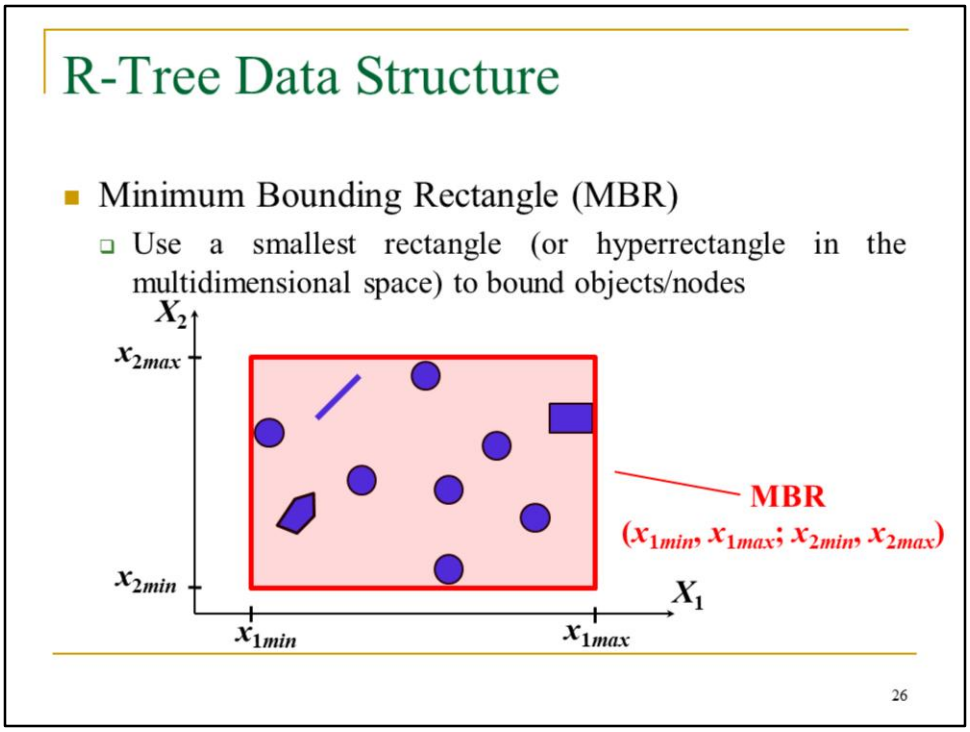
3D R-Tree:



https://en.wikipedia.org/wiki/R-tree

25

The R-tree works not only for 2D objects, but also for 3D objects like the one shown in the figure. It can also support even higher dimensional data, say 4D, 5D, and so on.



Next, we will discuss an important term in the R-tree structure, minimum bounding rectangle (or MBR). The MBR can be used to summarize a number of spatial objects. As shown in the figure, spatial objects can be of many types, such as points, lines, rectangles, circles, polygons, and so on. We can use a minimum rectangle (i.e., MBR) to bound these spatial objects. To represent this MBR, we only need to record the minimum boundary on each dimension. For example, in the 2D case, we only need to record [x_{1min}, x_{1max}] on X1-axis, and [x_{2min}, x_{2max}] on the X2-axis.

R-Tree Data Structure (cont'd)

- In a *d*-dimensional space, an MBR in the R-tree is in the form of:
  - $(x_{1min}, x_{1max}; x_{2min}, x_{2max}; \ldots; x_{dmin}, x_{dmax})$

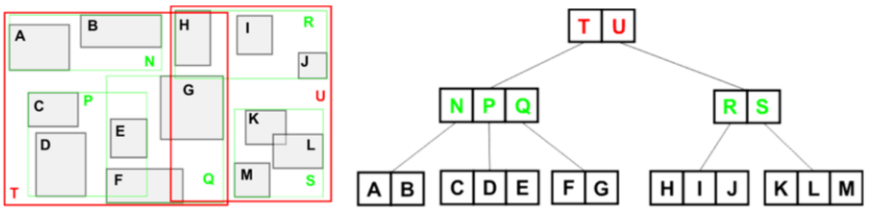
27

In a general d-dimensional case, each MBR needs 2d floating-point numbers, (x_{1min}, x_{1max}; x_{2min}, x_{2max}; ...; x_{dmin}, x_{dmax}), where [x_{imin}, x_{imax}] is the tightest interval that bounds all objects in the MBR along the i-th dimension. For example, in 2D space, we need 4 numbers to represent the MBR; for 3D, it needs 6 numbers; and so on.

R-Tree Data Structure (cont'd)

- R-tree is a *height-balanced* **multi-way external memory** tree over n multidimensional data objects (height: $log_F(n)$)
  - **Non-leaf node (or intermediate node):** contains a number of entries (MBRs) that minimally bound their child nodes, as well as pointers pointing to child nodes
  - **Leaf node:** contains spatial objects

node *fanout*: $F \in [m, M]$, where $m \leq M/2$



28

R-tree is a height-balanced tree over multidimensional objects. It means that to access spatial objects in leaf nodes, the numbers of page accesses are the same for all objects, due to the same height from root to leaf nodes.

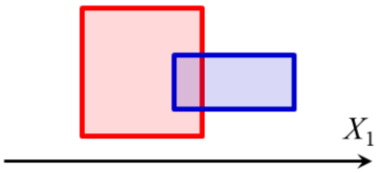
The R-tree contains two types of nodes, non-leaf nodes (or called intermediate nodes) and leaf nodes. Each non-leaf node contains a number of child entries, and each child entry consists of an MBR (minimally bounding all objects under this entry) and a pointer pointing to its child (or subtree).

On the other hand, each leaf node contains multiple spatial objects, also represented by MBRs or multidimensional points.

Each leaf/non-leaf node in the R-tree has a constraint of its fanout F within interval [m, M], such that m<=M/2, where M is the maximum number of possible entries that can be stored in a node (or disk page).

R-Tree Data Structure (cont'd)

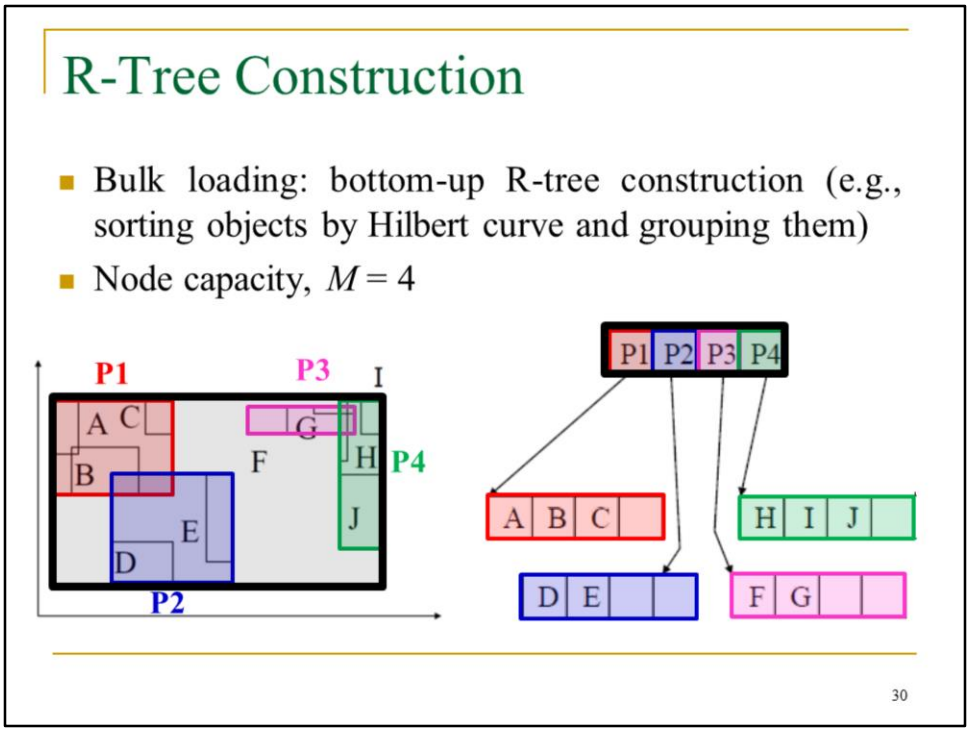
- In a *d*-dimensional space
  - Given 2 MBRs, $A = (x_{1min}, x_{1max}; x_{2min}, x_{2max}; \ldots; x_{dmin}, x_{dmax})$ and $B = (y_{1min}, y_{1max}; y_{2min}, y_{2max}; \ldots; y_{dmin}, y_{dmax})$
  - What is the MBR, E, that bounds both A and B?
    - $E = ?$



29

Here is a question for you. In a d-dimensional space, if we have two MBRs, A and B, then what is the MBR E that minimally bounds both A and B?

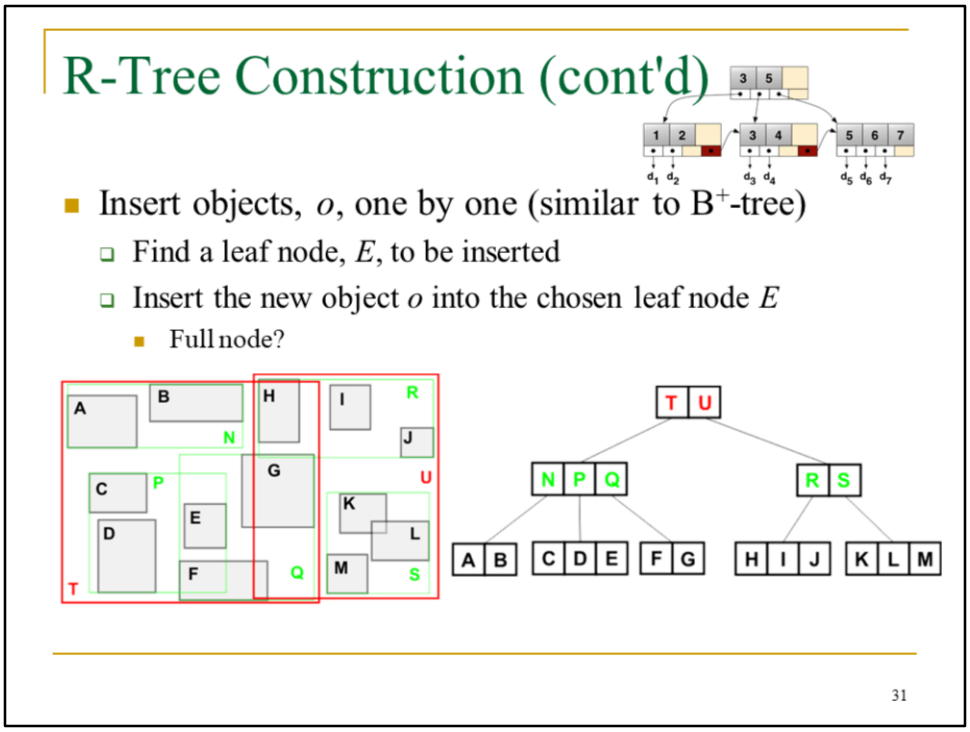
Hints: you should consider each dimension of the MBR E separately. You can check a 2D example in X1-X2 space first, and then generalize it to d-dimensional space.



http://infolab.usc.edu/csci587/Fall2016/slides/session3-spatial-index-rtree.pdf

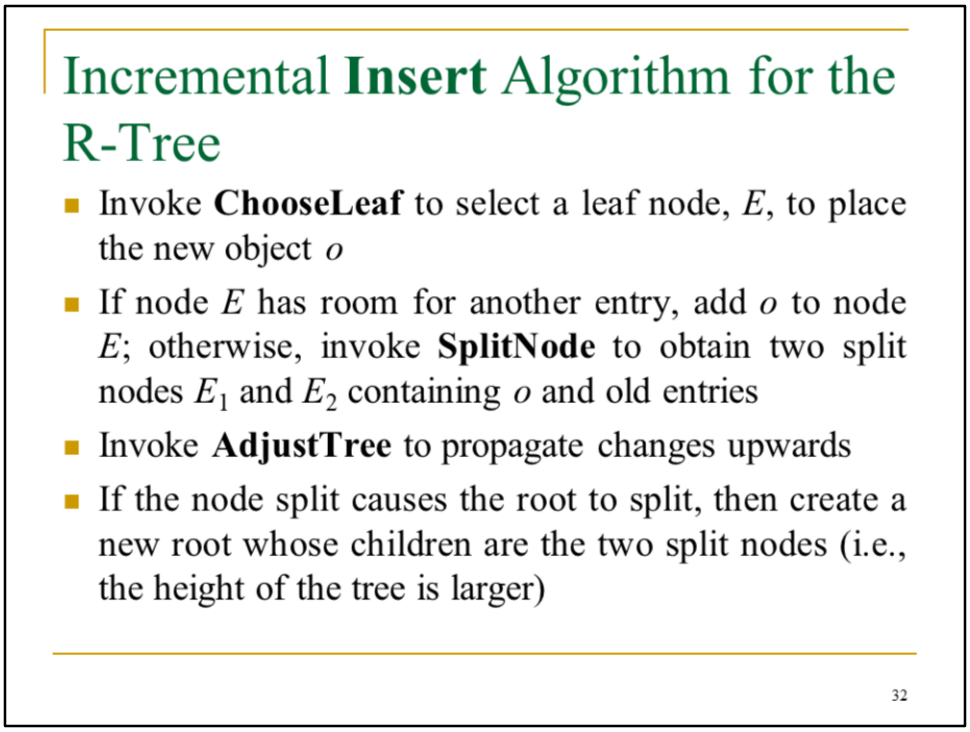
Bulk loading of R-tree
http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-74/files/FORUM_18.pdf

To construct the R-tree, there are two styles, bottom-up and top-down. The bottom-up style is to first obtain leaf nodes of the R-tree, by dividing the data sets into multiple partitions (i.e., leaf nodes), and then iteratively group partitions via MBRs until one root is obtained. This method is also called bulk loading, which sorts spatial objects by their Hilbert values, and then performs the partitioning. The figure shows the process of the bulk loading. It first obtains leaf nodes P1 ~ P4, such that each leaf node satisfies the constraint [m, M] of the fanout F. Then, it groups spatially close MBRs as a larger MBR so that an R-tree structure is built.



A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

The other style, top-down approach, is to incrementally construct the R-tree by inserting spatial objects one by one, similar to the B+-tree. In order to insert an object o into the R-tree, we first need to find a leaf node E to accommodate this object o, and then insert this object into the chosen leaf node. During the insertion, we need to solve the overflow problem similar to B+-tree (i.e., what if the chosen leaf node is full). We will discuss later that we will also spilt the overflowing leaf node, and update the tree index. Note that, with different orders of inserting objects, we may obtain different R-tree indexes.

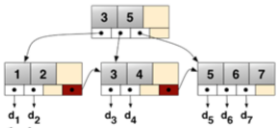


A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

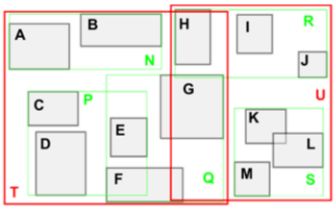
Here is the pseudo code of the insert algorithm for the R-tree. First, we call ChooseLeaf function to descend from the root of the R-tree, and find an appropriate leaf node to place the new object o. If the leaf node is not full, we can directly add o to the leaf node; otherwise, we need to split this leaf node by invoking SplitNode function. Due to the node splitting, we also need to update the index structure by invoking AdjustTree function to propagate changes.

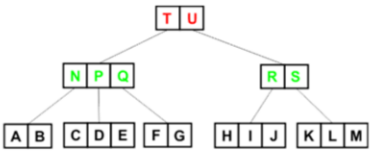
If the root is split, then a new root is created to point to the 2 split nodes, and the height of the tree is increased by 1.

ChooseLeaf Algorithm



- Find a leaf node, E, to insert a new object o
  - Decide the branch to descend
  - Select a branch such that the insertion causes the least enlargement of the rectangle, or MBR (intuition?)
  - In the case of ties, choose the branch with the MBR of the smallest area (intuition?)



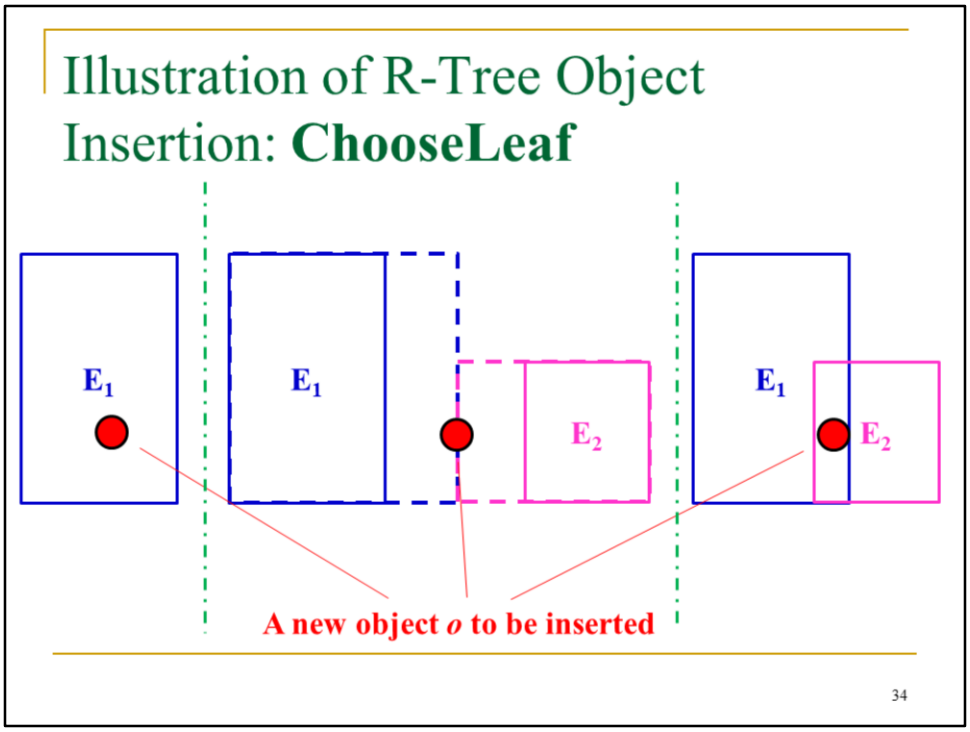


33

For the ChooseLeaf function, we will start from the root of the R-tree, and decide which branch to descend in order to accommodate the new object o.

For R-tree, the strategy is to select a branch such that the insertion will cause the least enlargement of the MBR. Then, in the case of ties, a branch with the smallest area will be selected.

Intuitively, if the MBR is enlarged very significantly due to the insertion of a new object, then the query performance of the R-tree will also degrade, that is, this enlarged MBR will have higher chance to be accessed. The selection of the MBR with the smallest area in the case of tie has the same reason.



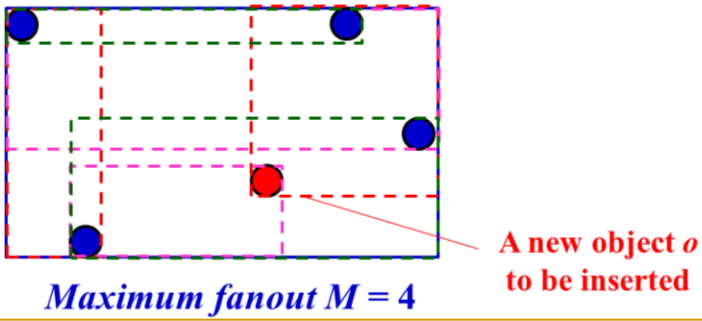
Here is an example of 3 cases for object insertion. In the left figure, if a new object is inside the MBR node E_1, we can insert it into the node without enlarging the MBR.

In the middle figure, by comparing the enlargement of MBRs E_1 and E_2, we can see that E_2 has smaller enlarged area than E_1. Therefore, according to ChooseLeaf condition, we will choose E_2 with the smaller enlargement.

In the right figure, the new object is inside both E_1 and E_2, with equal (0) enlargement if inserted. In this case of tie, we will select E_2 to insert the new object, since E_2 itself has smaller area.

R-Tree Object Insertion: **SplitNode**

- Insert the new object into the chosen leaf node
  - In the case of full node?
    - How to split the node

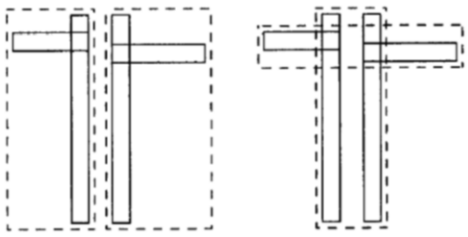


35

Once we find the leaf node to insert the new object, we need to handle the case where the leaf node is full. As we mentioned earlier, we can split the node into two, and redistribute objects between the two nodes. However, there are many different approaches to do the object distribution between two split nodes. We need to decide which splitting is the best.

SplitNode Heuristics

- Exhaustive Algorithm
  - Generate all possible groupings and choose the best one with ***the minimum area***
  - Time complexity: in the worse case, $O(2^{M-1})$ possible groupings (M can be as large as 50)



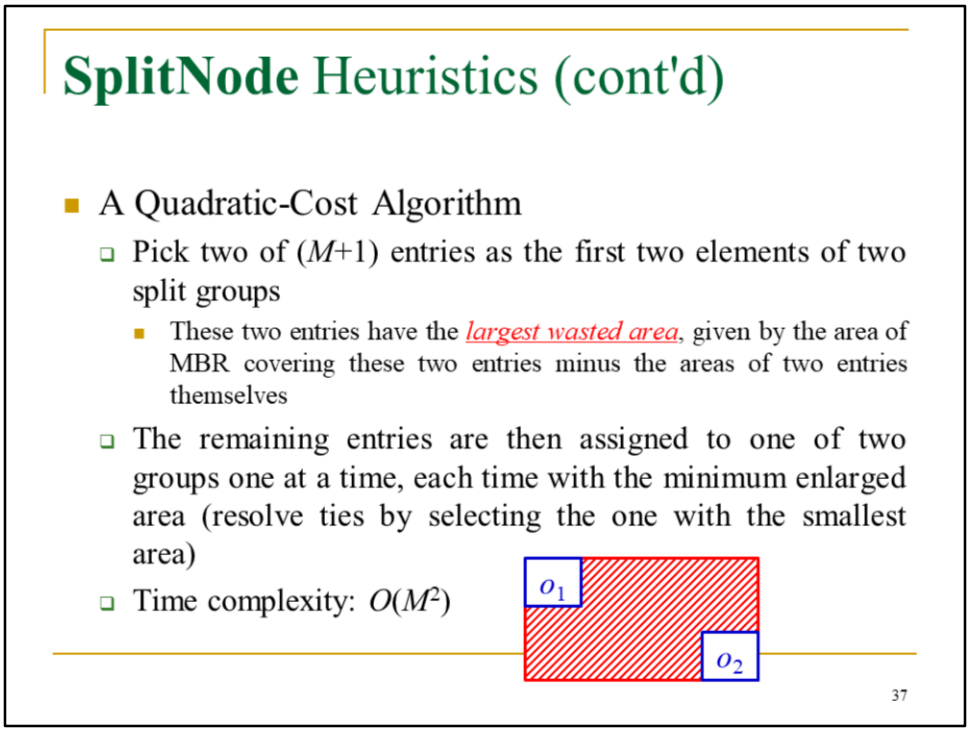
Bad split Good split

36

A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

Intuitively, we would like to obtain MBRs of nodes as small as possible, so that the chance that two split nodes are accessed during query processing is low (i.e., with high query performance). As shown in the figure, for the same objects/nodes, there are different splitting methods. The left splitting method is bad, since it incurs large areas that do not contain any objects. On the other hand, the right one is good, although there are some small overlaps between two MBRs after grouping. Therefore, we aim to have the splitting of the node such that the resulting two MBRs have minimum area.

One exhaustive algorithm is to enumerate all possible grouping strategies, and then select the one with the minimum area. The algorithm has exponential time complexity O(2^{M-1}), where M is the maximum page capacity. When M=50, this time complexity is rather high.

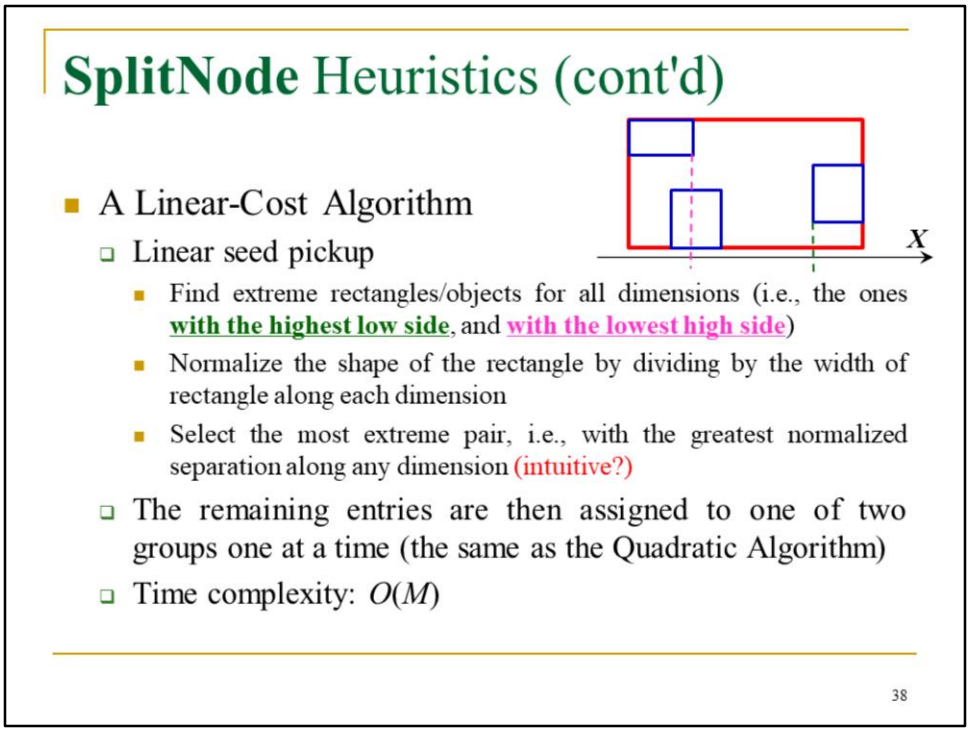


A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

Therefore, some heuristic-based algorithms are proposed with approximate solution, but higher efficiency. One of them is the quadratic-cost algorithm. The basic idea is to select 2 out of (M+1) entries that are far away from each other in the overflow node, and then assign the remaining entries to one of these two entries, in order to form 2 groups.

To select two entries, e.g., o_1 and o_2 in the figure, a wasted area is defined, as shown in the area with sloped lines, which is the area of MBR covering these two entries minus the areas of two entries themselves. We select two of (M+1) entries with the largest wasted area, which takes quadratic cost w.r.t. M. Each selected entry serves as the seed of a group.

The remaining entries are assigned to one of the two entries one at a time, with the minimum enlarged area. In the case of tie, we assign the entry to a group with the smallest MBR area. This step has O(M) cost. So the total time complexity of the algorithm is O(M^2).



A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.

The main cost of the quadratic-cost algorithm is in the seed pickup. To further reduce such a cost, another heuristic-based algorithm is proposed, called the linear-cost algorithm, which considers the seed pickup with linear cost. The basic idea is to find extreme rectangle objects for all dimensions, and select two objects/MBRs far away from each other on some dimension.

Specifically, for each dimension, we find two objects/MBRs with the highest low side (green part in the figure) and the lowest high side (purple part in the figure). In order to avoid the impact of scales along each dimension, we normalize the shape of the rectangle by dividing by the side length on each dimension, when computing the highest low side and the lowest high side. Finally, we select an extreme pair of objects on a dimension with the highest normalized separation (i.e., the highest low side – the lowest high side). This pair of 2 objects can be used as the seeds for splitting the node. Intuitively, the objects with the highest normalized separation are expected to be far away from each other.

The next step is the same as the quadratic method, which assigns each of the remaining entries to one of the two groups (e.g., closest to a seed object).

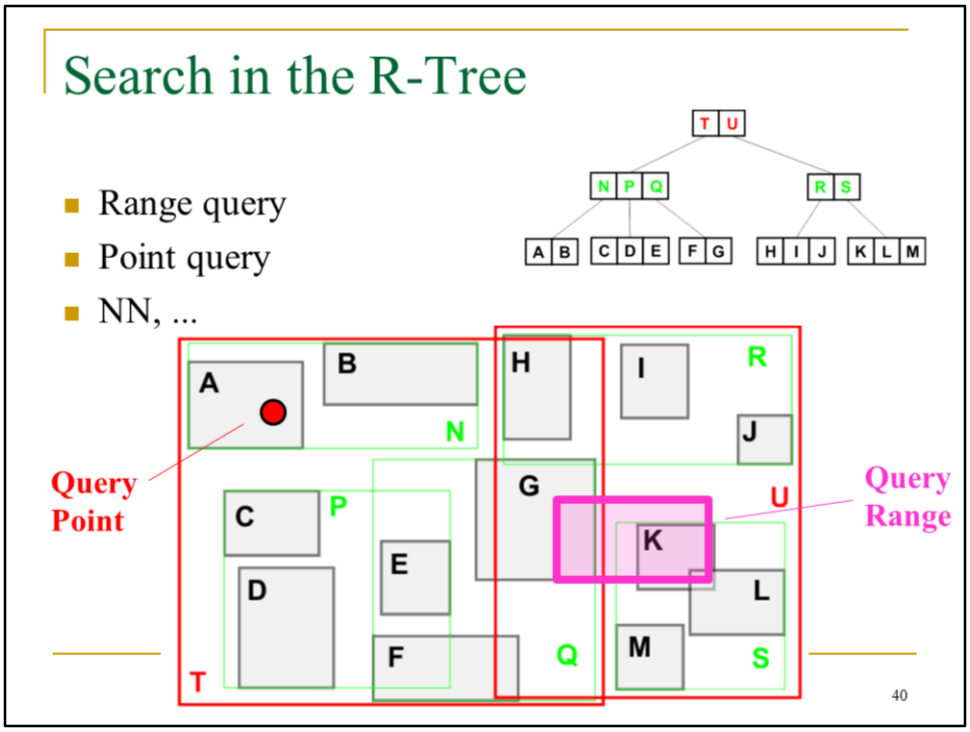
Due to lower time complexity for seed pickup, the time complexity is now given by O(M), that is, linear cost.

Object Deletion for the R-Tree

- Find the location of an object o in a leaf node E
- Delete the object o in the leaf node E
- If the leaf node E has less than m objects, then handle the underflow of the leaf node
  - Delete the leaf node E from the R-tree
  - Re-insert objects in E by Insert Algorithm

39

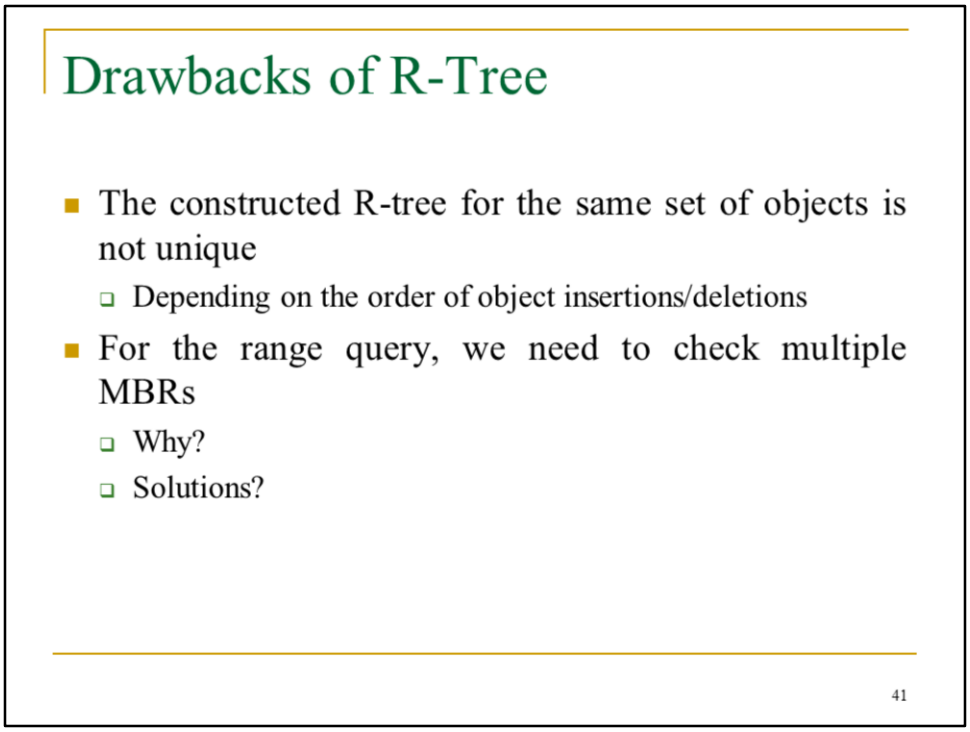
To delete an object o, we first find the location of object o in the leaf node, and then delete this object from this leaf node. If the deletion causes the underflow (i.e., less than m objects after deletion), then we need to handle the underflow. Different from the B+-tree that merges underflow leaf node with its siblings, in the R-tree, we will directly delete the underflow leaf node from the R-tree, and re-insert objects in the leaf node using Insert algorithm as mentioned before. It is said that by using re-insertion, the R-tree can achieve better grouping effect of MBR nodes (e.g., the pruning power for range queries).



There are many query types that can be supported by the R-tree index, such as range query, point query and nearest neighbor query. Given a query range, a range query retrieves all the objects/MBRs that fall into this range. In the example of the figure, objects G, K, and L are overlapping with the query range, and should be returned by the range query.

A point query checks if a given query point exists in the spatial database. We can also find the answer to the point query by searching within the R-tree.

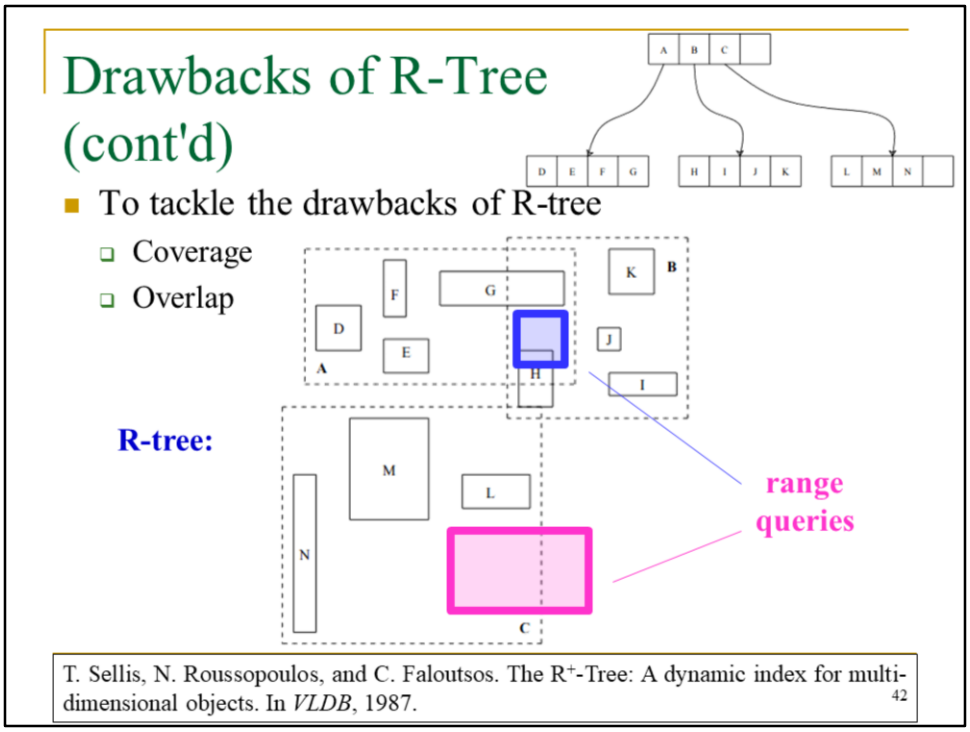
In addition, we can also use the R-tree to answer nearest neighbor query (or NN query for short), which returns an object nearest a given query point.



http://infolab.usc.edu/csci587/Fall2016/slides/session3-spatial-index-rtree.pdf

The R-tree index has its own drawbacks. For example, we may not be able to efficiently obtain an optimal index structure, since different orders of object insertions/deletions may result in different R-trees.

Moreover, for queries like range query, when we traverse the R-tree index, we may have to access multiple MBRs that are not necessary. The reason for this is that MBR nodes in the R-tree may overlap with each other. Therefore, even if an MBR does not have any object in that overlapping area, we have to access it when the overlapping area intersects with the query range.

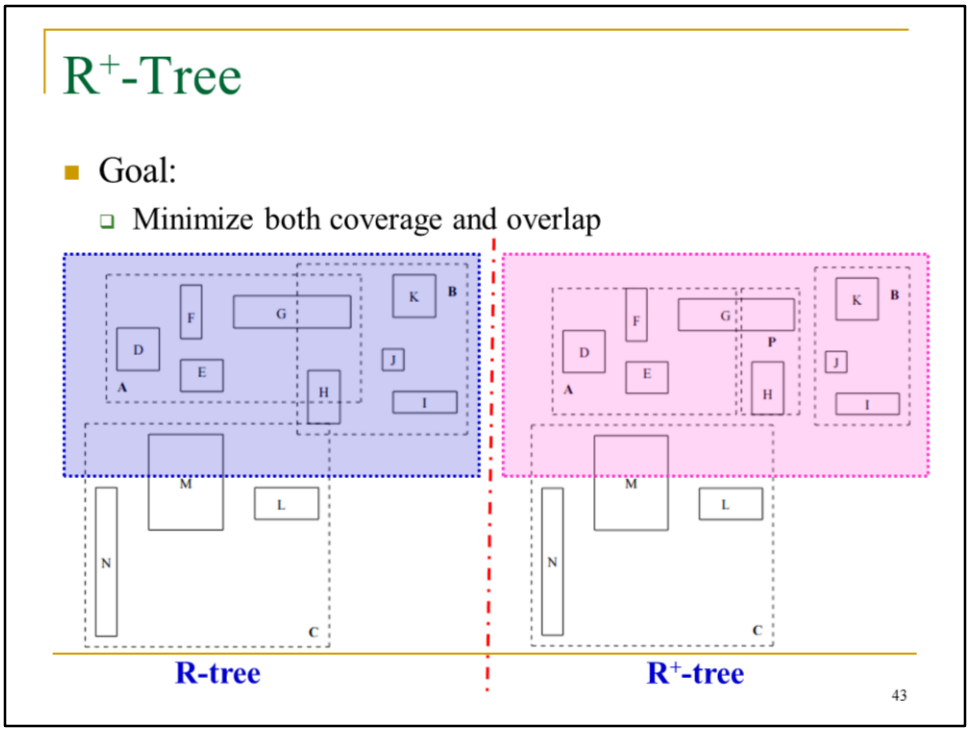


T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-Tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.

As an example in the figure, even if node C does not contain any objects falling into the purple query range, we have to access it (as MBR of node C intersects with the query range).

For the other range query, since the blue query range intersects with the overlapping area of nodes A and B, we have to access both nodes (even if only node B containing object H actually needs to be accessed).

From these examples, we can see that coverage and overlap are two important factors of MBR nodes when we evaluate the performance of the constructed R-tree. The coverage is the area covered by the MBR, whereas the overlap is the overlapping area among MBRs.



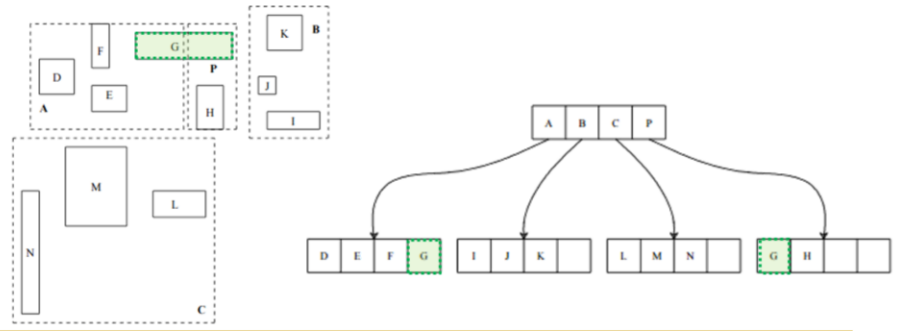
T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.

R+-tree is exactly a variant of the R-tree, which proposes to construct a tree index that minimizes both coverage and overlap.

In the two figures, we show the differences between R-tree and R+-tree using the same example. In the R-tree, nodes A and B are overlapping with each other, but in the R+-tree, nodes A, B, and P do not have any overlaps, and they have the best coverage.

Differences of R^+-Tree from R-Tree

- R^+-tree nodes are not guaranteed to be at least half filled
- The entries of any internal node do not overlap
- An object ID may be stored in more than one leaf node

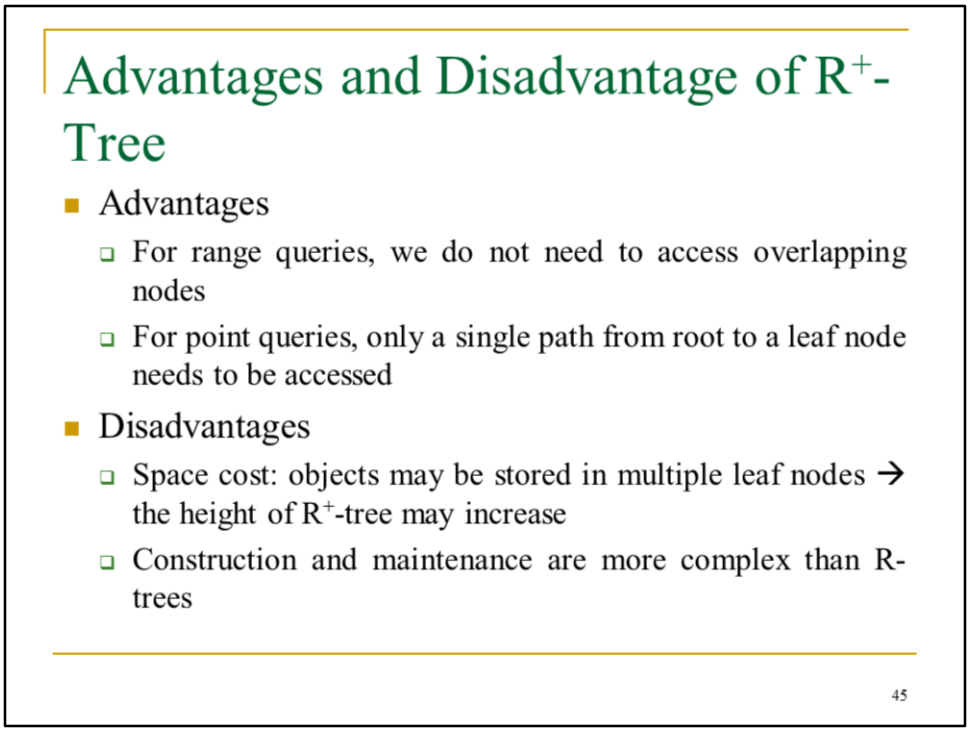


https://en.wikipedia.org/wiki/R%2B_tree

44

https://en.wikipedia.org/wiki/R%2B_tree

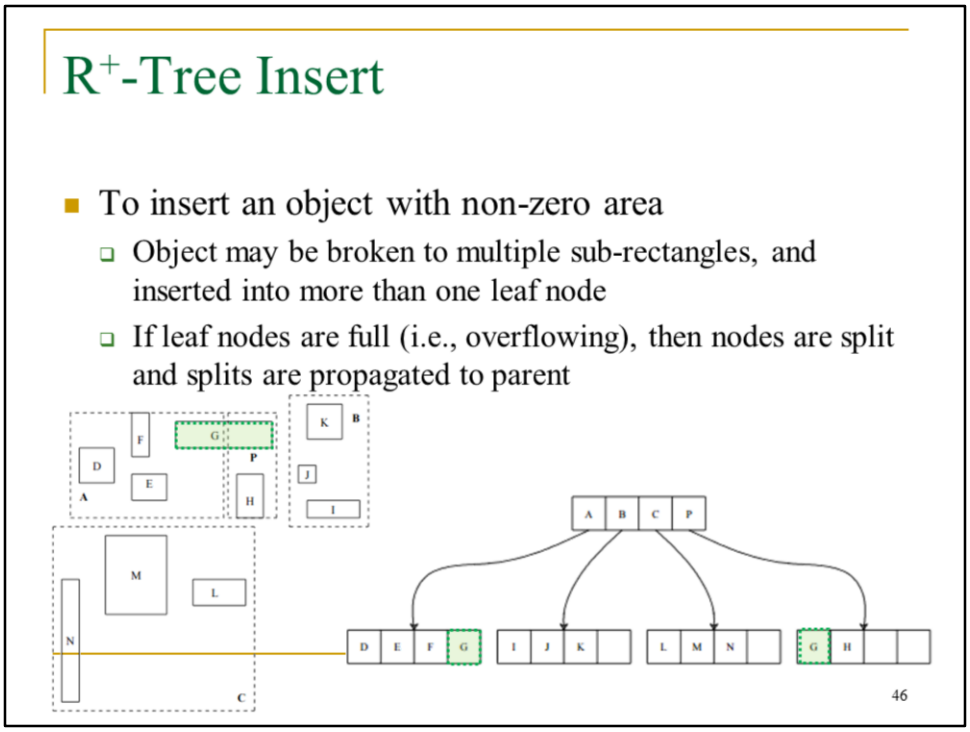
Different from R-tree, R+-tree nodes are not guaranteed to be at least half full, and entries of any internal node do not overlap. As shown in the figure, object G may span across multiple nodes. Therefore, G may appear in more than 1 leaf node in the R+-tree, which is also another difference from the R-tree.



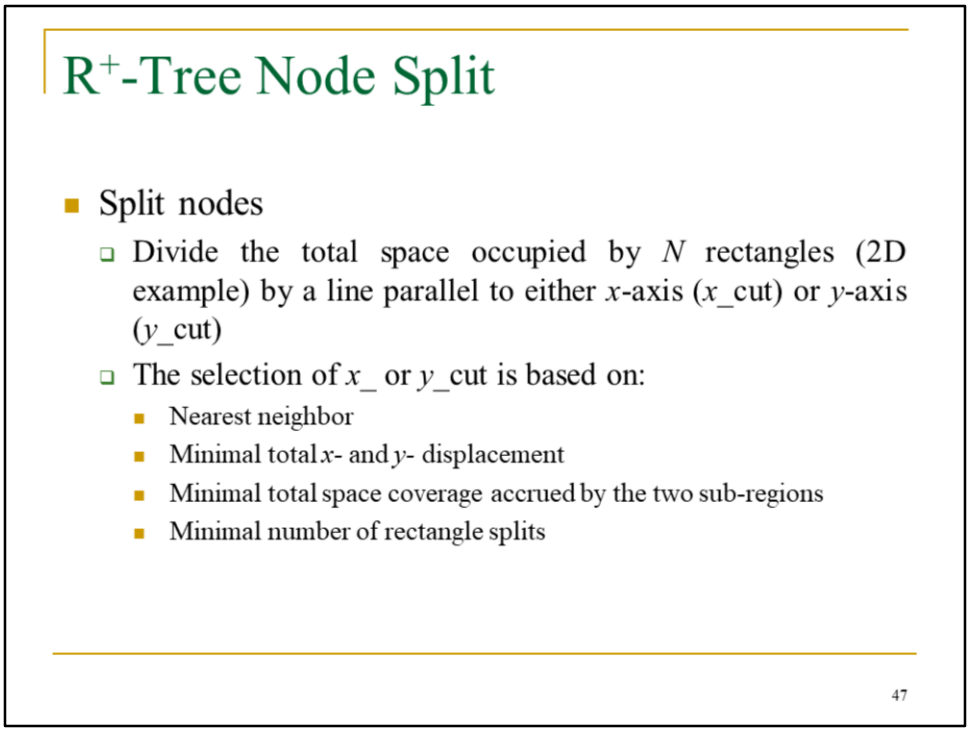
https://en.wikipedia.org/wiki/R%2B_tree#Advantages

There are pros and cons of the R+-tree. For the advantages, since there are no overlapping nodes, we do not need to access multiple overlapping nodes for range queries and point queries.

For the disadvantages, since objects may appear multiple times in the leaf nodes, it requires extra spaces and the height of the R+-tree may increase, which leads to higher I/O cost. Moreover, due to duplicate objects in leaf nodes, the construction and dynamic maintenance of the R+-tree are more complex.

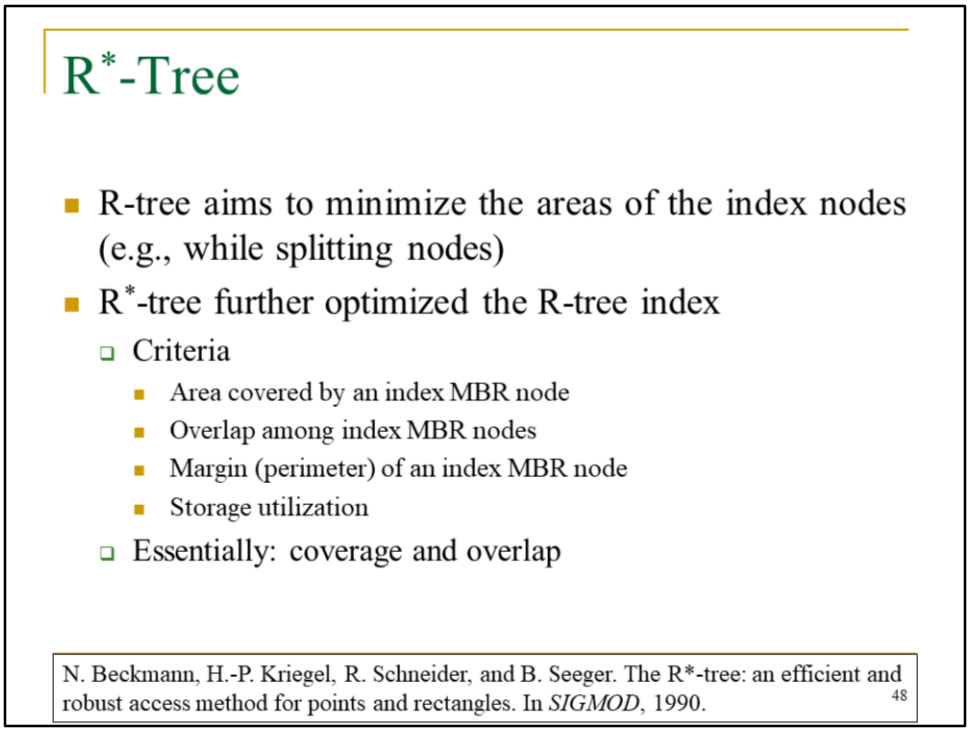


For the insert operator in R+-tree, objects with non-zero area may be broken into multiple sub-rectangles, and inserted into different leaf nodes. For the overflow of the leaf node, we also need to split the node into two, and propagate the changes to ascendants.



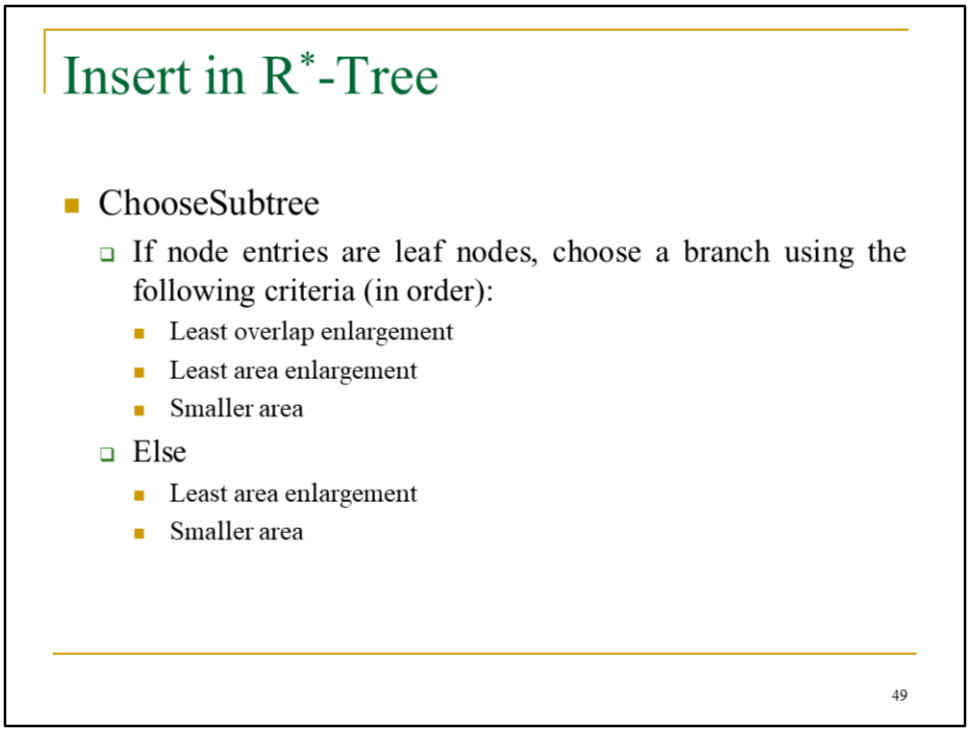
To split the nodes in R+-tree, in the 2D case, we divide the space by a line parallel to either x- or y- dimension, and obtain two groups of objects for two split nodes, respectively.

The selection of cut on x- or y- axis can be based on different criteria. For your interest, you can check the details in the R+-tree paper.



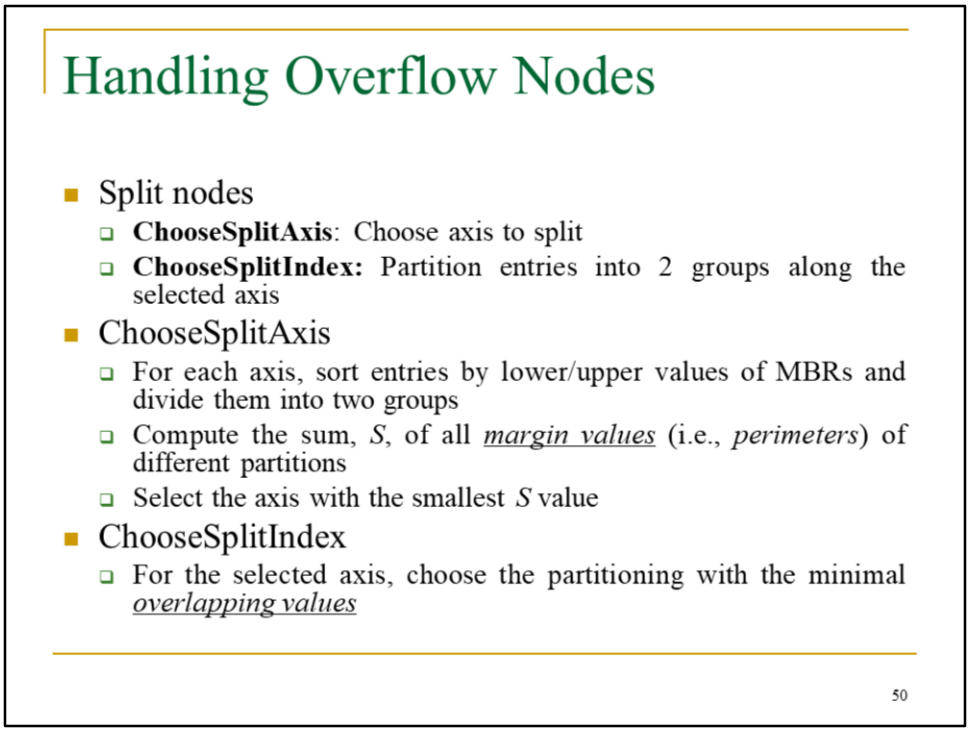
https://en.wikipedia.org/wiki/R*_tree
http://infolab.usc.edu/csci587/Fall2016/slides/session3-spatial-index-rtree.pdf

R*-tree is another variant of the R-tree index, which optimizes the R-tree by considering more detailed criteria. For example, in addition to the area covered by an index MBR and overlap among index nodes for R-tree index, it also considers the margin (perimeter) of the MBR node (which is related to the shape of the MBR) and the storage utilization. But essentially it is still to minimize the coverage and overlap.



N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

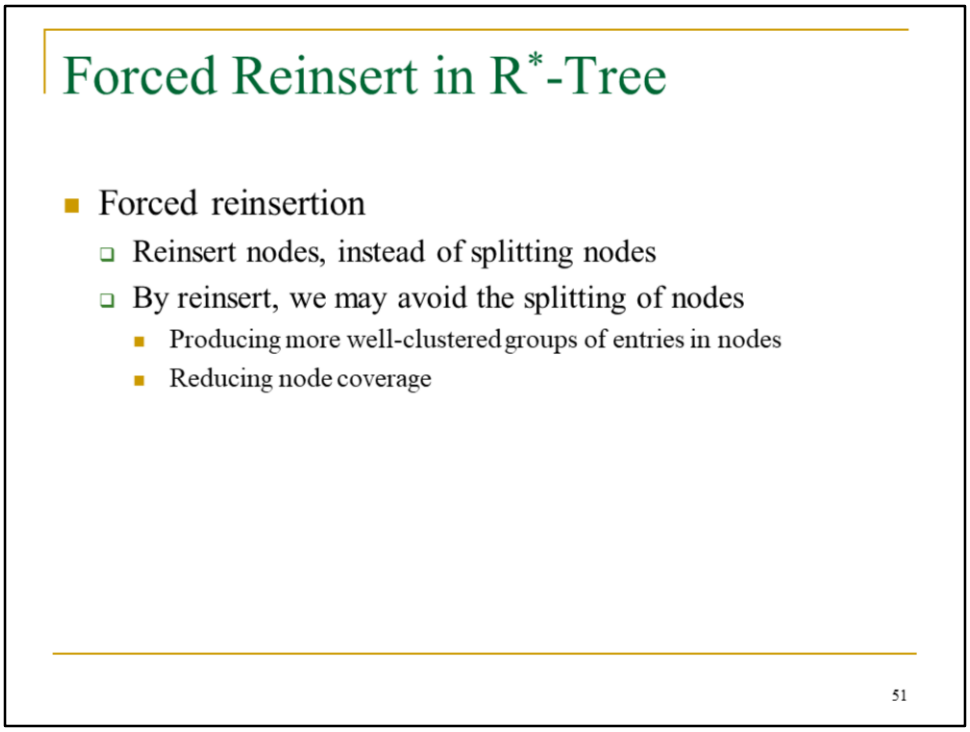
Specifically, for the insert algorithm, the ChooseSubtree function selects a branch among node entries to descend. If the node entries point to leaf nodes, then we select the entry with the least overlap enlargement and solve the tie by choosing the least area enlargement (if tie again, choosing the one with the smallest area). Otherwise, if node entries are not leaf nodes, we select the entry with the least area enlargement (or the smallest area in case of tie) to descend.



http://infolab.usc.edu/csci587/Fall2016/slides/session3-spatial-index-rtree.pdf

To handle overflow nodes, we first choose the split dimension, and then split the entries into two groups. For choosing the split dimension, ChooseSplitAxis algorithm sorts all entries by lower/upper bound values of MBRs and divide them into two groups. Then, we compute the sum of all margin values for different partitions, and select the axis with the smallest summed margin values. Here, small margin value means small perimeter of the MBR, which implies that the MBR is not a long rectangle.

For splitting entries into two groups, we choose the partitioning along the selected axis, such that the overlapping values are the smallest.



Similar to the deletion in the R-tree, when we encounter overflow leaf node, we can also apply re-insertion instead of splitting the node. By using the reinsertion, we may avoid the splitting of nodes, and produce more well-clustered groups, reducing the node coverage.