Spatial queries – matching candidates

- **Very slow for larger tables**: ***M x N* comparisons**, each possibly proportional to the **number of vertices of participating** geometries;
- Retrieval from **hard disk is much slower** than from **memory**.
- We need a **strategy to reduce search space** to **reduce search time.**

- **FILTER → REFINE:**
  - **Filter**: Coarse, approximate search – on disk
  - **Refine**: Detailed matching and checking for the relationship – in memory if possible (this may include checking all vertices in a geometry).

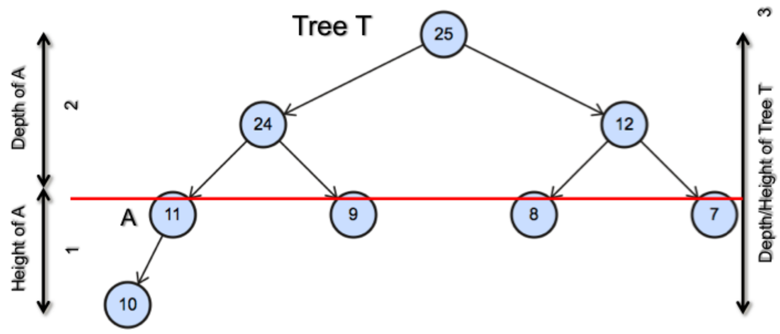
- **Such strategies are applied also in non-spatial contexts**

Indexes – retrieval complexity

- **Records with close values should be close together** in an index (close index values);
- In **clustered indices**, they are also **<u>stored</u> nearby** (speed up!)
- In tables, direct lookup (sequential scan) is of linear complexity (proportional to the number of records, *O(n)*);
- Indexing allows to reduce this to non-linear time (e.g., *O(log(n))*).
- Order in which you specify indices on your composite index matters – search for `city|city_part|suburb` or `suburb|city_part|city` ?

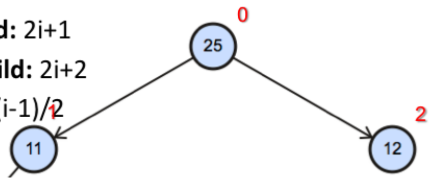
Tree data structures

Trees (cont.): terminology



- **Index of Left child:** 2i+1
- **Index of Right child:** 2i+2
- **Index of parent:** (i-1)/2

[*i* is index of THIS node]



- **Tree traversal**
  - Depth first search
  - Breadth first search

Indexing

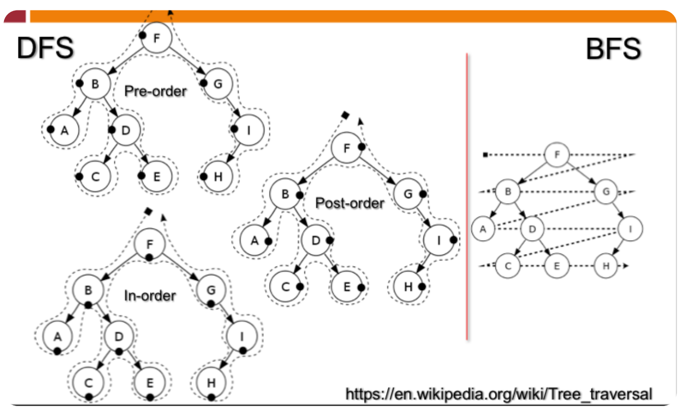
- **Index:** a data structure that improves the speed of retrieval of data matching a query;
- **Speeds up the identification of records** based on a given **attribute – to <u>not</u> inspect <u>all</u> records for <u>exact</u> value;**
- **Aim:** sub-linear time lookup <*O(n)*
- Imposes an **ordering on records** based on the values of an **attribute:**
  - this ordering can be **logical** only**,** or
  - **Clustered index**: if ordering reflected in storage as **physical** ordering (**only one per table**).
- **An index stores pointers to the page with the data item**
  - Key:value store of pageid.itemid, **sorted by whichever attribute of the tuple** (with some nuances, more to come)
- **Cons:**
  - Indices have to be maintained (re-built);
  - Indices take additional storage space;
  - In some cases, the index match may be very inaccurate (see later)

Index: sacrifice storage space, in favour of execution time.

- **Usage:**
  - **To enforce database constraints (e.g.,** UNIQUE, PRIMARY KEY**);**
  - **To improve lookup on any other column frequently used in** WHERE, ORDER BY**, and** JOIN **clauses**
  - **Multi-column indexes are also possible**

Trees (cont.): Properties

- **Full tree:** every node has 2 children
- **Complete tree:** all levels filled except possibly bottom row, filled from left to right



Trees: B^+-Tree

- B-Trees: family of trees - generalisation of the binary search tree enabling for more children (e.g., B^+Tree, B*Tree,BSP tree…)
- Order of tree: num children = *m<=M (M often called b)*
- Children can be leafs or internal nodes
- Root has at least one key, non-root nodes at least *M/2* subtrees
- All leaves are sorted, map to disk space (page)
- All leaves are at the same level

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b=7$	Example $b=100$
Root Node (when it is the only node in the tree)	Records	1	$b-1$	1–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil$	b	4–7	50–100

In the following, assume

- M branching factor (max number of children per node, also often *b*)
- Consider tree of order 4 {M=4}
- Root node should only have one entry
- Every non-leaf node should have at least m children nodes: M/2 <*m* entries per node in leaves <= M
- M-1 search key values in internal nodes
- Will be a balanced tree
- The leaf pages must store enough records to remain at least half full

Trees: B^+-Tree

- **B Tree: Keys and records both can be stored in the internal as well as leaf nodes.**
- **B+ tree: extension of the idea. Records (data) can only be stored on the leaf nodes, while internal nodes can only store the key values.**

B-Tree Properties

- A B-tree is completely balanced (path from root to leaf is constant) at all stages in its evolution
- Search time is bounded by the length of the path, and so is $O(\log n)$
- Insertion and deletion of records require $O(\log n)$ time
- Each node is guaranteed to be at least half full (or almost half full with odd fan-out ratios) at all stages in a B-tree's evolution

B+-trees, where pointers to records are only stored at leaf nodes, are more often used in practice

Spatial indexes

- Generalisation of the previous ideas into 2+ dimensions
- We need to sort things to order and number them in space somehow.
- We want close things to be close together in the index;
- Not natural way to sort x and y coordinate!

Spatial indexes - common approach

- We divide space or group objects based on some criteria;
- These spatial divisions or groups are somehow numbered (indexed);
- The numbering is meaningful, in order to support sequential search;
- Problem:
  - large objects, long objects, concave objects
- **Solution: hierarchical divisions**

- **Two approaches:**
  - Space-driven indexing: space is regularly partitioned into regions (cells) with an assigned index value).
  - Data (content) driven indexing
- **Problems:**
  - We have points, lines, irregular polygons
  - Sizes and spatial distribution of objects can vary hugely
  - Application type: frequent writes or frequent reads?
  - *Recent development: Is the DB partitioned*?
- **Compromises are necessary, know your priorities**
  - What kind of searches (range, exact,…)
  - How often we insert new objects;
  - How costly is the update of the index.

Space-driven spatial indexes

- We divide space into regular cells;
- These cells may be hierarchically organised;
- Cells may be always present (index slots always stored), or they may be generated based on the data inserted – but the scheme pre-defines the division.

Quadtrees: variants (Region, Point – KD tree)

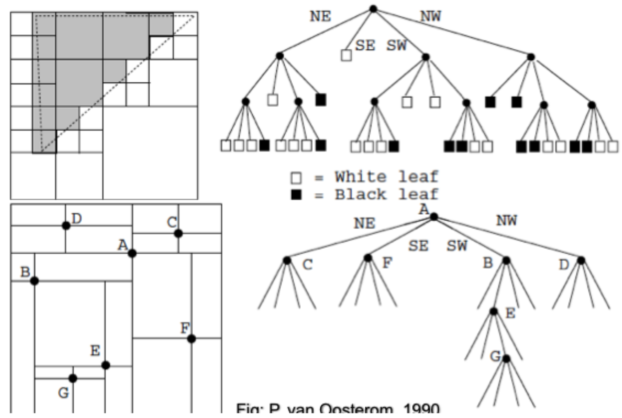


Fig: P. van Oosterom, 1990

Grid index

- We divide space into regular cells;
- Objects get and address in this space – either numbered, or as x and y index
- One object may be assigned to multiple cells
- Sorting is based on binary sorts (e.g., B-Tree)
- Numbering of cells following some logical order:
  - Row ordering (below, aka row-prime)
- Pros:
  - Simple application of 1D sorting;
  - Fast insert
- Cons:
  - Inefficient for storage size, may include large amounts of empty, pre-allocated space or redundancy
  - Depending on object distribution less efficient range search (object close in space may be far in index);

Quad tree index

- Divide space into hierarchically nested, adaptable cells;
- Each cell has a limit size (N of objects)
- The directory (the index array) follows a spatially motivated numbering scheme.
- Recursive (self-similar) numbering – space filling curves
- Pros:
  - Simple to index using modifications of standard 1D sorting;
  - Useful for raster indexing!
  - Adapt better to data (less wasted space)
  - Requires only local recomputing after insert/deletion
- Cons:
  - Depending on object distribution, may be less efficient for range search (object close in space may still be far in index);
  - Trees can be unbalanced in depth – hard to estimate retrieval time;
  - Some quadtrees depend on the order in which objects are inserted

Object driven indexes

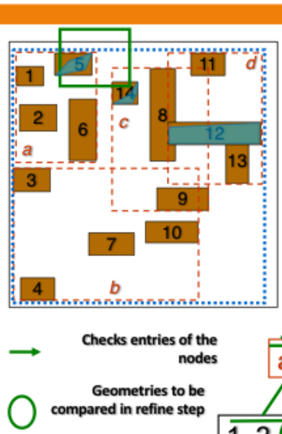
- Adapt to the distribution of objects (approximated by bounding shapes) in the plane/space;
- Do not fully cover space – only space occupied by spatial objects;
- Based on containment relationship, recursively
- Rely on a balanced hierarchical structure with direct mapping into a disk space (page)

R-Tree indexes

- Various refinements: R*Tree, R+Tree, STR-Tree…
- Main principle:
  - **Depth-balanced tree** (maps well to B+Tree) – all leaves on the same level;
  - Each node corresponds to a disk page
  - Each leaf node contains an array of leaf entries *[(mbb: o_id)]*;
  - Non-leaf nodes contain an array of node entries *[(mbb: node_id)]*;
  - Number of entries in node is ***m<x<M***, where ***m in [0,M/2]***

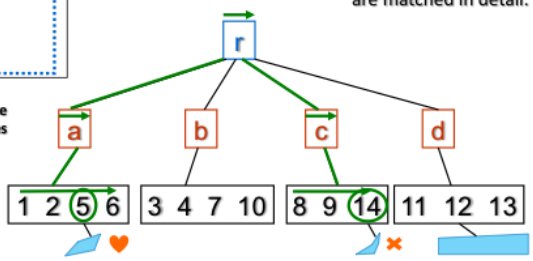
Filter-Refine data lookup

R-Tree indexes – Lookup/spatial search



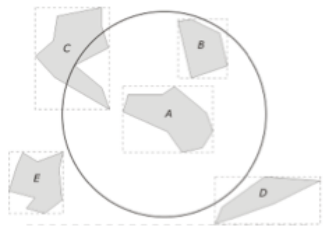
Compares the query window with the incidence of all the nodes' MBBs (directory or leaf) in the tree, hierarchically.

The result is a coarse filter - returns all the geometries with incident bounding boxes. Refinement happens in the second step, where all the geometries identified by their bounding boxes are matched in detail.



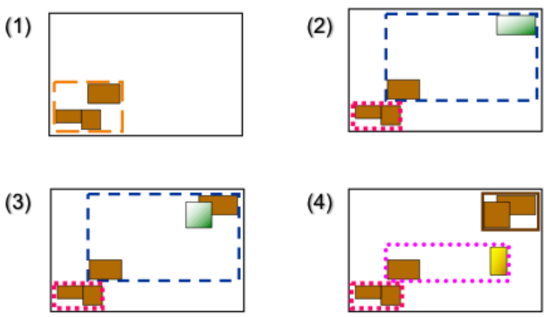
Rectangles and Minimum Bounding Boxes

- Minimum bounding box (MBB/MBR): the smallest rectangle bounding a shape with its axes parallel to the sides of the Cartesian frame
- Using MBB, some queries may be answered without retrieving the geometry of an object
- E.g., find all objects which lie entirely within a specified region



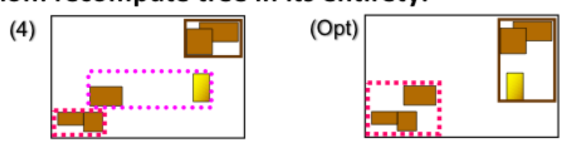
R-Tree indexes: Problem cases

R-Trees can lead, after many I/Os to some problem cases, deteriorating the quality of the tree:



R-Tree indexes: Problem cases

Solution: recompute tree in its entirety:

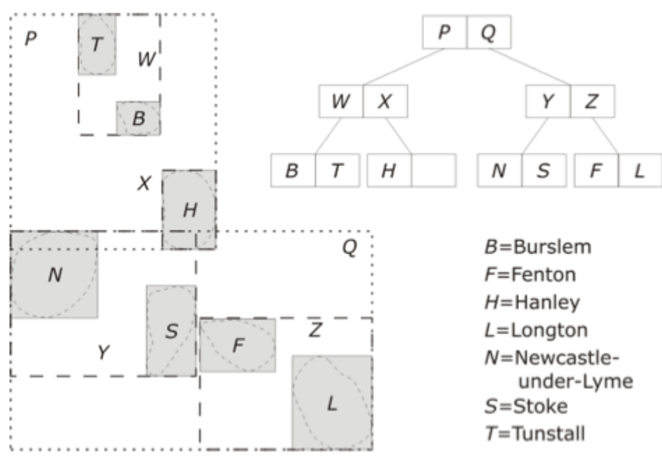


These cases are the focus of the R*Tree,R$^+$Tree and the SRT Tree

R-Tree

- Multidimensional dynamic spatial data structure similar to the B-tree
- Leaf nodes represent actual rectangles to be indexed
- Internal nodes represent smallest axes-parallel rectangle containing all descendents
- Rectangles at any level may overlap
- Good subdivisions:
  - Minimize the total area of containing rectangles
  - Minimize the total area of overlap of containing rectangles
- Overlap is critical: point and range searches are inefficient with large overlap (R+-tree aims to eliminate overlaps)

R-Tree



Ordered and Unordered Files

- In ***unordered files*** new records are inserted in the next physical location on the disk
  - Insertion is very efficient
  - Retrievals require search through every record in sequence: ***linear search*** with time complexity $O(n)$
  - Deletion causes "holes" to appear in sequence
- In ***ordered files*** each record is inserted in the order of the values of one or more of its fields
  - Slows the insertion of new records
  - Allows efficient ***binary search*** with time complexity $O(\log_2 n)$ on indexed field, but not on other fields

Two- Dimensional Ordering

From one to two dimensions

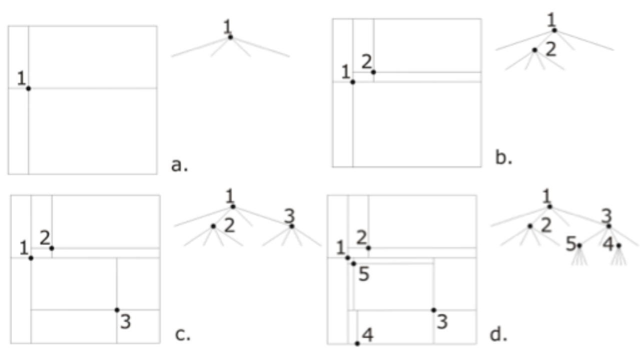
The main problem facing multidimensional spatial data structures is that data storage is essentially one-dimensional

- Many common indexes assume a grid-based representation (***tile indexes***)
- Tile indexes aim to provide a path through the grid that visits each cell
- Indexes differ in how well they preserve proximity, i.e., cells that are spatially close are close in the index

Summary

- Physical file organization affects database performance
- Indexes are needed to go beyond the limitations of physical file organization
- Non-spatial indexes, like B-trees, are inadequate for storing spatial data
- The key issue in spatial indexes is representing two dimensional data in a one-dimensional index

Point Quadtree



Spatial Queries

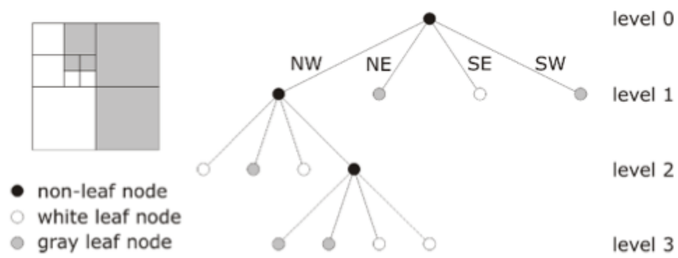
- ***Point query***: retrieve all records with spatial references located at a particular point
- ***Range query***: retrieve all records with spatial references located within a given range (spatial ranges may be any shape, but are often rectangular)

Example

- Non spatial query: Retrieve the point location of Trentham Gardens
- Spatial point query: Retrieve any site at location (37, 43)
- Spatial range query: Retrieve any site in the rectangle defined by (20, 20)–(40, 50)

Region Quadtrees

- Quadtrees take full advantage of the spatial structure, adapt to variable spatial detail
- Inefficient for highly inhomogeneous rasters
- Very sensitive to changes in the embedding space (e.g., translation, rotation)



Point Quadtree

- Combination of grid approach with multidimensional binary search tree
- Each non-leaf node has four descendents
- Each quadrant partition is centered on a data point
- Quadtree build time is $O(n \log n)$; search time is $O(\log n)$