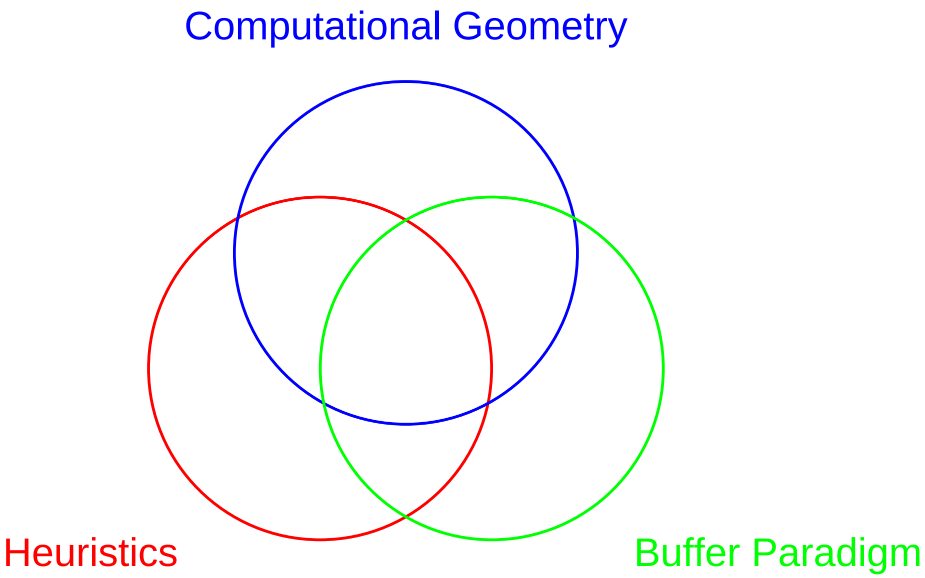
R-trees



Spatial Data

Spatial data: points, lines, polygons/polyhedra in $\mathbb{R}^d$.

For simplicity: assume $d = 2$.

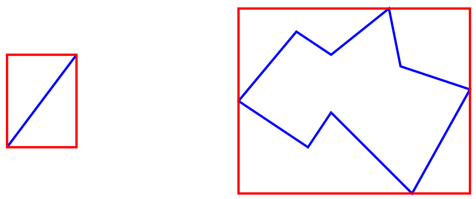
Typical queries:

Point:	All objects containing query point
Intersection:	All objects intersecting query object
Enclosure:	All objects containing query object
Spatial Join:	All intersections between stored objects and query set of objects

Example: windowing in GIS.

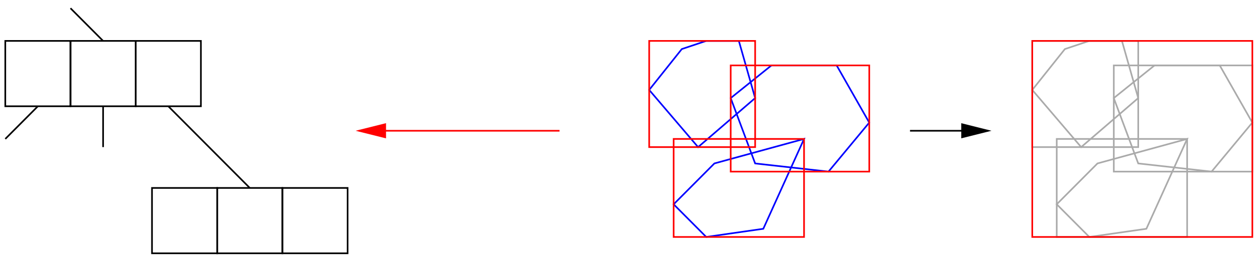
R-trees

Approximate objects by Bounding Box: [Guttman, 84]



R-tree = B-tree where

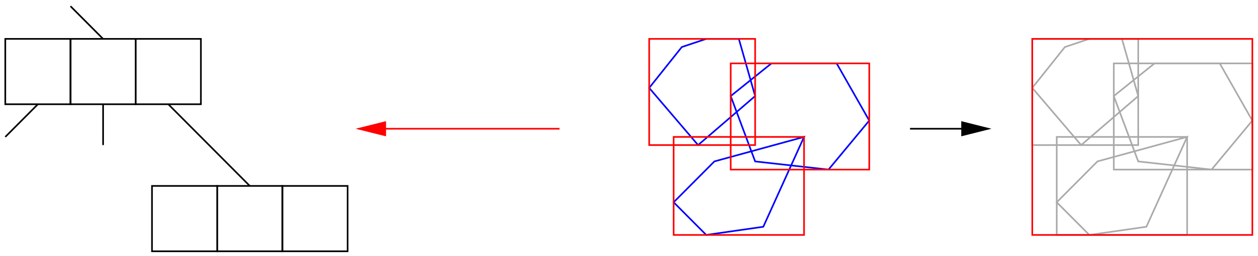
- Leaves store objects and their BBs.
- Internal nodes store for each child a BB containing all objects in that subtree.



Searching in R-trees

Example: Point Query

Search recursively in all subtrees of node where BB contains the query point.



Observe:

- Worst case query time: $\Theta(N)$!
- Average case performance depends heavily on distribution of objects in leaves.

Heuristic data structure.

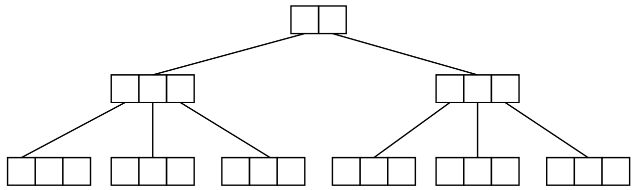
Static R-Trees

Build ("bulk load" in database community):

1. Distribute objects in leaves ($\Leftrightarrow$ assign linear order).
2. Build R-tree bottom up.

Number of I/O's:

1. $O(\mathrm{Sort}(N))$
2. $O(\mathrm{Scan}(N))$



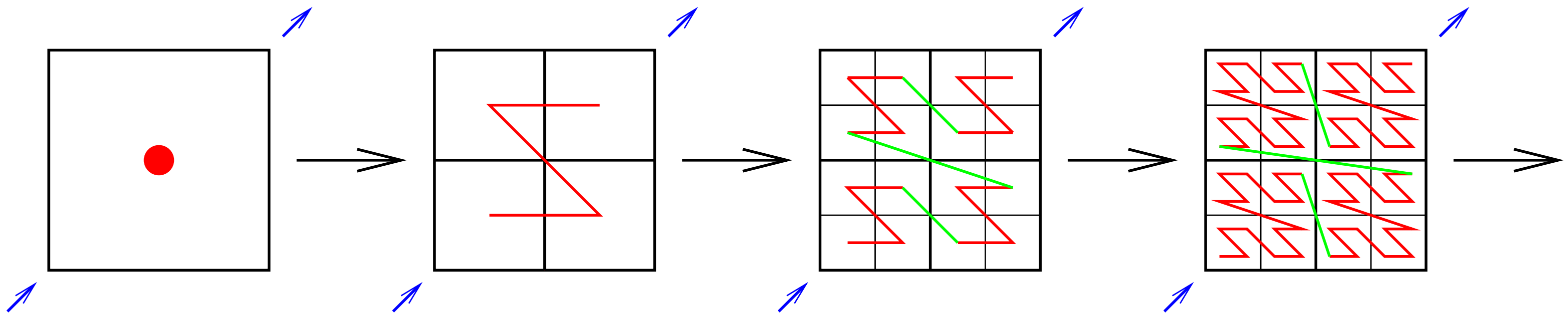
Query efficiency: depends heavily on the linear order.

Example: random distribution in leaves $\Rightarrow$ most leaves has BB close to BB of entire set of objects $\Rightarrow$ queries inefficient.

Goal: linear order where objects close in order $\Leftrightarrow$ close in Euclidean distance.

Space Filling Curves

The z-curve:



Comparing (x, y) and (x', y') having binary expansions $(x_1x_2\ldots x_k, y_1y_2\ldots y_k)$ and $(x'_1x'_2\ldots x'_k, y'_1y'_2\ldots y'_k)$:

$$\boxed{\text{Compare } y_1 \text{ and } y'_1\text{, then } x_1 \text{ and } x'_1\text{, then } y_2 \text{ and } y'_2\ldots}$$

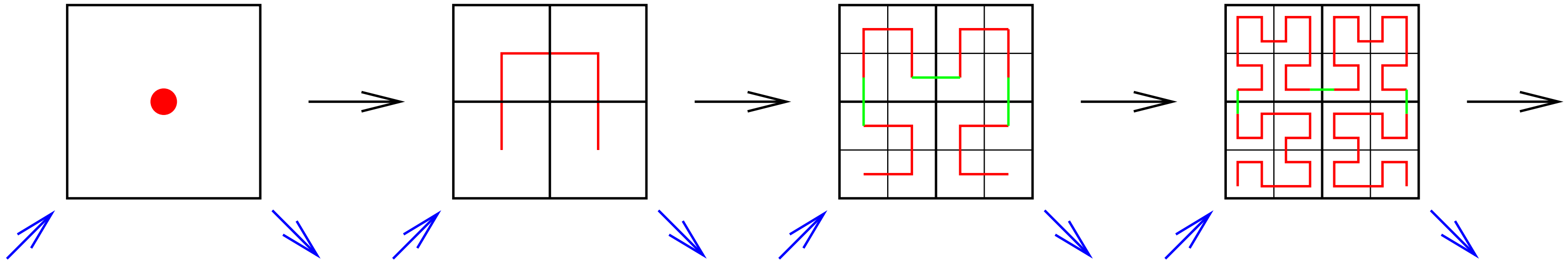
$$\Updownarrow$$

$$\boxed{\text{Compare } y_1x_1y_2x_2\ldots x_k \text{ and } y'_1x'_1y'_2x'_2\ldots x'_k}$$

"Bit interleaving"

Space Filling Curves

The Hilbert-curve:



Note: subpattern = pattern flipped along a diagonal. Generates four possibilities (as $\mathrm{flip}_1^2 = \mathrm{flip}_2^2 = \mathrm{ID}$ and as $\mathrm{flip}_1\mathrm{flip}_2 = \mathrm{flip}_2\mathrm{flip}_1$).

Comparing (x, y) and (x', y') having binary expansions $(x_1x_2 \ldots x_k, y_1y_2 \ldots y_k)$ and $(x'_1x'_2 \ldots x'_k, y'_1y'_2 \ldots y'_k)$:

Top level: Compare x_1 and x'_1, then y_1 and y'_1. Advance to next bit, choosing proper variant of this scheme.

Space Filling Curves and R-Trees

Choose ordering in leaves by sorting midpoints of bounding boxes of objects according to position on some space filling curve.

Empirically:

A static R-tree build this way has good query time in practice.

Dynamic R-trees

Insert:

We need two subroutines:

`Route()` Given a node v and a new object x, decide which subtree of v that x should belong to.

`Split()` Given a node v that is overflowing, decide how to split the subtrees in two groups.

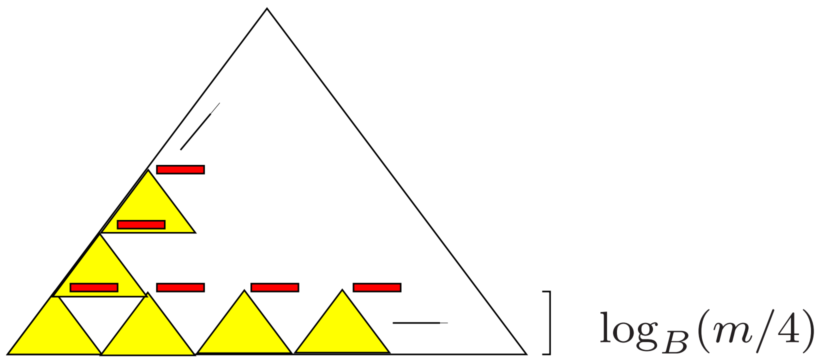
Various heuristics for these have been proposed, based on minimizing various properties of changed BBs (total area, area not used by objects, circumference, overlap with siblings BB, . . .).

Bottom line: $\exists$ fast heuristics giving good query times in practice.

Buffered R-Trees

[AHVV, 99]

Add buffers of size $\Theta(M)$ to layers $i \cdot \log_B(m/4)$, for $i = 1, 2, \ldots$.



Principle and analysis: exactly as for buffer trees (using `Route` and `Split`)

Advantage: Efficient bulk updates and queries. For $\Theta(N)$ size bulks:

Number of I/O's is $O(\frac{1}{B}\log_m(N))$ per insert instead of $\Theta(\log_B(N))$.

Buffered R-Trees

Similar statements for queries and deletions (see paper).

Note that spatial join is a bulk query.

Empirically: Faster bulk insertions than for previous proposals. Query time in resulting structure competitive with repeated insertions.