Indexation d'objets ayant une extension spatiale : R(*)-Tree et M-Tree

François-Xavier Pineau[1]

[1]CDS, Observatoire Astronomique de Strasbourg

Réunion Infusion du 24 mai 2013

Requirements and current solutions

- Simple x-match : k-nn queries and/or small cone searches
  - Solution : kd-tree (generalization of binary-search in more than 1D)
    - pros : very fast, can be very simple (one array!)
    - cons : hardly support updates ($\Rightarrow$ not used in DBMS)
- Error-based x-match : variable size cone-searches
  - Solution : kd-tree, remove outliers by box-plot, upper limit on cone radius
    - pros : very fast, few changes
    - cons : not exact, not symmetric, if heterogeneous errors?
- Heterogeneous errors + extended objects x-match
  - Solution 1 : R-tree, R*-tree
    - cons : 3D boxes not really adapted for geom on the sphere?
  - Solution 2 : M-tree
    - pros : our shapes are circles and ellipses, naturally deals with geom on the sphere!
- Heterogeneous errors + extended objects + proper motion x-match
  - Solution 1 : TPR-tree, TPR*-tree
    - cons : same as R-tree, R*-tree
  - Solution 2 : modified M-tree with $r(t)$
    - pros : same as M-tree

The R-tree original paper by Antonin Guttmani (1984)

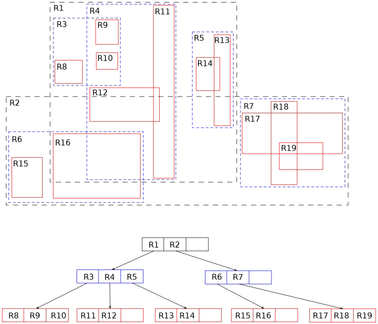
- "R-trees: A Dynamic Index Structure for Spatial Searching"

Principle

- Object extension approximated by a covering rectangle (CR)
- Only leaves contain objects
- Sub-tree's CR overlaps all the CR of its sub-elements

Range search

- Invoke range search on root
- if sub-tree is not a leaf: for each elem overlapping the range, invoke range search on it
- if sub-tree is a leaf: add overlapping elements to the result



Example of a 2 D R-tree. Crédits: wikipedia, from Fig. 3.1 of the original paper.

Building a R-tree

- When created, the tree root is a leaf
- Add a new entry:
  - Choose a leaf
    - choose the sub-tree whose CR needs least enlargement
    - resolve ties by choosing CR of smallest area
  - Add record to the leaf
    - if leaf contains empty space, add new entry
    - if leaf is full, split it, creating a new leaf
  - Propagate changes upward
    - adjust CR of the father
    - if previous split: add new sub-tree (can cause a node split)
    - move upward till root is reached
  - Grow tree taller (if node split propagation caused the root to split)
    - create a new root
    - add the two nodes resulting from the split

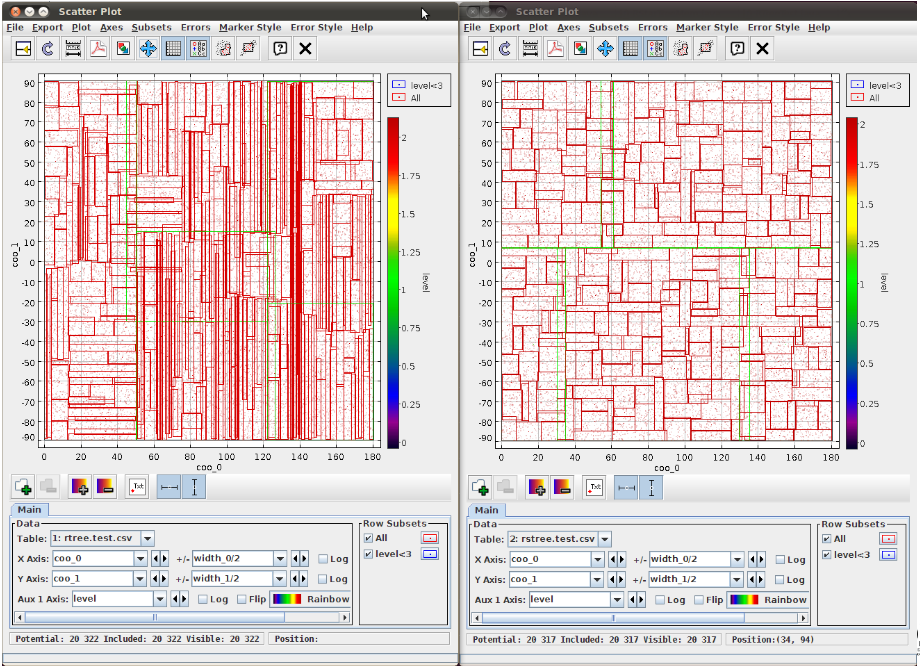
See on live animated schema on blackboard!

The R*-tree original paper by N. Beckmann et al. (1990)

- "R*-tree: An Efficient And Robust Access Method for Points and Rectangles"

Principle

- Same principle as the R-tree
- Same range search algorithm
- Almost same insertion algorithm
- Differences from R-tree insertion:
  - criteria when choosing the sub-tree (overlap enlargement, ...)
  - criteria when splitting a sub-tree (minimize overlap enlargement, ...)
  - when splitting, 30% of elements are re-inserted


Scatter Plot
File Export Plot Axes Subsets Errors Marker Style Error Style Help
level<3
All
coo_1
coo_0
level
Main
Data
Table: 1: rtree.test.csv
X Axis: coo_0 +/- width_0/2 Log
Y Axis: coo_1 +/- width_1/2 Log
Aux 1 Axis: level Log Flip Rainbow
Row Subsets
All
level<3
Potential: 20 322 Included: 20 322 Visible: 20 322
Position:
Scatter Plot
File Export Plot Axes Subsets Errors Marker Style Error Style Help
level<3
All
coo_1
coo_0
level
Main
Data
Table: 2: rstree.test.csv
X Axis: coo_0 +/- width_0/2 Log
Y Axis: coo_1 +/- width_1/2 Log
Aux 1 Axis: level Log Flip Rainbow
Row Subsets
All
level<3
Potential: 20 317 Included: 20 317 Visible: 20 317
Position:(34, 94)

Comparative benchmark

- Made on my Java implementation, fully in memory
  - not fully optimized! (fully debugged?)
  - no parallelization
- Benchmark setup
  - 200 000 2D rectangles
    - random center $(x, y) \in ([0, 180], [-90, 90])$
    - random extension $(dx, dy) \in ([0, 0.4], [0, 0.4])$
  - 10 000 query rectangles (leads to 9 931 results in this bench)

	build time (ms)	mean query time (ms)
R-tree	1 264	0.0345
R*-tree	37 517	0.0146

Remarks

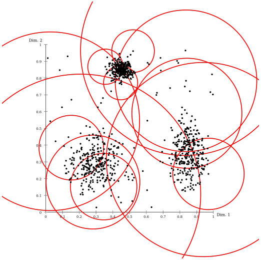
- Contrary to DBMS, no I/O operations here!
- Small tree, lots of queries ⇒ R*-tree
- Large tree, few queries ⇒ R-tree

The M-tree original paper by P.Ciaccia et al. (1997)

- "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces"

Principles

- Like the R-tree one, but CR replaced by covering balls (CB)
- Just need a distance function for the object it stores
- In each sub-tree, distance to the parent is stored
  - allow to save some computations
  - P: parent; C: child; Q: query; O= center; r = radius; d = distance function



Example of a 2 D M-tree. Crédits: wikipedia.

$$|d(O_P, O_Q) - d(O_P, O_C)| > r_C + r_Q \Rightarrow d(O_C, O_Q) > r_C + r_Q$$

$\Rightarrow$ no need to compute $d(O_C, O_q)$!
See schema on blackboard!

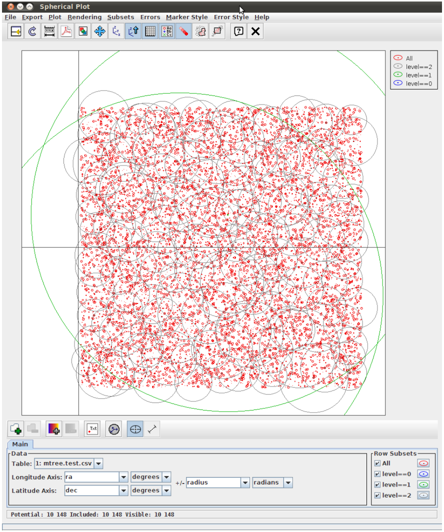
Benchmark (code V.1!)

- Fast multi-threading possible (trick: data stored following a z-curve)
- Bench characteristics:
  - Machine: cdsxmatch3
  - M-tree size: 10 000 000
  - Number of queries 1 000 000
  - $\bar{n}_{res}$: 6.4
- Results for 1 and 24 threads (T):
  - building time

Dist.	1 T	24 T	fac
Eucl.	65 s	3.5 s	x18
+ asin	237 s	11 s	x21
Haver.	736 s	32 s	x23

  - querying time (all queries)

Dist.	1 T	24 T	fac
Eucl.	103 s	3.4 s	x30
+ asin	309 s	6 s	x52
Haver.	463 s	10 s	x46



Example of an (α, δ) M-tree on a random distribution of sky coordinates.

Source code

- M-tree code put in a library
  - Almost done: refactoring, tests and debug to be done
  - Currently ≈ 2500 lines of code (Metrics ⇒ comments excluded)
  - 2 versions:
    - one storing point-like objects
    - one storing extended objects
  - Uses design patterns (e.g. COMPOSITE to manipulate node and leaf)
- R-tree + R*-tree ≈ 2200 lines of code (can be factorized!)

Perspectives

- M-tree can be used for fuzzy word search (Levenshtein distance).
  - first test show poor performances
    - computing Levenshtein distance is time consuming!!
    - change split strategy?!
- Next implementation: M-tree with $r(t)$ to account for proper motions!!
- Long term: create index in a file (bulk-loading, ...)?