>>

Multi-dimensional Search Trees

❖ Multi-dimensional Tree Indexes

Over the last 20 years, from a range of problem areas

- different multi-d tree index schemes have been proposed
- varying primarily in how they partition tuple-space

Consider three popular schemes: kd-trees, Quad-trees, R-trees.

Example data for multi-d trees is based on the following relation:

```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

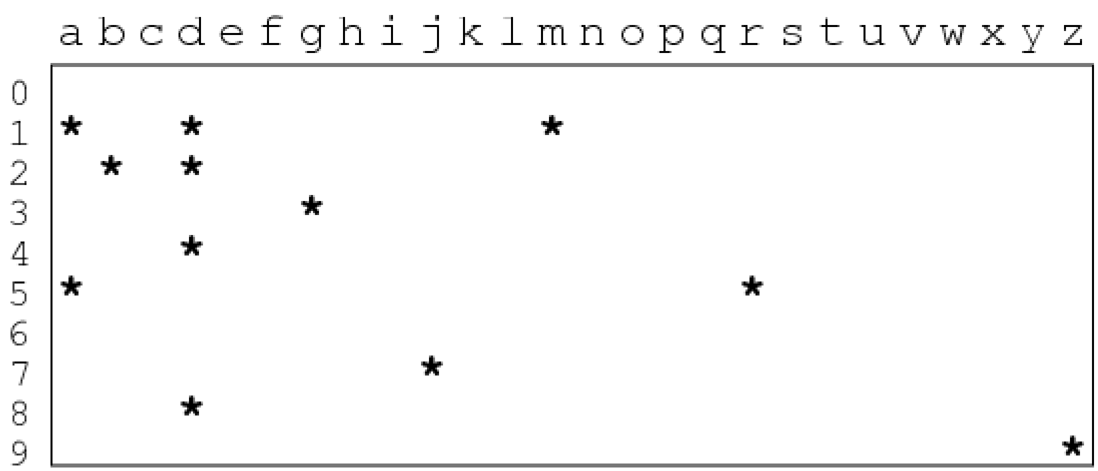
<< ∧ >>

❖ Multi-dimensional Tree Indexes (cont)

Example tuples:

```
R('a',1)  R('a',5)  R('b',2)  R('d',1)
R('d',2)  R('d',4)  R('d',8)  R('g',3)
R('j',7)  R('m',1)  R('r',5)  R('z',9)
```

The tuple-space for the above tuples:

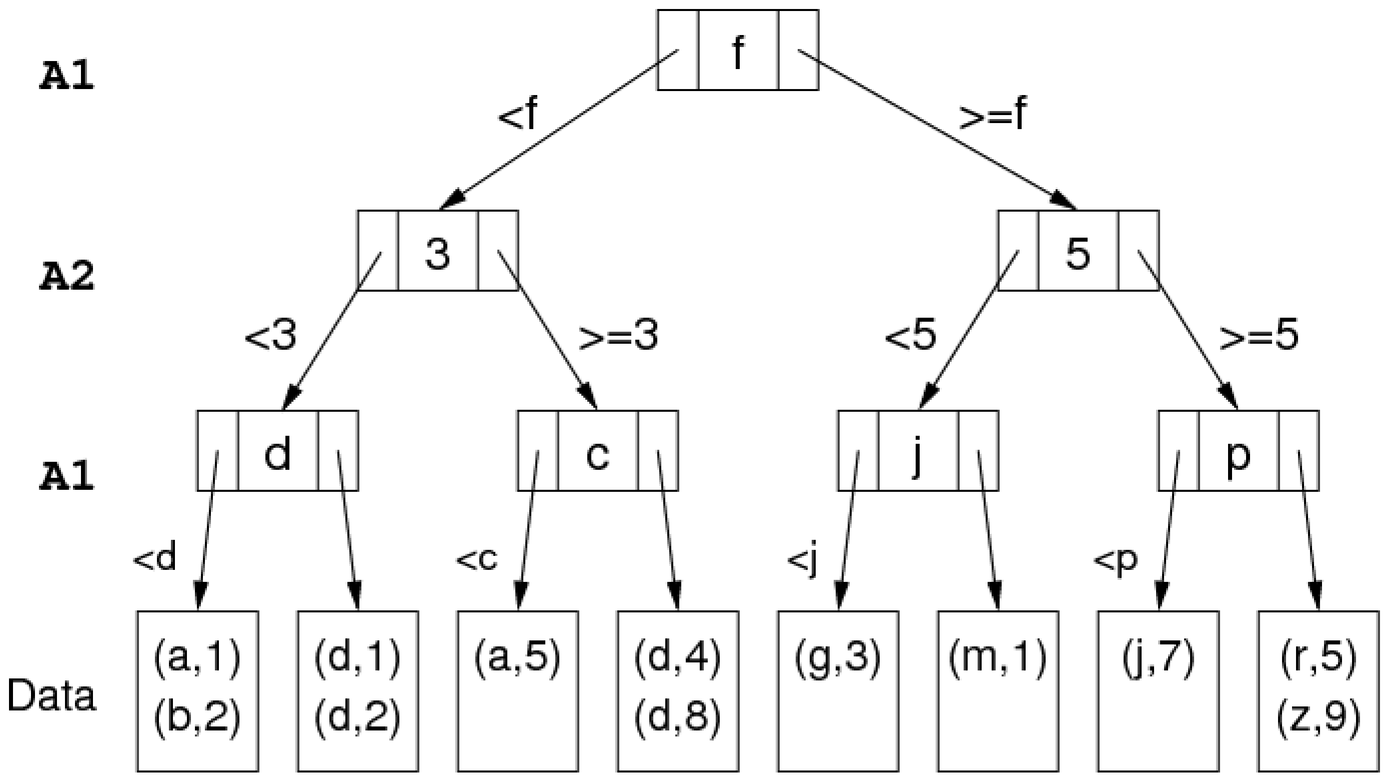


<< ∧ >>

❖ kd-Trees

kd-trees are multi-way search trees where

- each level of the tree partitions on a different attribute
- each node contains *n-1* key values, pointers to *n* subtrees

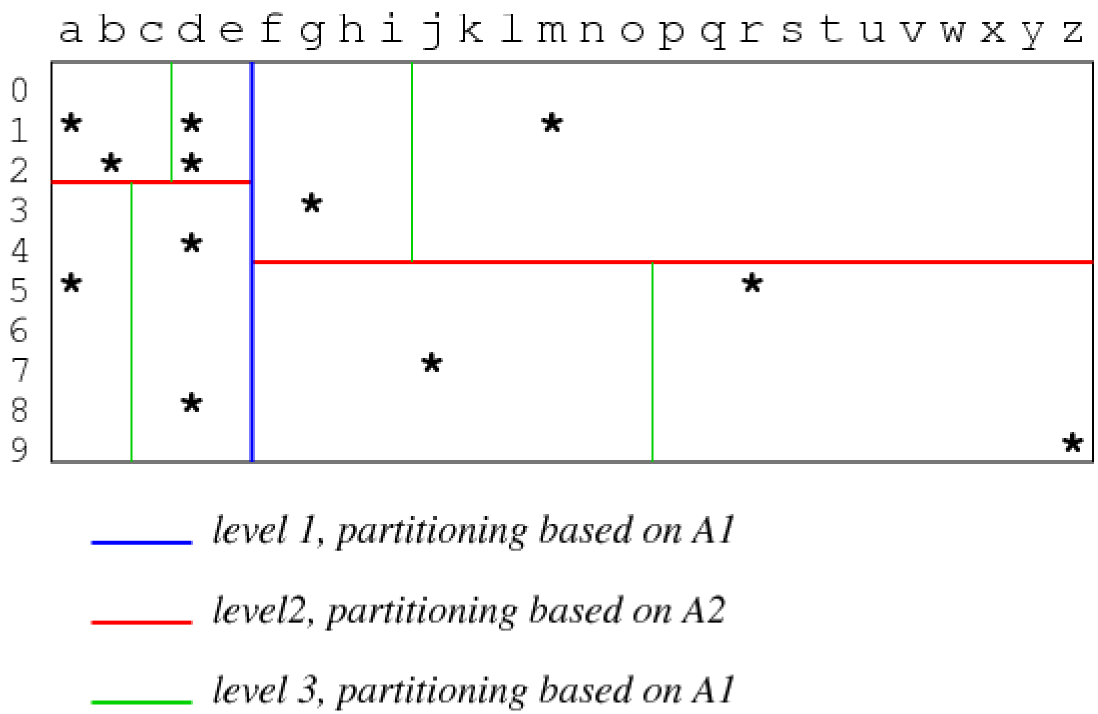


COMP9315 21T1 ♢ Nd Search Trees ♢ [3/16]

<< ∧ >>

❖ kd-Trees (cont)

How this tree partitions the tuple space:



<< ∧ >>

❖ Searching in kd-Trees

```
// Started by Search(Q, R, 0, kdTreeRoot)
Search(Query Q, Relation R, Level L, Node N)
{
   if (isDataPage(N)) {
      Buf = getPage(fileOf(R),idOf(N))
      check Buf for matching tuples
   } else {
      a = attrLev[L]
      if (!hasValue(Q,a))
         nextNodes = all children of N
      else {
         val = getAttr(Q,a)
         nextNodes = find(N,Q,a,val)
      }
      for each C in nextNodes
         Search(Q, R, L+1, C)
}  }
```

COMP9315 21T1 ♢ Nd Search Trees ♢ [5/16]

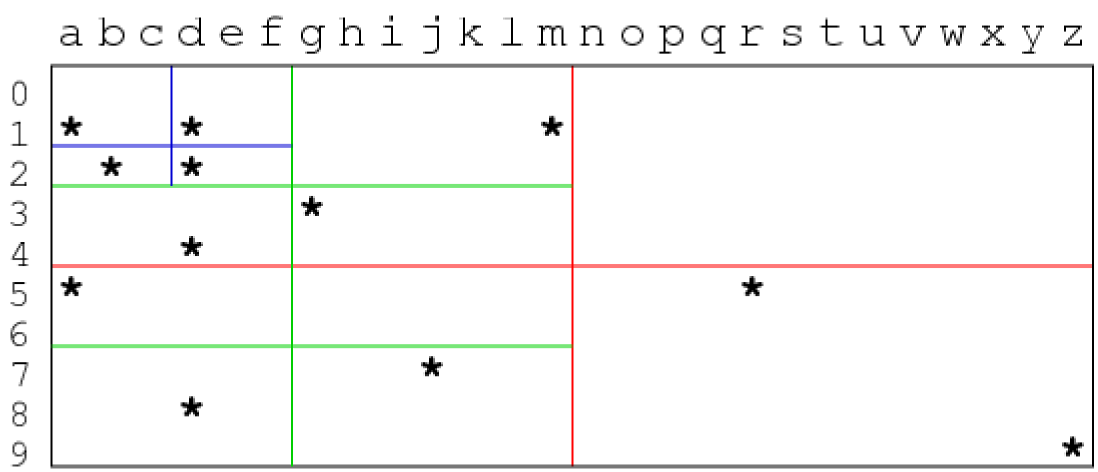
<< ^ >>

❖ Quad Trees

Quad trees use regular, disjoint partitioning of tuple space.

- for *2d*, partition space into quadrants (NW, NE, SW, SE)
- each quadrant can be further subdivided into four, etc.

Example:



<< ∧ >>

❖ Quad Trees (cont)

Basis for the partitioning:

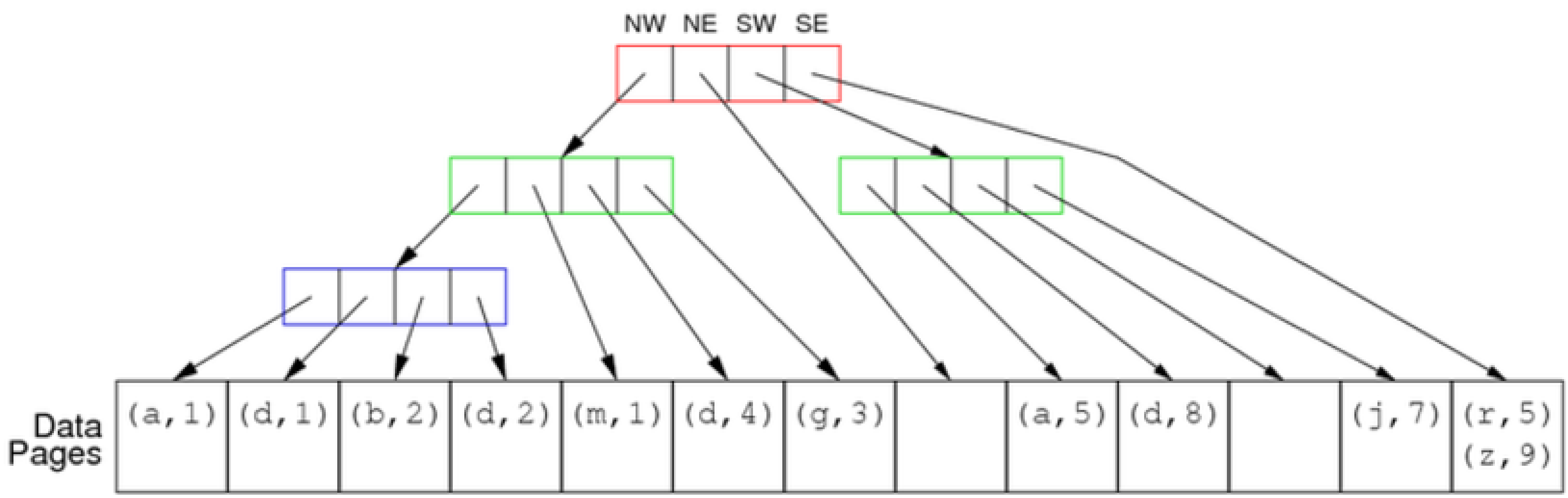
- a quadrant that has no sub-partitions is a leaf quadrant
- each leaf quadrant maps to a single data page
- subdivide until points in each quadrant fit into one data page
- ideal: same number of points in each leaf quadrant (balanced)
- point density varies over space
  ⇒ different regions require different levels of partitioning
- this means that the tree is not necessarily balanced

Note: effective for $d \leqslant 5$, ok for $6 \leqslant d \leqslant 10$, ineffective for $d > 10$

<< ∧ >>

❖ Quad Trees (cont)

The previous partitioning gives this tree structure, e.g.

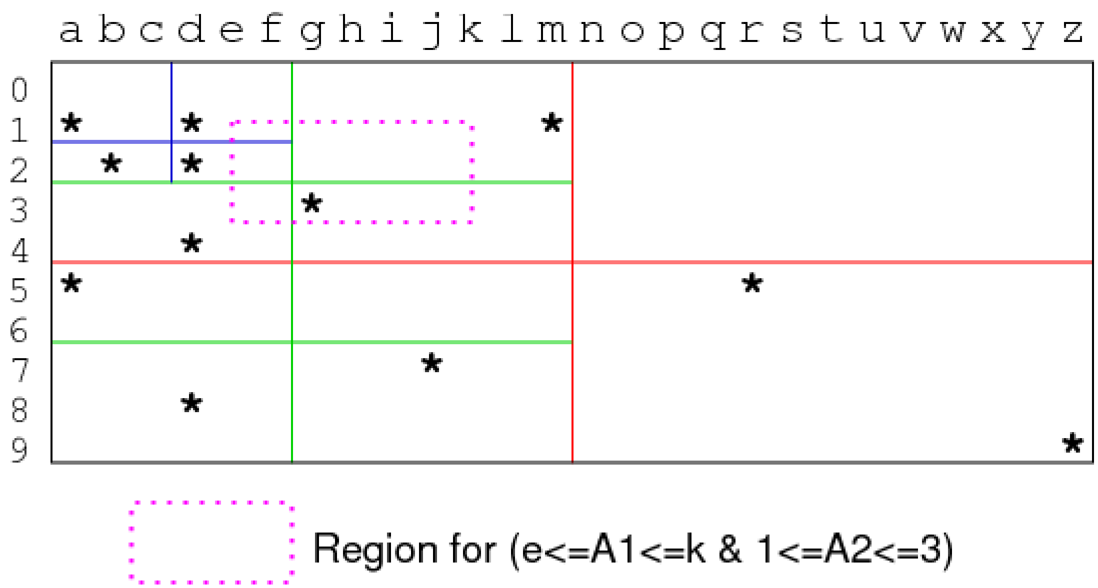


In this and following examples, we give coords of top-left,bottom-right of a region

<< ^ >>

❖ Searching in Quad-trees

Space query example:



Need to traverse: red(NW), green(NW,NE,SW,SE), blue(NE,SE).

❖ Searching in Quad-trees (cont)

Method for searching in Quad-tree:

- find all regions in current node that query overlaps with
- for each such region, check its node
  - if node is a leaf, check corresponding page for matches
  - else recursively repeat search from current node

Note that query region may be a single point.

<< /\ >>

❖ R-Trees

R-trees use a flexible, overlapping partitioning of tuple space.

- each node in the tree represents a *k*d hypercube
- its children represent (possibly overlapping) subregions
- the child regions do not need to cover the entire parent region

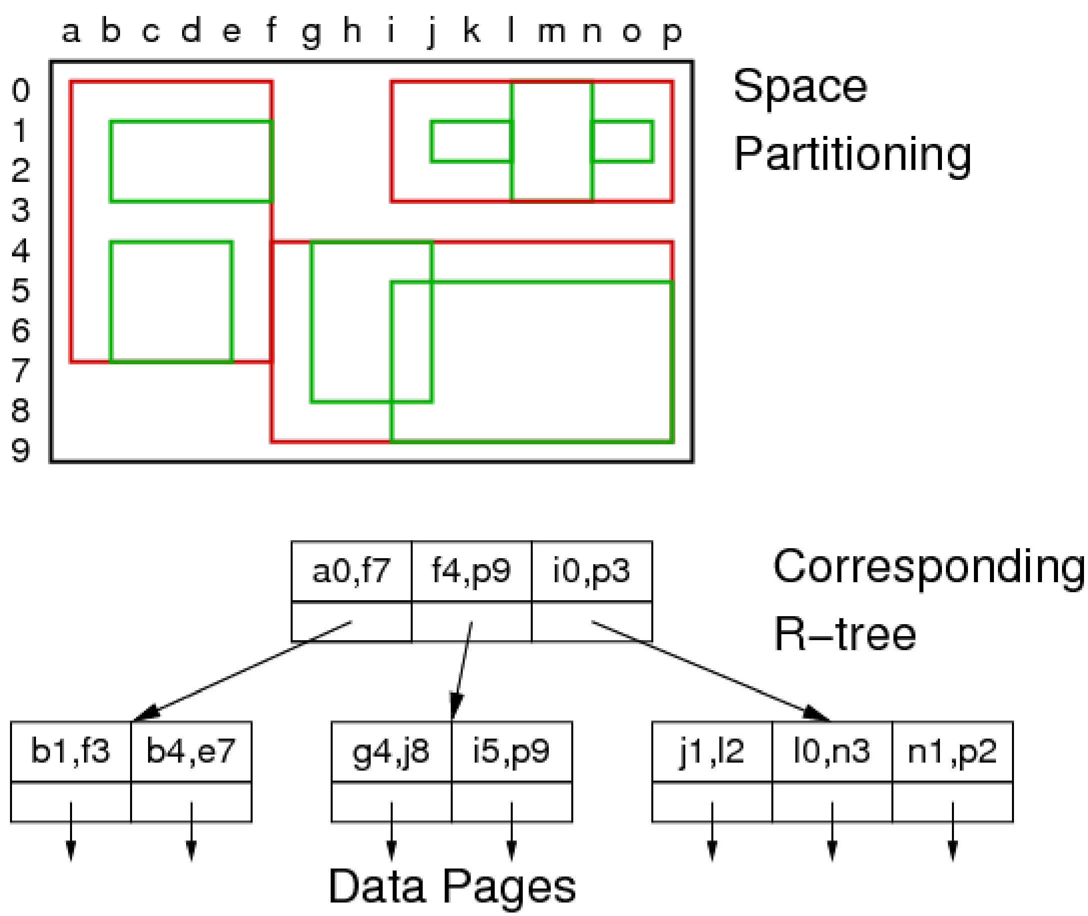
Overlap and partial cover means:

- can optimize space partitioning wrt data distribution
- so that there are similar numbers of points in each region

Aim: height-balanced, partly-full index pages (cf. B-tree)

<< ∧ >>

❖ R-Trees (cont)


a b c d e f g h i j k l m n o p
0
1
2
3
4
5
6
7
8
9
Space
Partitioning
a0,f7
f4,p9
i0,p3
Corresponding
R–tree
b1,f3
b4,e7
g4,j8
i5,p9
j1,l2
l0,n3
n1,p2
Data Pages

<< ∧ >>

❖ Insertion into R-tree

Insertion of an object *R* occurs as follows:

- start at root, look for children that completely contain *R*
- if no child completely contains *R*, choose one of the children and expand its boundaries so that it does contain *R*
- if several children contain *R*, choose one and proceed to child
- repeat above containment search in children of current node
- once we reach data page, insert *R* if there is room
- if no room in data page, replace by two data pages
- partition existing objects between two data pages
- update node pointing to data pages
  (may cause B-tree-like propagation of node changes up into tree)

Note that *R* may be a point or a polygon.

❖ Query with R-trees

Designed to handle *space* queries and "where-am-I" queries.

"Where-am-I" query: find all regions containing a given point *P*:

- start at root, select all children whose subregions contain *P*
- if there are zero such regions, search finishes with *P* not found
- otherwise, recursively search within node for each subregion
- once we reach a leaf, we know that region contains *P*

Space (region) queries are handled in a similar way

- we traverse down any path that intersects the query region

❖ Costs of Search in Multi-d Trees

Cost depends on type of query and tree structure

Best case: *pmr* query where all attributes have known values

- in kd-trees and quad-trees, follow single tree path
- cost is equal to depth *D* of tree
- in R-trees, may follow several paths (overlapping partitions)

Typical case: some attributes are unknown or defined by range

- need to visit multiple sub-trees
- how many depends on: range, choice-points in tree nodes

<< ^

❖ Multi-d Trees in PostgreSQL

Up to version 8.2, PostgreSQL had R-tree implementation

Superseded by GiST = Generalized Search Trees

GiST indexes parameterise: data type, searching, splitting

- via seven user-defined functions (e.g. **`picksplit()`**)

GiST trees have the following structural constraints:

- every node is at least fraction *f* full (e.g. 0.5)
- the root node has at least two children (unless also a leaf)
- all leaves appear at the same level

Details: Chapter 64 in PG Docs or src/backend/access/gist

Produced: 18 Mar 2021