Electronics and
Computer Science

UNIVERSITY OF
Southampton

Multidimensional Access Structures

COMP3211 Advanced Databases

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk
2016-2017

Overview

- Conventional indexes
- Hash-like
  - grid files, partitioned hashing
- Hierarchical indexes
  - multiple key, kd-trees, quad trees, r-trees, ub-trees
- Bitmap indexes

Multidimensional Access Structures

Indexes discussed so far are one-dimensional

- assume a single search key
- require a single linear order for keys (B-trees)
- require that the key be completely known for any lookup (hash tables)

Applications

Geographic information systems

- partial match queries
- range queries
- nearest-neighbour queries

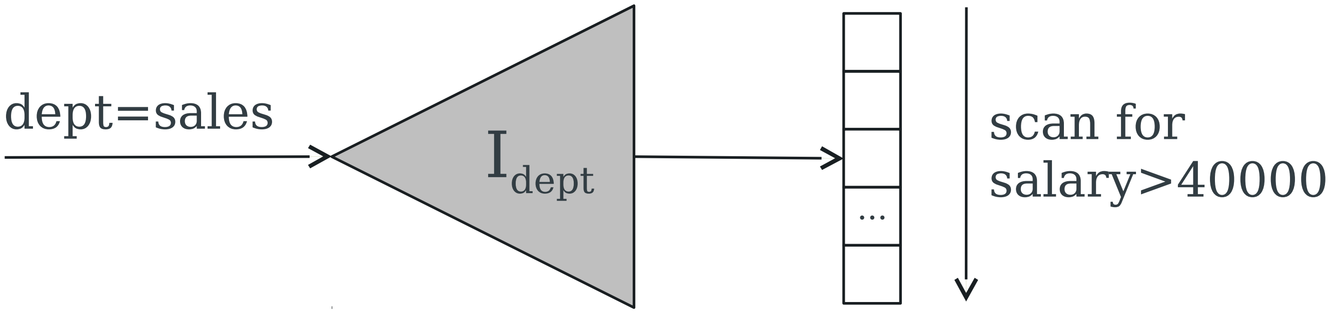
Conventional Indexes

Scenario

- Personnel database
- EMPLOYEE table with attributes
  - dept
  - salary
- How can we find employees who work in the sales department and have salaries greater than £40,000?

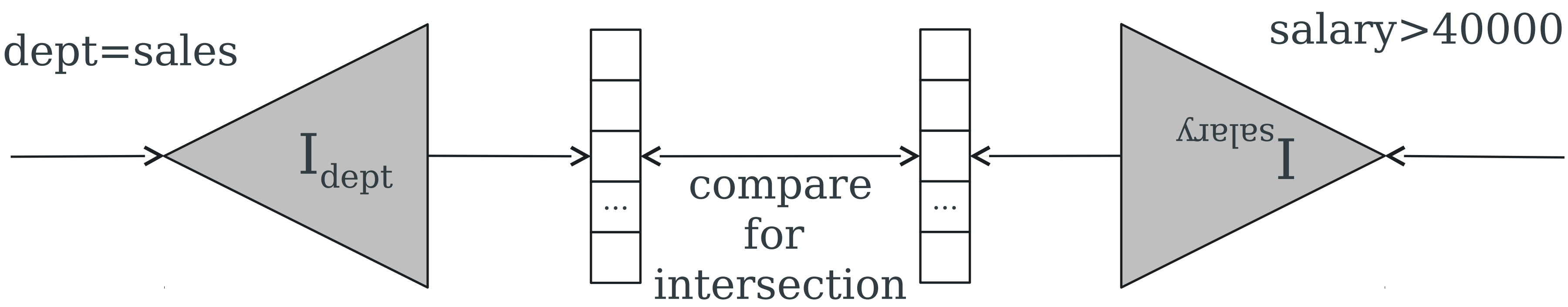
Approach #1

1. Get all matching records using an index on one attribute
2. Check values of other attribute on those records



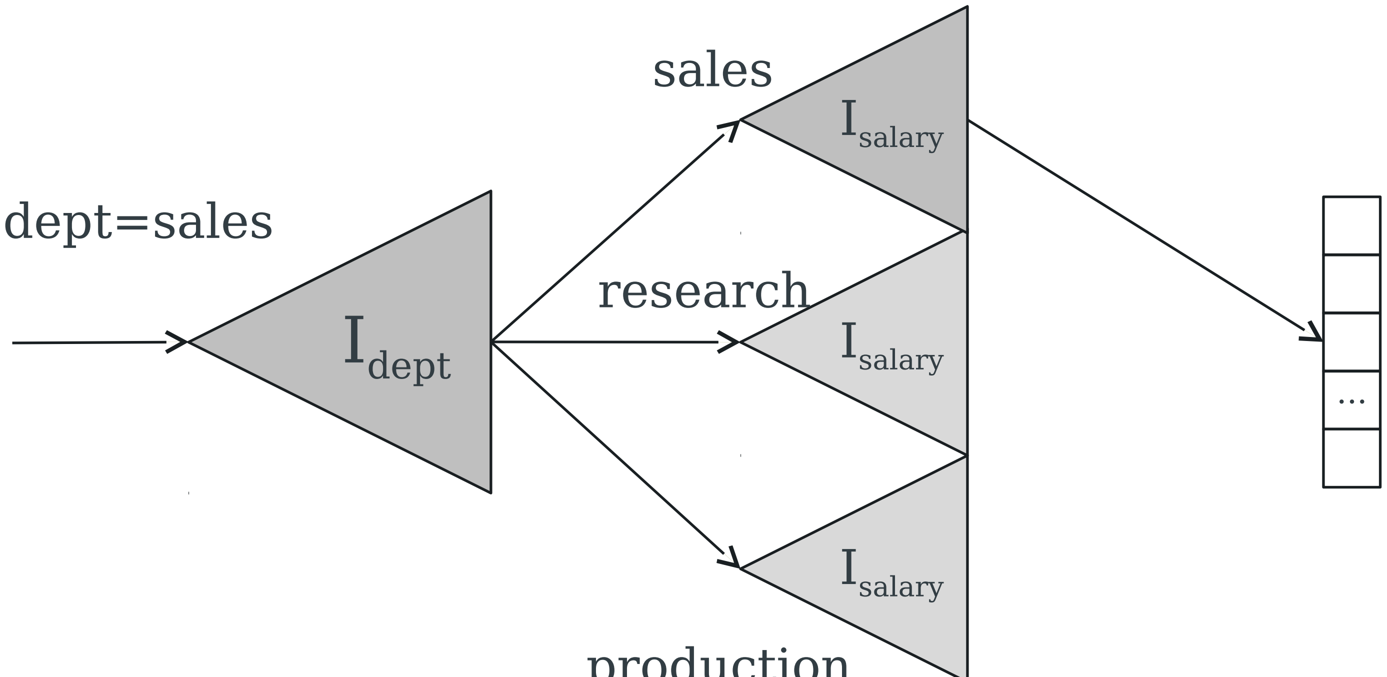
Approach #2

1. Use secondary indexes on each attribute to get two sets of record pointers
2. Take intersection of sets



Approach #3

1. Use secondary index on one attribute to select suitable index on other attribute
2. Get all matching records using selected index



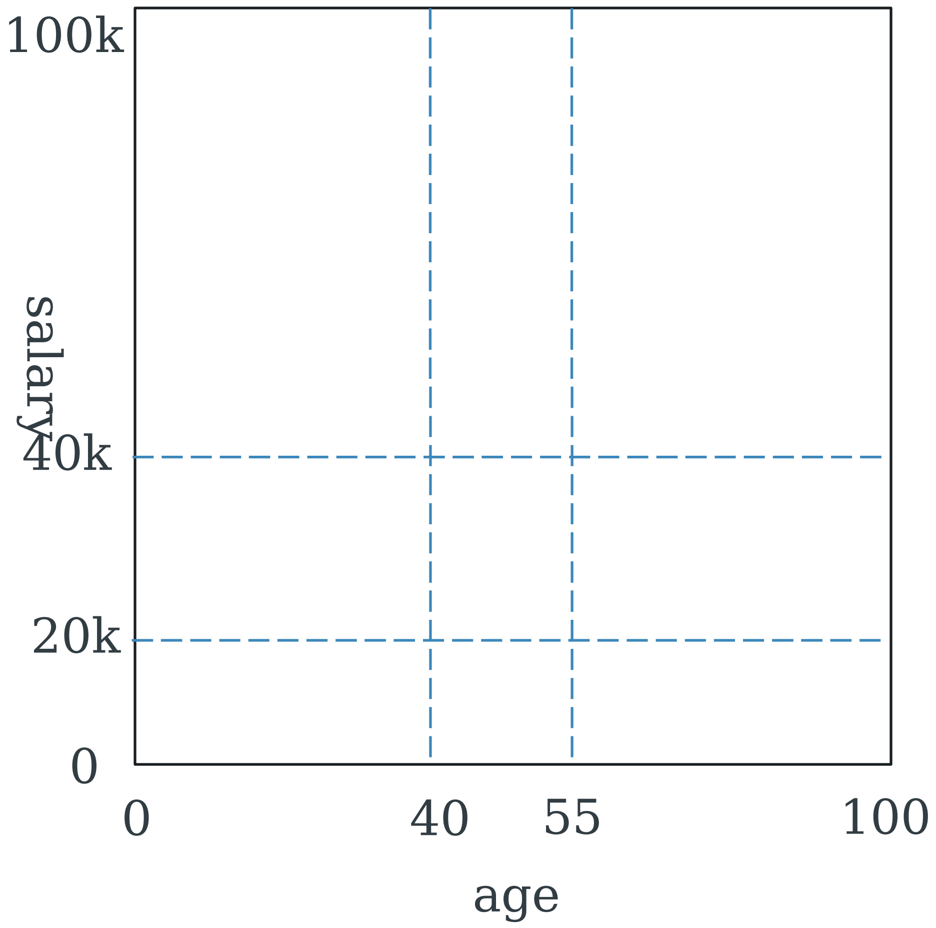
For which queries is this index good?

- $dept=sales \wedge salary=40000$
- $dept=sales \wedge salary>40000$
- $dept=sales$
- $salary = 40000$

Grid Files

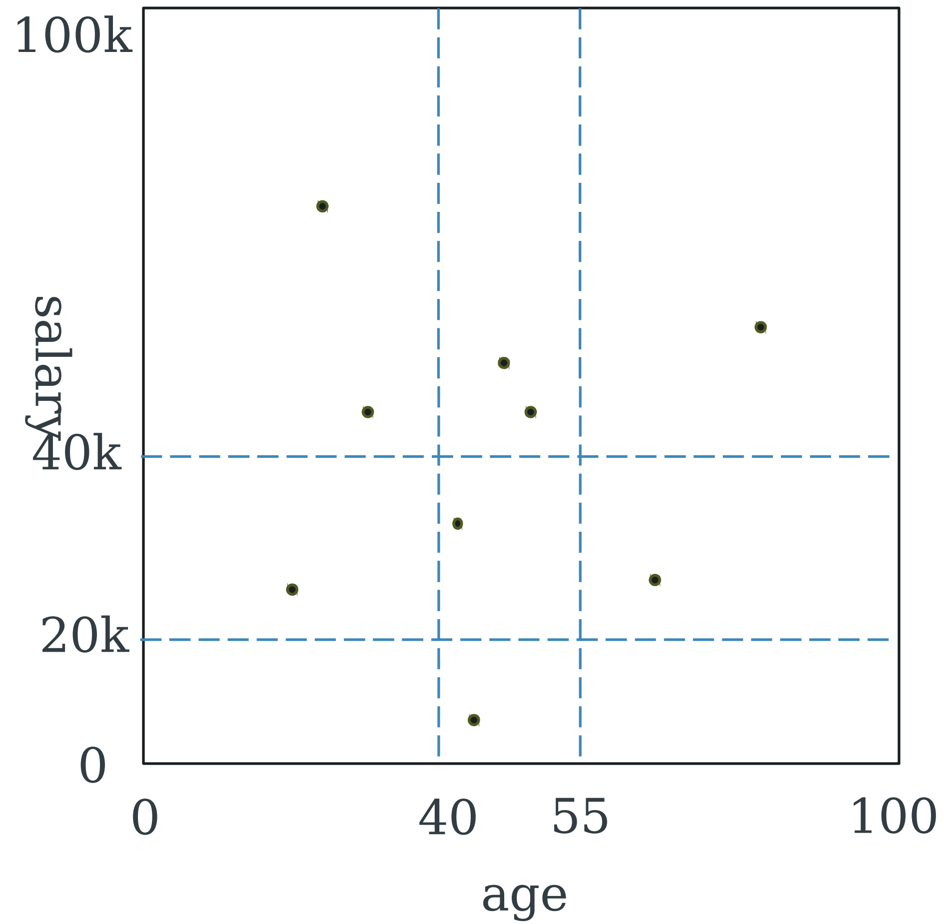
Grid File

- Partition multi-dimensional space with a grid
- Grid lines partition space into stripes
- Intersections of stripes from different dimensions define regions



Grid File

- Each region associated with a pointer to a bucket of record pointers
- Attribute values for record determine region and therefore bucket
- Fixed number of regions – overflow blocks used to increase bucket size as necessary
- Can index grid on value ranges



Grid files

Pro

- Good for multiple-key search
- Supports partial-match, range and nearest-neighbour queries

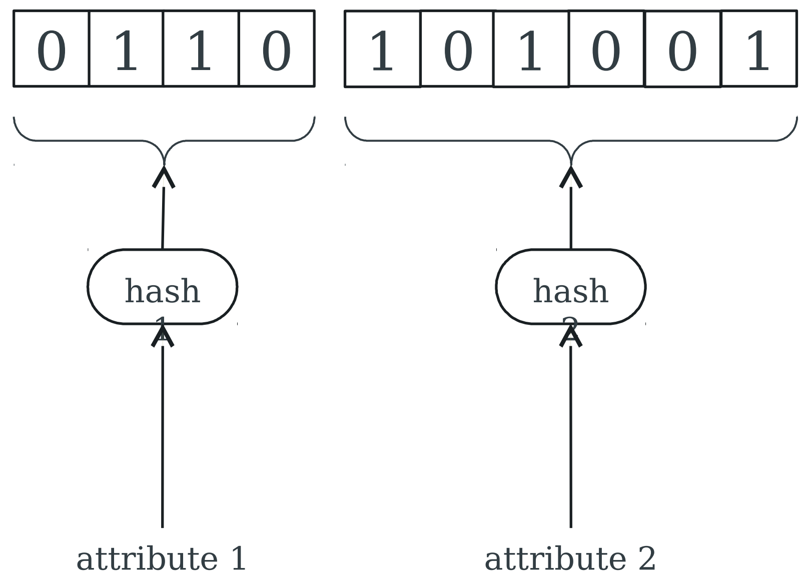
Con

- Space, management overhead (nothing is free)
- Need partitioning ranges that evenly split keys

Partitioned Hash

Partitioned Hash

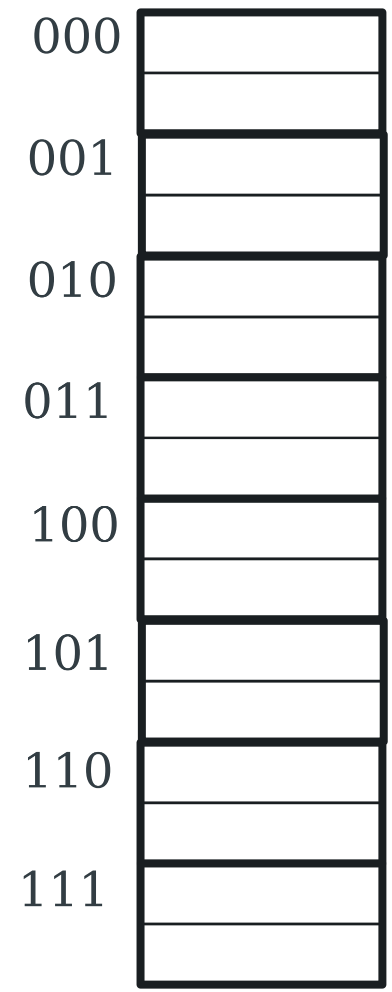
- Hash function takes a list of attribute values as arguments
- Bits of hash value divided between attributes
  - Effectively, a hash function per attribute



Example

hash1(sales) = 0
hash1(research) = 1

hash2(10000) = 00
hash2(20000) = 01
hash2(40000) = 10
hash2(100000) = 11

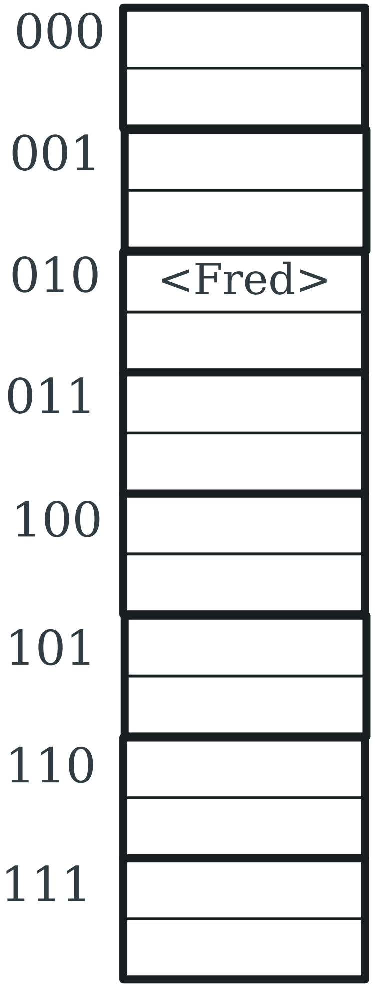


Insertion

hash1(sales) = 0
hash1(research) = 1

hash2(10000) = 00
hash2(20000) = 01
hash2(40000) = 10
hash2(100000) = 11

Fred works in sales
Fred’s salary is £40,000

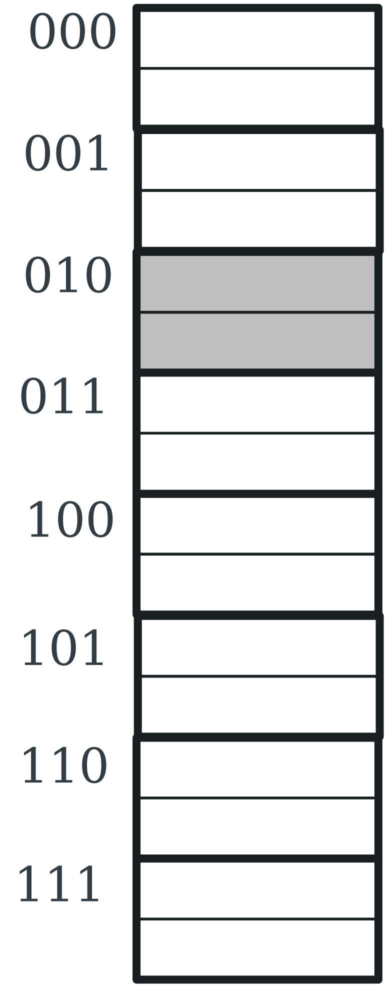


Retrieval

hash1(sales) = 0
hash1(research) = 1

hash2(10000) = 00
hash2(20000) = 01
hash2(40000) = 10
hash2(100000) = 11

dept=sales $\wedge$ salary=40000

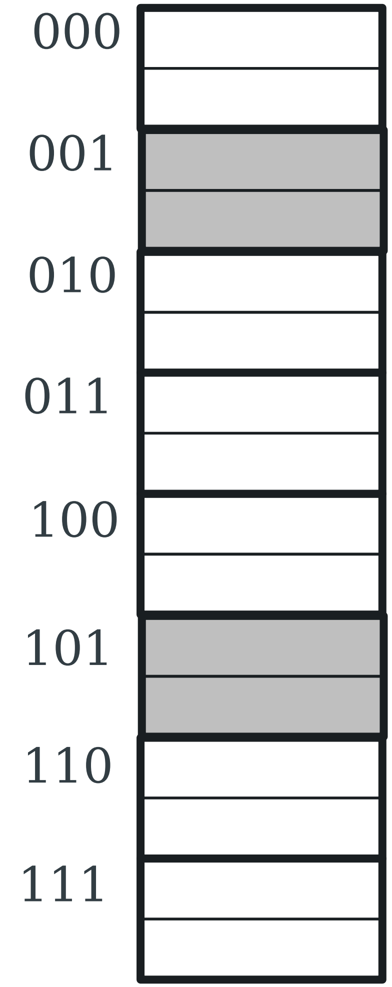


Retrieval

hash1(sales) = 0
hash1(research) = 1

hash2(10000) = 00
hash2(20000) = 01
hash2(40000) = 10
hash2(100000) = 11

salary=20000

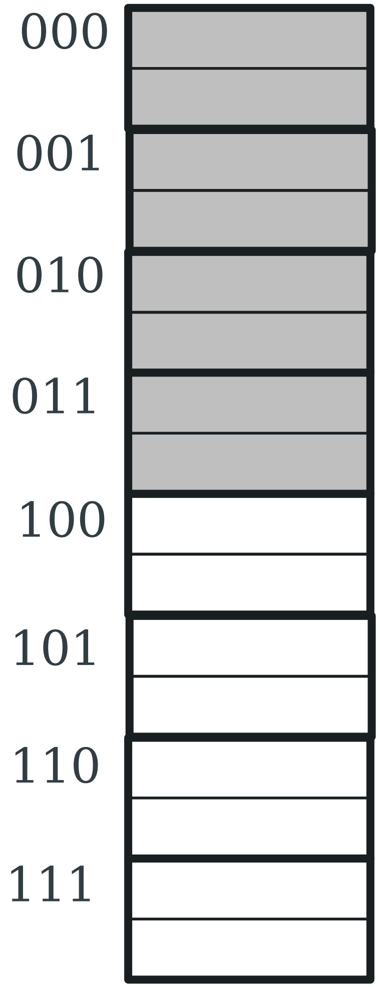


Retrieval

hash1(sales) = 0
hash1(research) = 1

hash2(10000) = 00
hash2(20000) = 01
hash2(40000) = 10
hash2(100000) = 11

dept=sales



Partitioned hash

Pro

- Good hash function will evenly distribute records between buckets
- Supports partial-match queries

Con

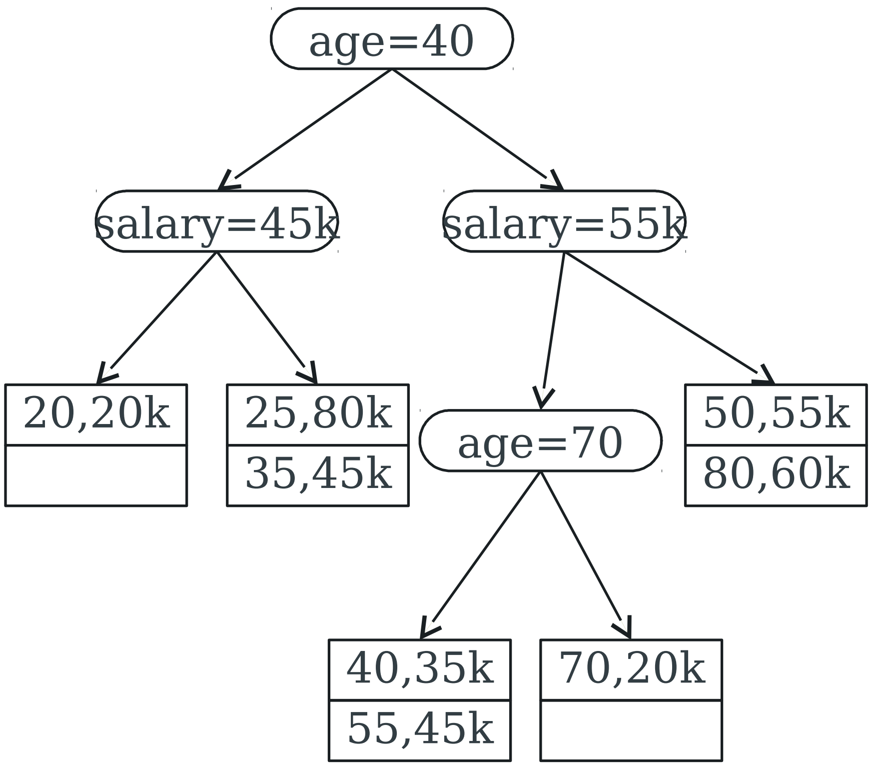
- No good for nearest-neighbour or range queries

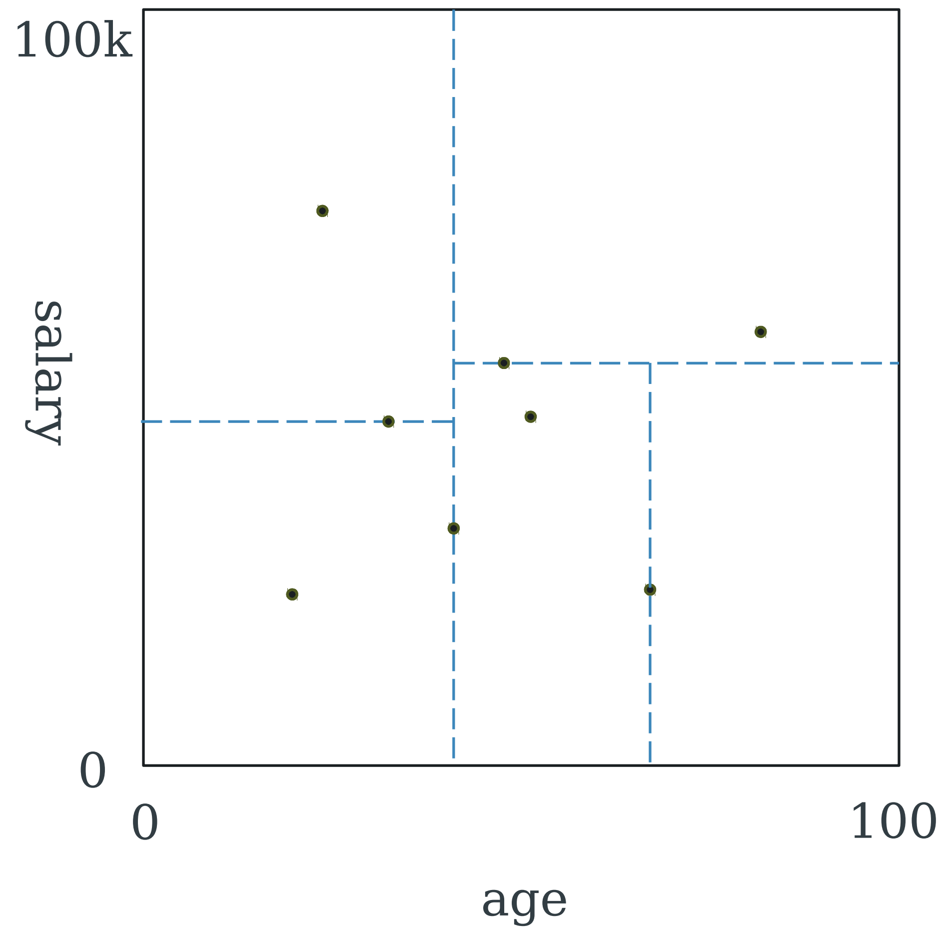
kd-Tree

kd-Tree

- Multidimensional binary search tree
- Each node splits the k-dimensional space along a hyperplane
- Nodes contain
  - an attribute-value pair
  - a pair of pointers
- All nodes at the same level discriminate for the same attribute
- Levels rotate between attributes of all dimensions

Example, k=2


age=40
salary=45k
salary=55k
20,20k
25,80k
35,45k
age=70
50,55k
80,60k
40,35k
55,45k
70,20k


100k
salary
0
0
100
age

Partial-Match Queries

- If we know value of attribute, we can choose which branch to explore
- If we don’t know value of attribute, must explore both branches

Adapting kd-Trees to Secondary Storage

Average path length from root to leaf: $\log_2 n$

Disk accesses should be kept as few as possible

Two approaches:

1. Multiway nodes (split values into n ranges)
2. Group nodes in blocks (node plus descendants to a given ply)

Quad-Tree

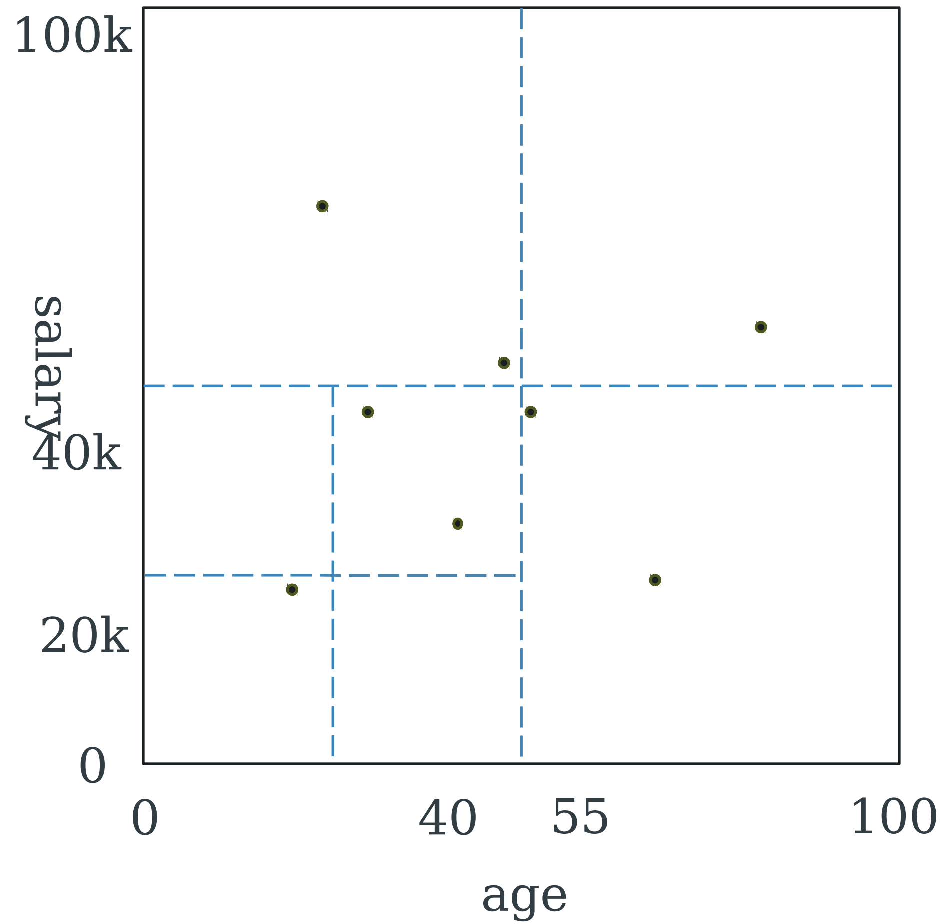
Quad-Trees

Two main types:

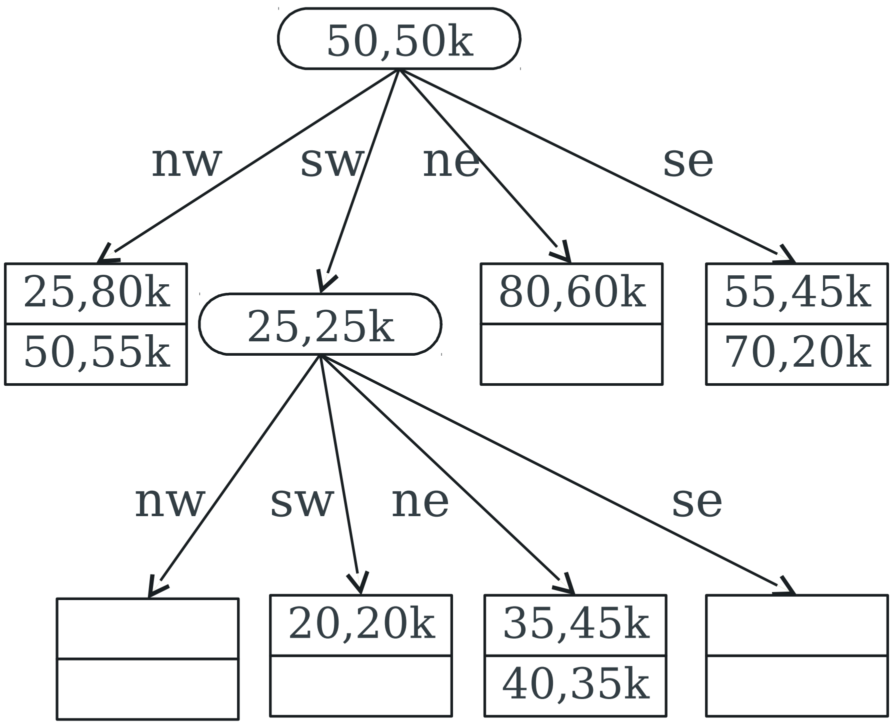
- Region quad-tree
- Point quad-tree

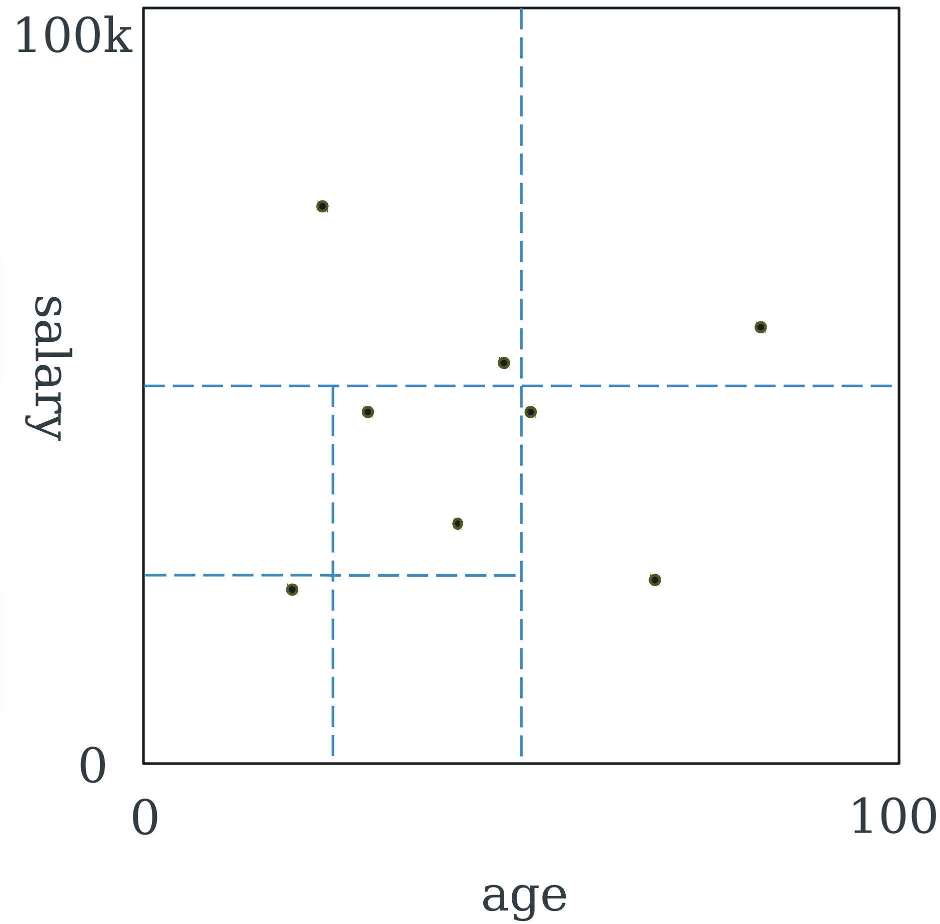
Region Quad-tree

- Each partition divides the space into four equal area sub-regions
  - ne, nw, se, sw
- Split regions if they contain more records than will fit into a block
- Operations similar to those for kd-trees



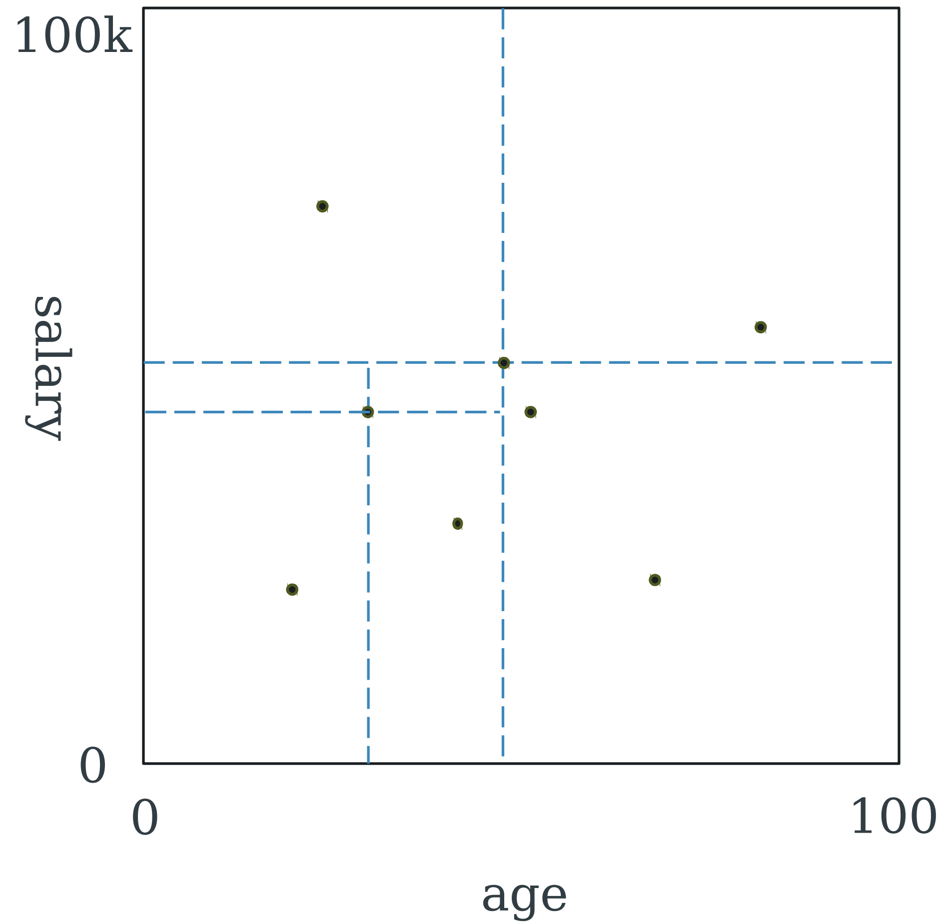
Region Quad-tree



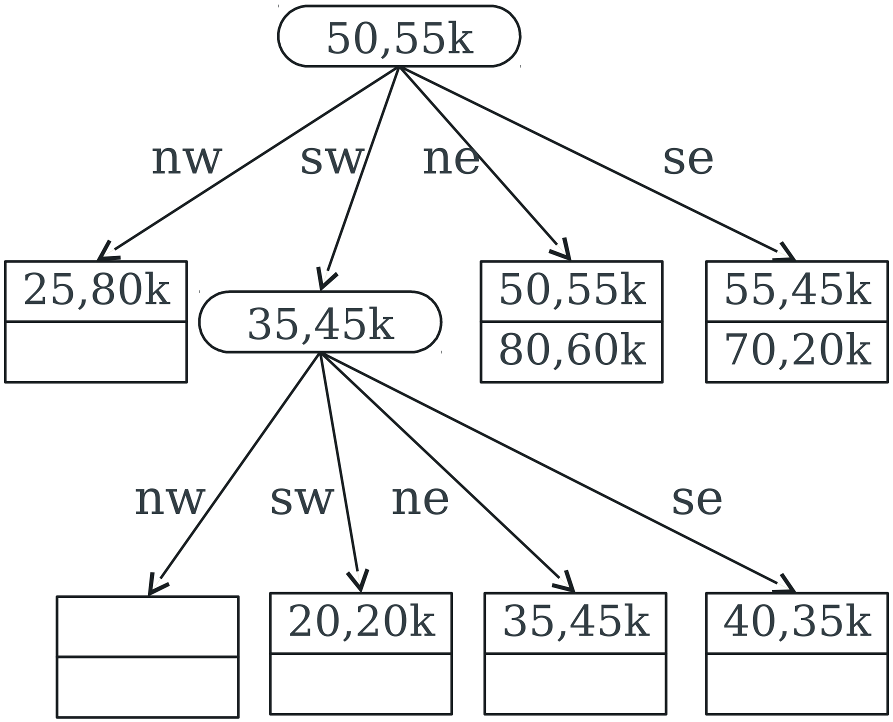


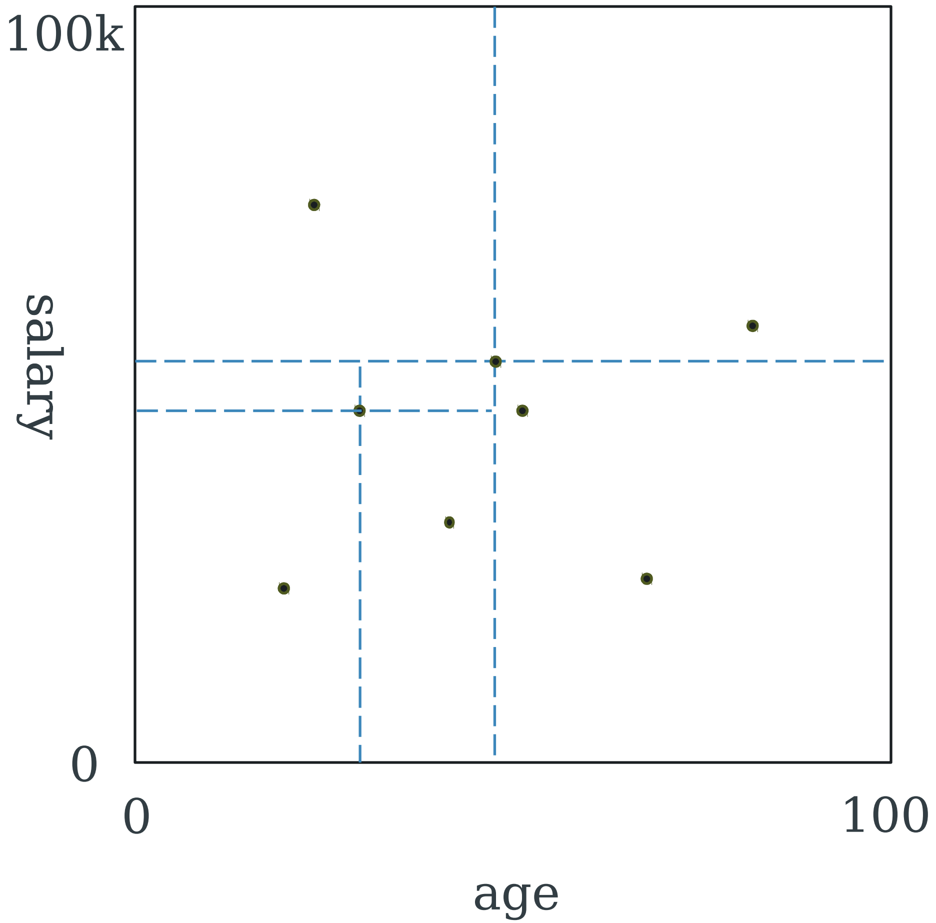
Point Quad-Tree

- Partitions are not equal area
  - Split lines centred on data points
  - ne/nw/se/sw sub-regions
- Otherwise, equivalent to region quad-tree



Point Quad-Tree

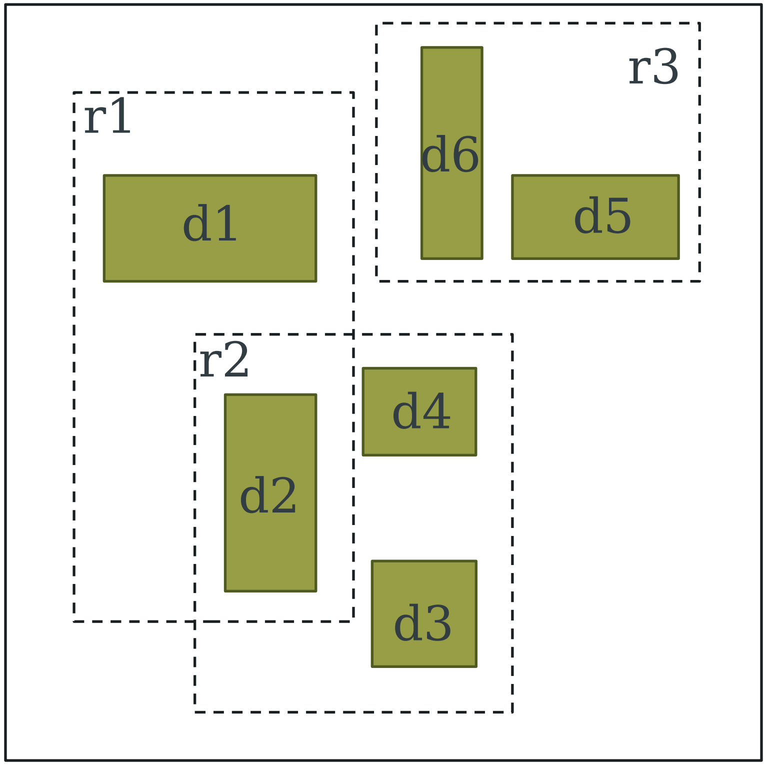




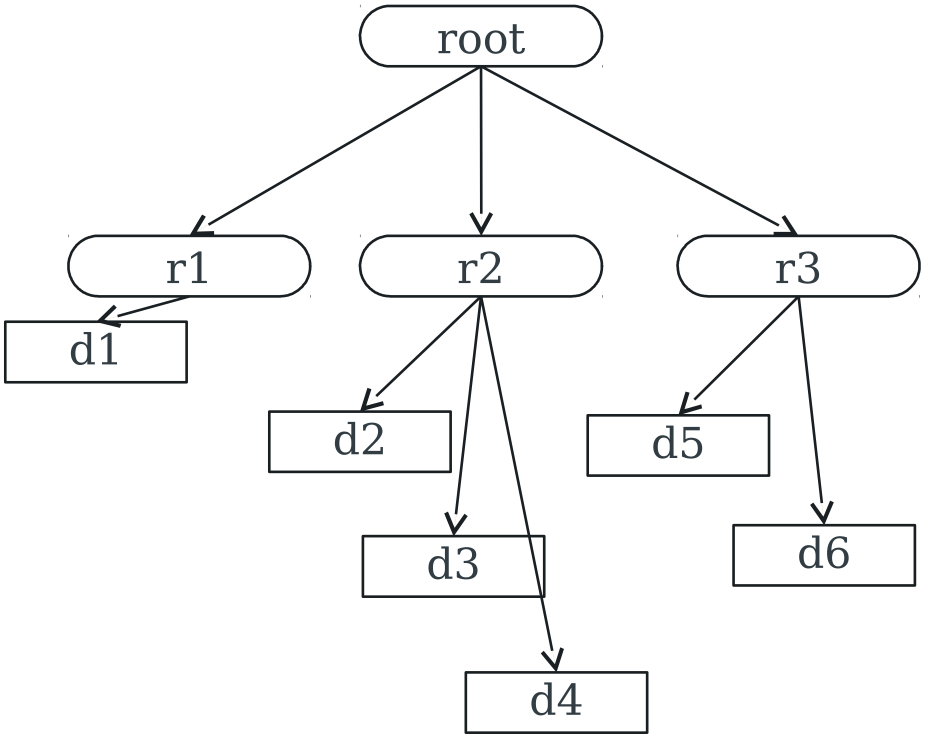
R-Tree

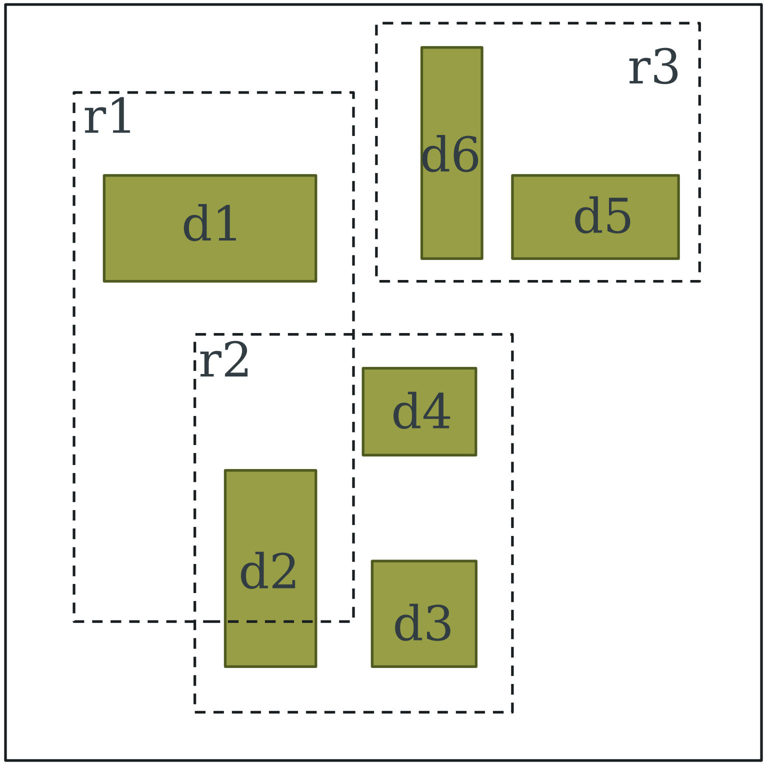
R-Trees

- Used to represent data that consists of k-dimensional *data regions*
- Internal nodes of tree represent regions that contain data regions
- Regions typically defined as top-right, bottom-left coordinates



R-Trees



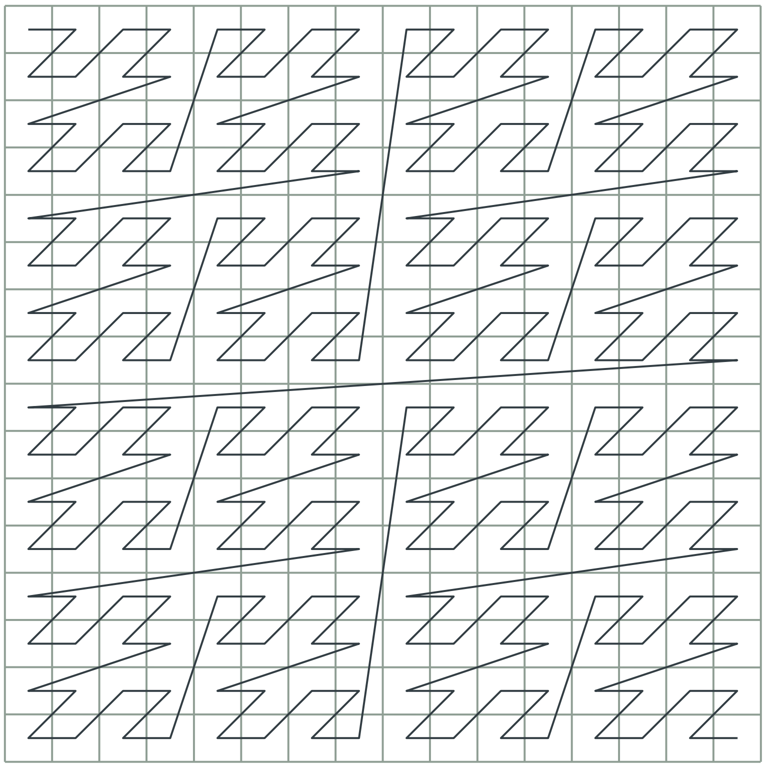


UB-Tree

UB-Tree

Basic approach:

- Map n-dimensional space onto a 1-dimensional line using a fractal space-filling curve
- Partition ranges and index using a B+tree
- When querying, identify regions of n-d space (= segments of 1-d line) that intersect with query rectangle



Z-Index

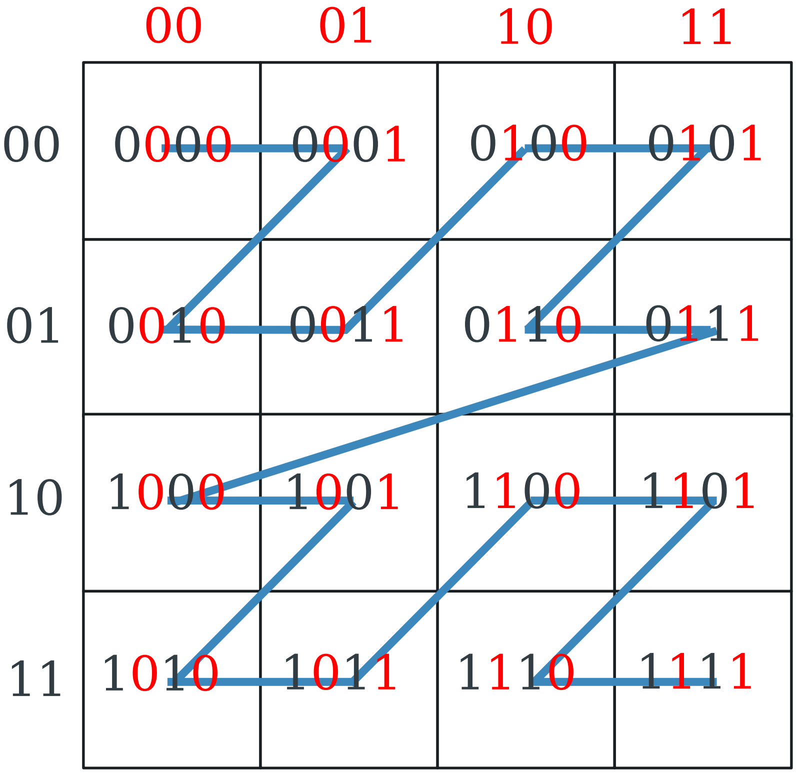
Map domain of each attribute onto n-bit integer

Order of points on Z-curve given by bit-interleaving the positions on the axes

$x = x_1x_2$

$y = y_1y_2$

z-index = $y_1x_1y_2x_2$



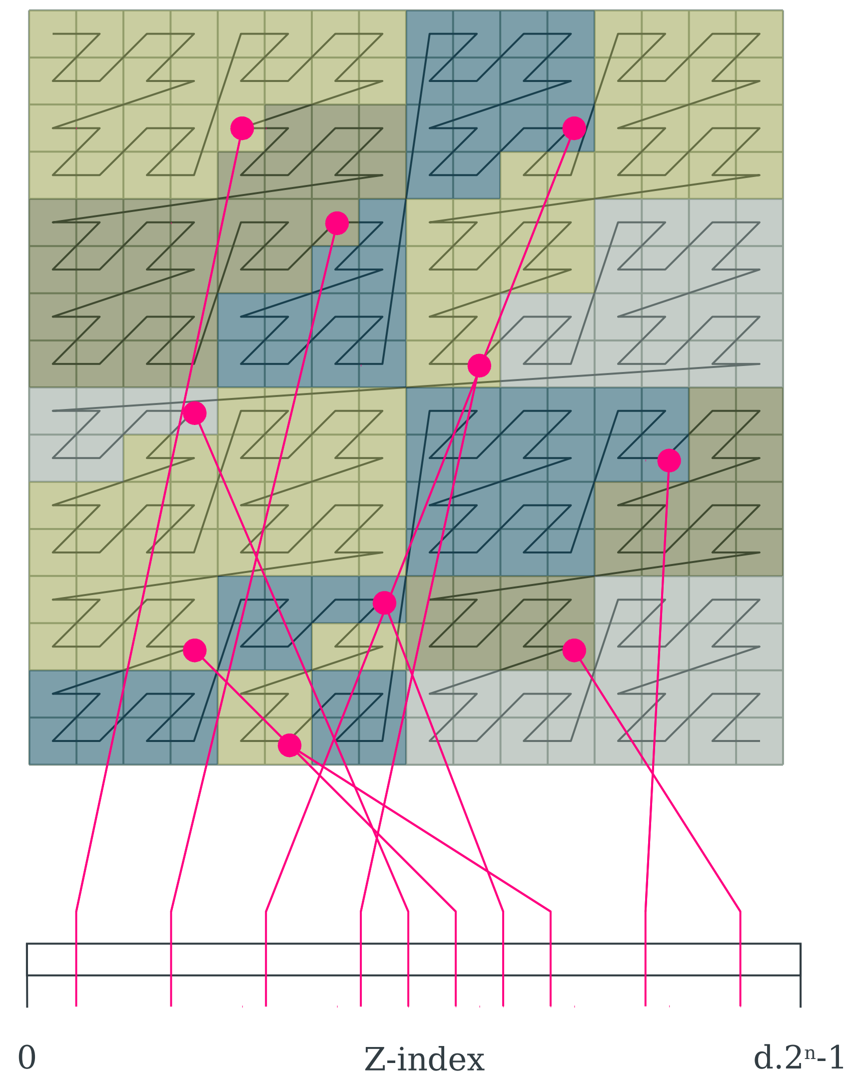
Z-Region Partition

Z-curve partitioned into contiguous ranges (*z-regions*)

- Note that these may not be contiguous regions in the multidimensional space

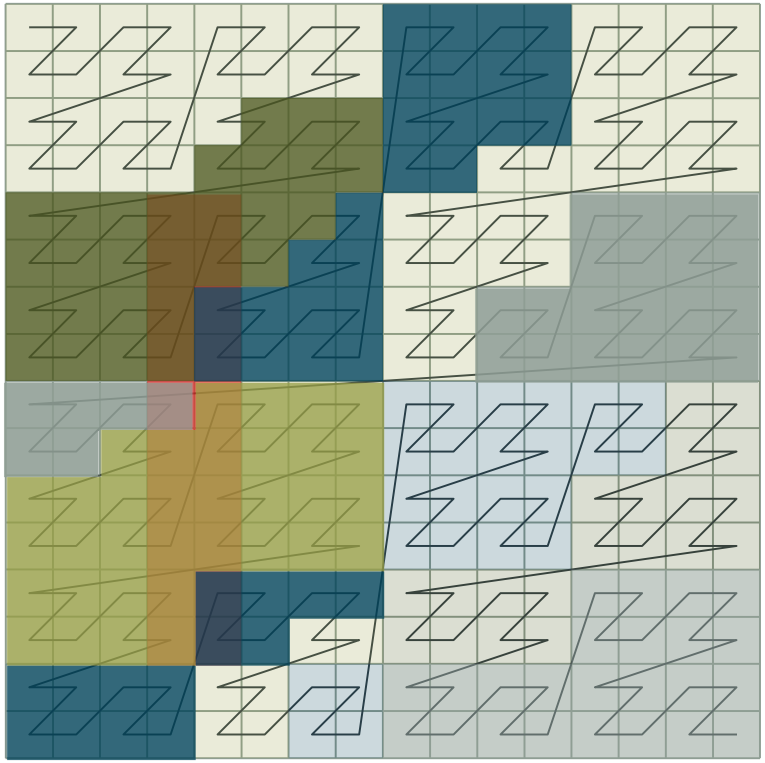
Z-regions mapped to leaf nodes of a B+tree

- A leaf node contain pointers to records whose attribute value locate them within the associated Z-region



Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



Bitmap Indexes

Bitmap indexes

Collection of bit-vectors used to index an attribute

- One bit-vector for each unique attribute value
- One bit for each record

Querying index involves combining bit-vectors with bitwise operators (&, |)

- A 1 in the *i*th position indicates that record *i* is a match

Example

An online homeware vendor sells products p1…p10

- Products p3 and p5 cost £100
- Product p1 costs £200
- Products p2, p7 and p10 cost £300
- Products p4, p6, p8 and p9 cost £400
- Products p1, p4, p5 and p9 are designed for lounges
- Products p5 and p7 are designed for dining rooms
- Products p3, p5, p6 and p10 are designed for kitchens

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

price=£300 $\wedge$ room=kitchen

0100001001 & 0010110001 = 0000000001

p10 is matching product

Compression

- Bit-vectors are typically sparse, with few 1 bits
  - Large amount of wasted space
  - Run-length encoding of bit-vectors to reduce stored size

- Bitwise operators must be applied to original bit-vectors
  - Can decode RLE bit-vectors one run at a time

Bitmap indexes

Pro

- Efficient answering of partial-match queries

Con

- Requires fixed record numbers
- Changes to data file require changes to bitmap index