Multidimensional Indexes

Chapter 5

Search Key

- Retrieve all the records with search key consists of values from multiple attributes.
- Search key is (F1, F2, ..Fk)
  - Separated by special markers.
  - Example
    - If F1=abcd and F2=123, the search key is “abcd#123”
- Indexes on sequential files and Btrees both take advantage of having the keys in sorted order.
- Hash table requires that each key should be known for any lookup
- Number of applications require query support for two or n dimensional space.

Outline

- Applications
- Hash-like structures
- Tree-like structures
- Bit-map indexes

Applications: GIS

- Geographic Information Systems
  - Objects are in two dimensional space
  - The objects may be points or shapes
  - Objects may be houses, roads, bridges, pipelines and many other physical objects.
- Query types
  - Partial match queries
    - Specify the values for one or more dimensions and look for all points matching those values in those dimensions.
  - Range queries
    - Set of shapes within the range
  - Nearest neighbor queries
    - Closest point to a given point
  - Where-am-I queries
    - When you click a mouse, the system determines which of the displayed elements you were clicking.

Applications: Data Cubes

- Data exists in high dimensional space
- A chain store may record each sale made, including
  - The day and time
  - The store in which the sale was made
  - The item purchased
  - The color of the item
  - The size of the item
- Attributes are seen as dimensions multidimensional space, data cube.
- Typical query
  - Give a class of pink shirts for each store and each month of 1998.

Multidimensional queries in SQL

- Represent points as a relation Points(x,y)
- Query
  - Find the nearest point to (10, 20)
  - Compute the distance between (10,20) to every other point.

Rectangles

- Rectangles(id, x11, y11, xur, yur)
- If the query is
  - Find the rectangles enclosing the point (10,20)
- SELECT id

  FROM Rectangles

  WHERE x11 <= 10.0 AND y11 <= 20.0 AND xur >=10.0 AND yur >=20.0;

Data Cube

- Fact table
  – Sales(day,store, item. color, size)
- Query
  – Summarize the sale of pink shirts by day and store
- SELECT day, store, COUNT(*) AS totalSales

  FROM Sales

  WHERE item='shirt' AND color='pink'

  GROUP BY day, store;

Executing range queries in Conventional Indexes

- Motivation: Find records where

  DEPT = “Toy” AND SAL > 50k
- Strategy 1: Use “Toy” index and check salary
- Strategy II: Use “Toy” index and SAL index and intersect.
  - Complexity
    - Toy index: number of disk accesses = number of disk blocks
    - SAL index: number of disk accesses= number of records
- So conventional indexes of little help.

Executing nearest-neighbor queries using conventional indexes

- There might be no point in the selected range
  - Repeat the search by increasing the range
- The closest point within the range might not be the closest point overall.
  - There is one point closer from outside range.

Other Limitations of Conventional Indexes

- We can only keep the file sorted on only attribute.
- If the query is on multiple dimensions
  - We will end-up having one disk access for each record
- It becomes too expensive

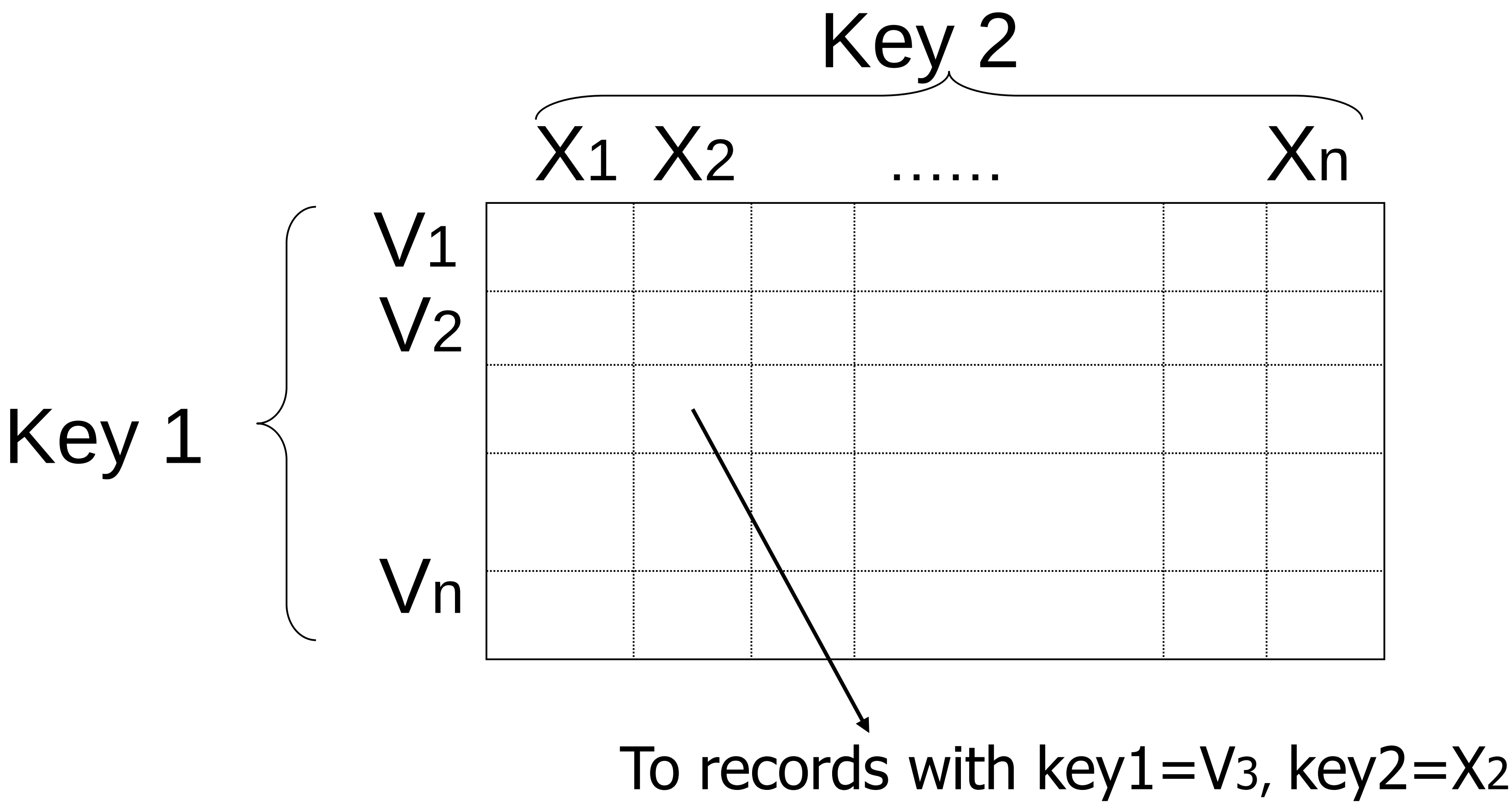
Overview of Multidimensional Index Structures

- Hash-table-like approaches
- Tree-like approaches
- In both cases, we give-up some properties of single dimensional indexes
- Hash
  - We have to access multiple buckets
- Tree
  - Tree may not be balanced
  - There may not exist a correspondence between tree nodes and disk blocks
  - Information in the disk block is much smaller.

Hash like structures: GRID Files

- Each dimension, grid lines partition the space into stripes,
  - Points that fall on a grid line will be considered to belong to the stripe for which that grid line is lower boundary
  - The number of grid lines in each dimension may vary.
  - Space between grid lines in the same dimension may also vary

Grid Index



CLAIM

- Can quickly find records with
  - key 1 = $V_i \wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

CLAIM

- Can quickly find records with
  - key 1 = V_i $\wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

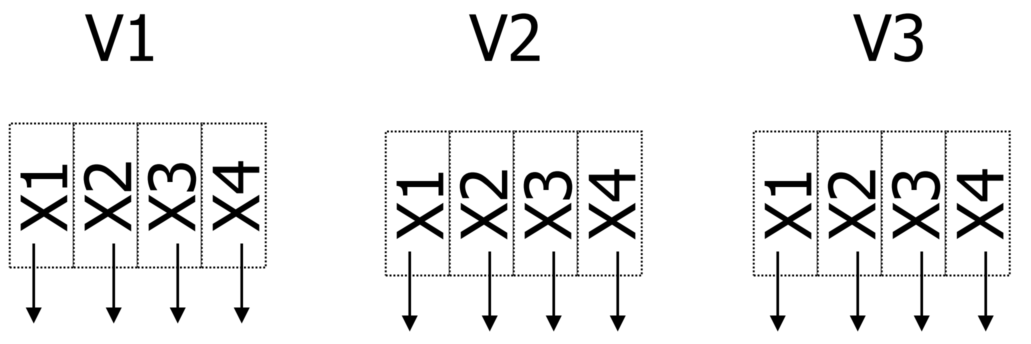
- And also ranges....
  - E.g., key 1 $\geq V_i$ $\wedge$ key 2 $< X_j$

✉ But there is a catch with Grid Indexes!

- How is Grid Index stored on disk?

Like

Array...

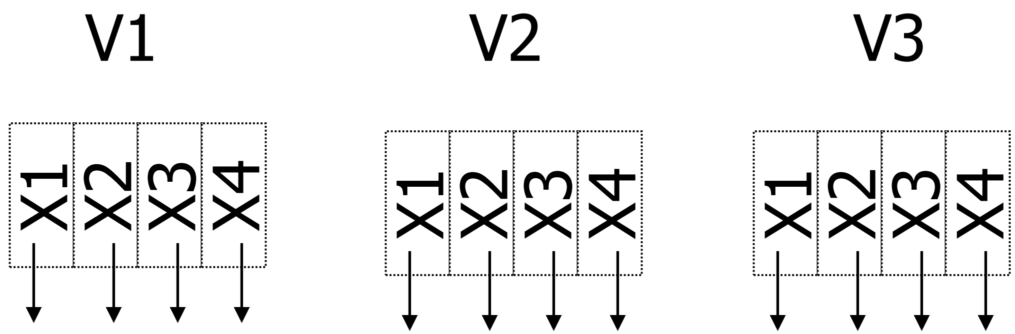


✉ But there is a catch with Grid Indexes!

- How is Grid Index stored on disk?

Like

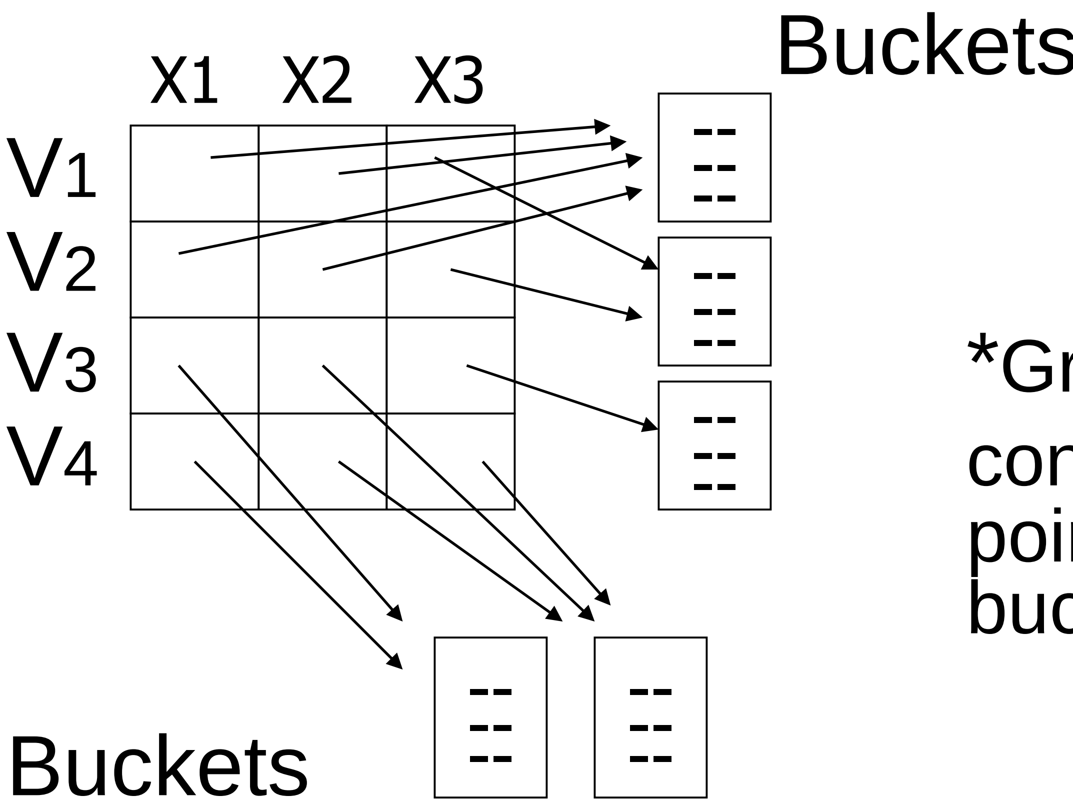
Array...



Problem:

- Need regularity so we can compute position of <Vi,Xj> entry

Solution: Use Indirection



*Grid only contains pointers to buckets

With indirection:

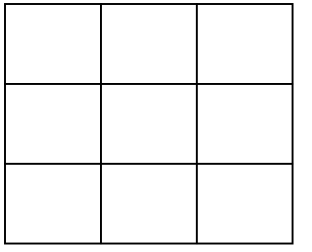
- Grid can be regular without wasting space
- We do have price of indirection

Can also index grid on value ranges

Salary

0-20K	1
20K-50K	2
50K- ∞	3

Grid



1	2	3
Toy	Sales	Personnel

Linear Scale

Grid files

+ Good for multiple-key search

- Space, management overhead

  (nothing is free)

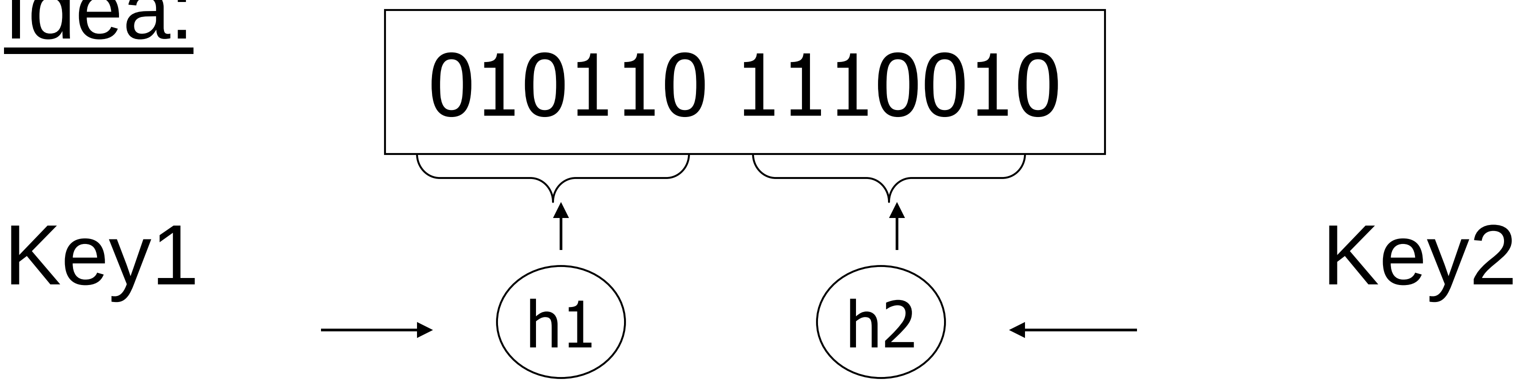
- Need partitioning ranges that evenly split keys

Partitioned Hash Functions

- Design a hash function so that it produces number of bits which are divided abomg attributes.

Partitioned hash function

Idea:



EX:

h1(toy) =0

h1(sales) =1

h1(art) =1

.

.

h2(10k) =01

h2(20k) =11

h2(30k) =01

h2(40k) =00

.

.

000	
001	
010	
011	
100	
101	
110	
111	

Insert

<Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	
001	<Fred>
010	
011	
100	
101	<Joe><Sally>
110	
111	

Insert <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales $\wedge$ Sal=40k

h1(toy) =0
h1(sales) =1
h1(art) =1
:
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
:

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales $\wedge$ Sal=40k

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Sal=30k

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Sal=30k

look here

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales

look here

Tree-like Structures

- Multiple-key indexes
- Kd-trees
- Quad trees
- R-trees

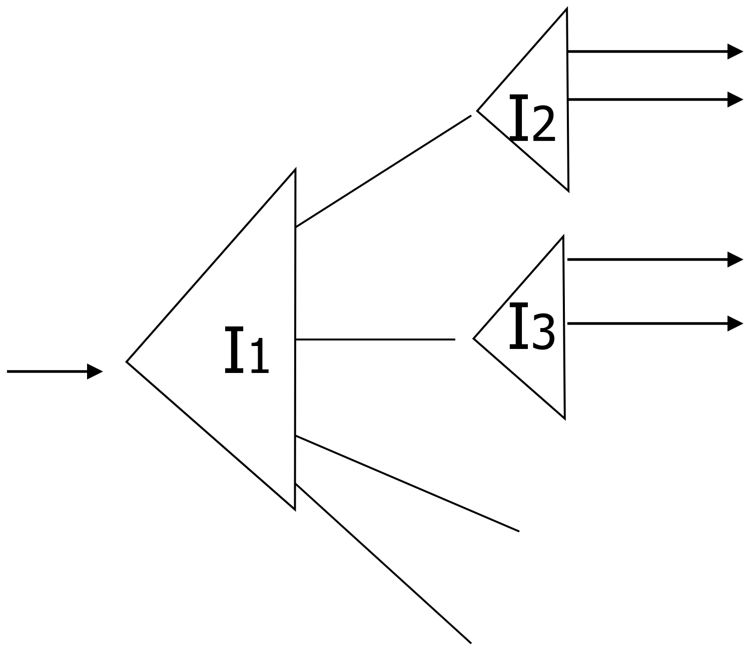
Multiple-key indexes

- Several attributes representing dimensions of data points
- Multiple key index is an index of indexes in which the nodes at each level are indexes for one attribute.

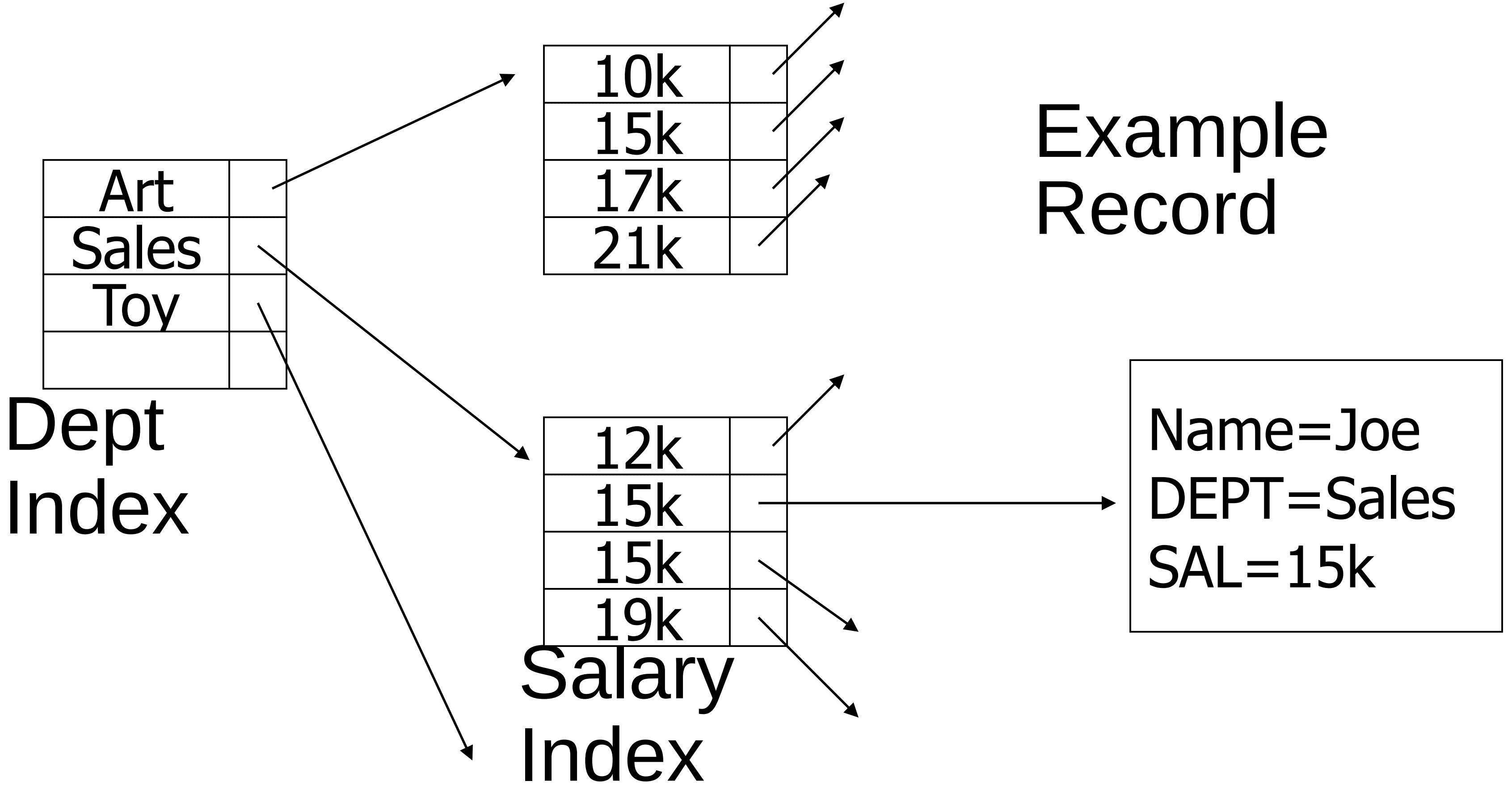
Strategy:

- Multiple Key Index

One idea:



Example



Art
Sales
Toy
Dept
Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary
Index
Example
Record
Name=Joe
DEPT=Sales
SAL=15k

For which queries is this index good?

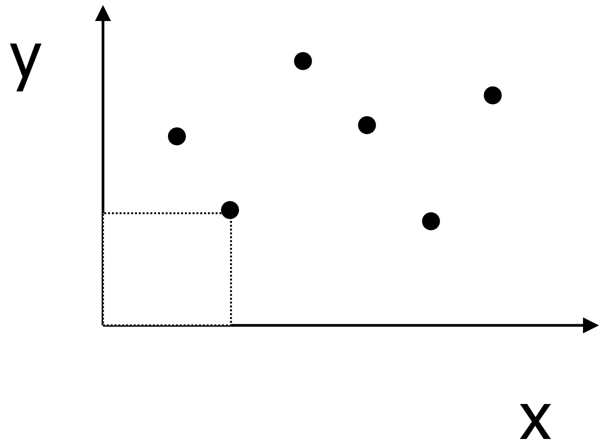
- ☐ Find RECs Dept = “Sales” $\wedge$ SAL=20k
- ☐ Find RECs Dept = “Sales” $\wedge$ SAL $\geq$ 20k
- ☐ Find RECs Dept = “Sales”
- ☐ Find RECs SAL = 20k

KD-trees

- K dimensional tree is generalizing binary search tree into multi-dimensional data.
- A KD tree is a binary tree in which interior nodes have an associated attribute “a” and a value “v” that splits data into two parts.
- The attributes at different levels of a tree are different, and levels rotating among the attributes of all dimensions.

Interesting application:

- Geographic Data



DATA:

$<X_1, Y_1$, Attributes$>$

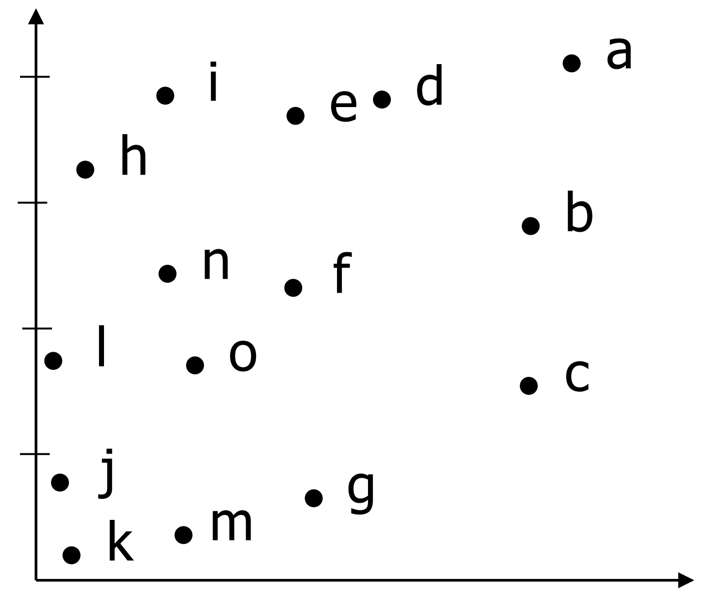
$<X_2, Y_2$, Attributes$>$

⋮

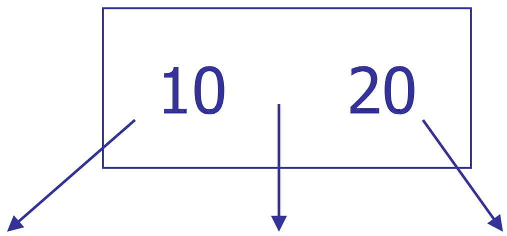
Queries:

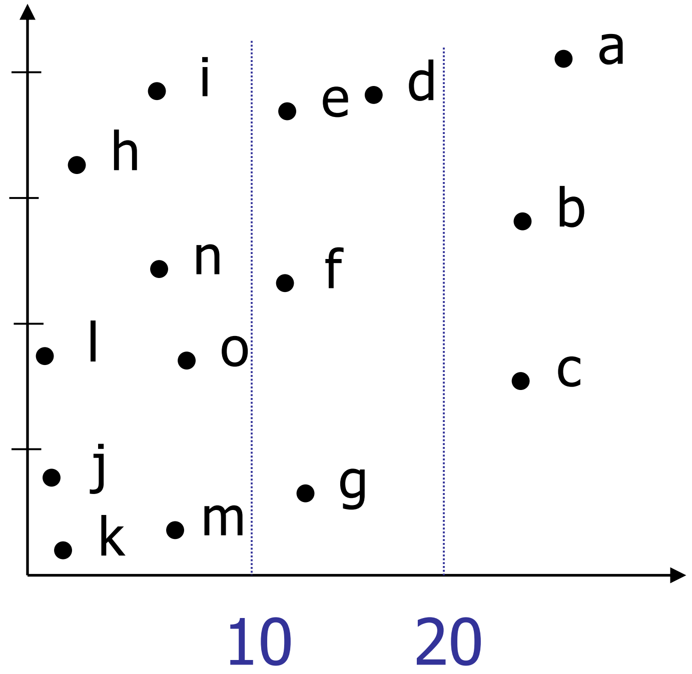
- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

Example

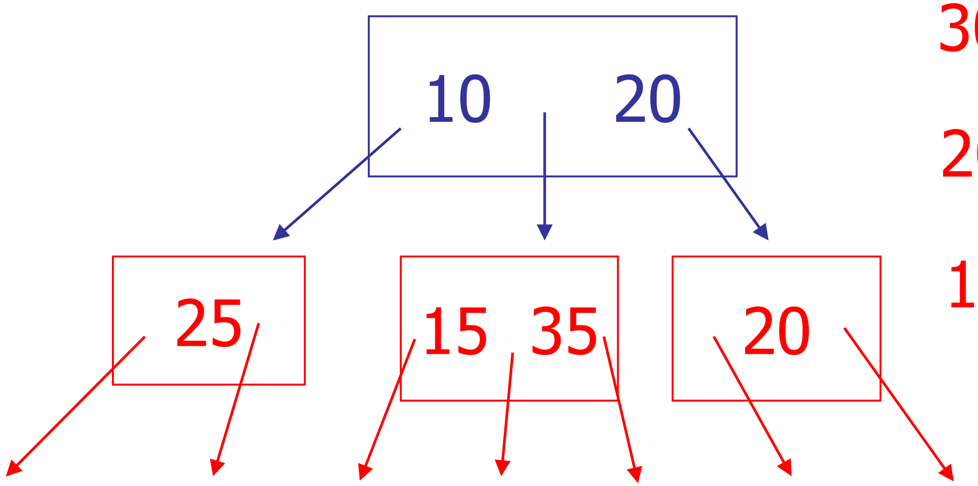


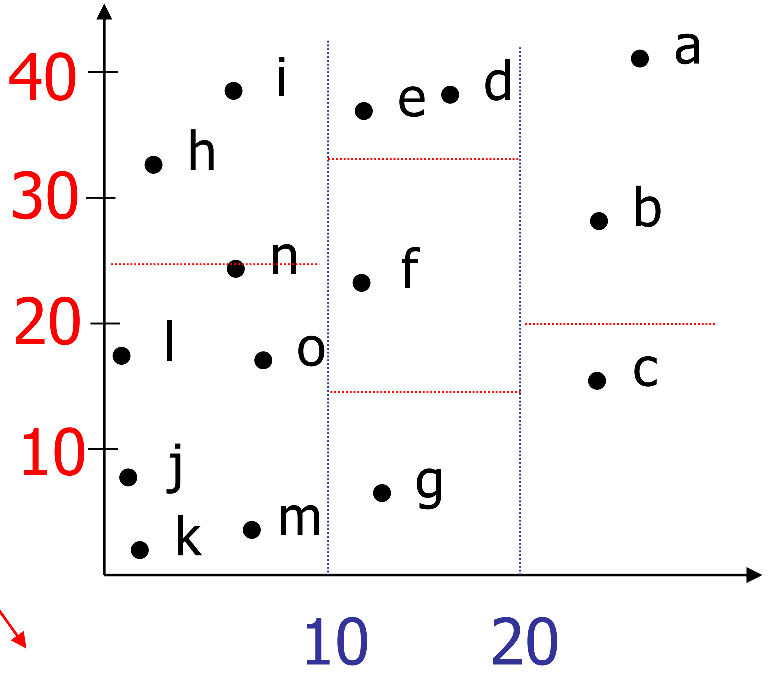
Example



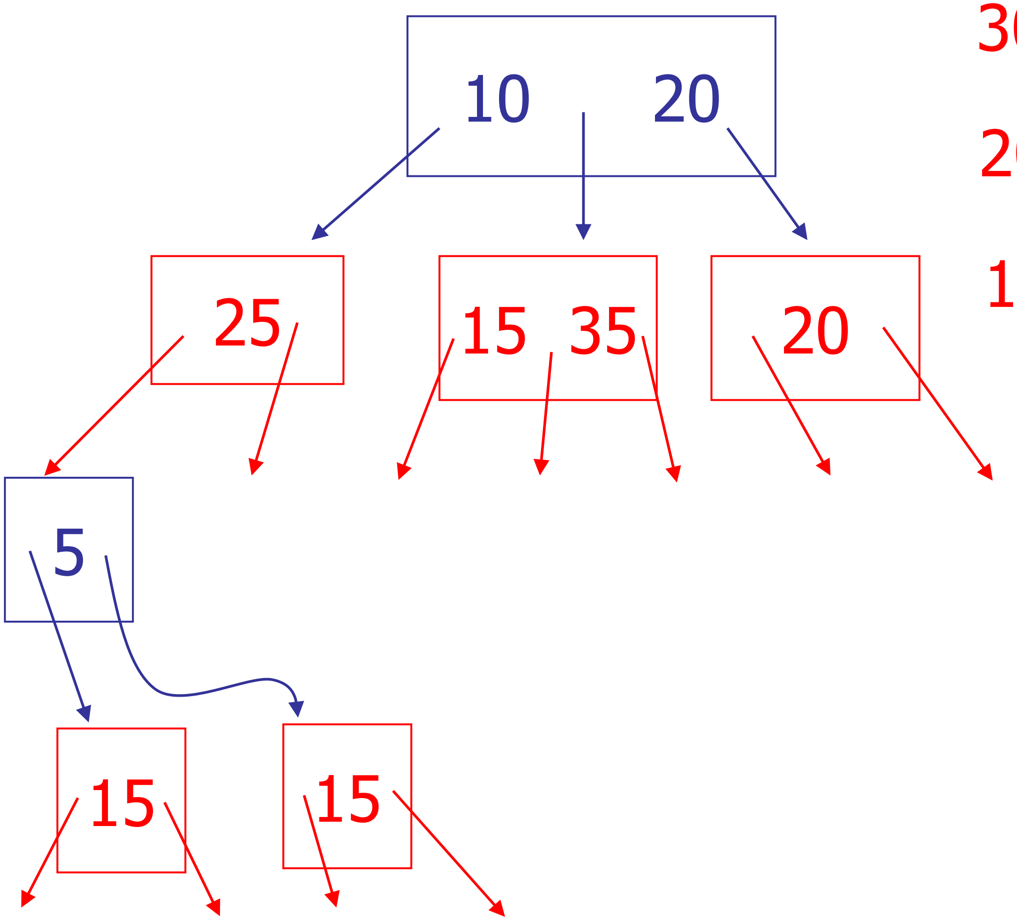


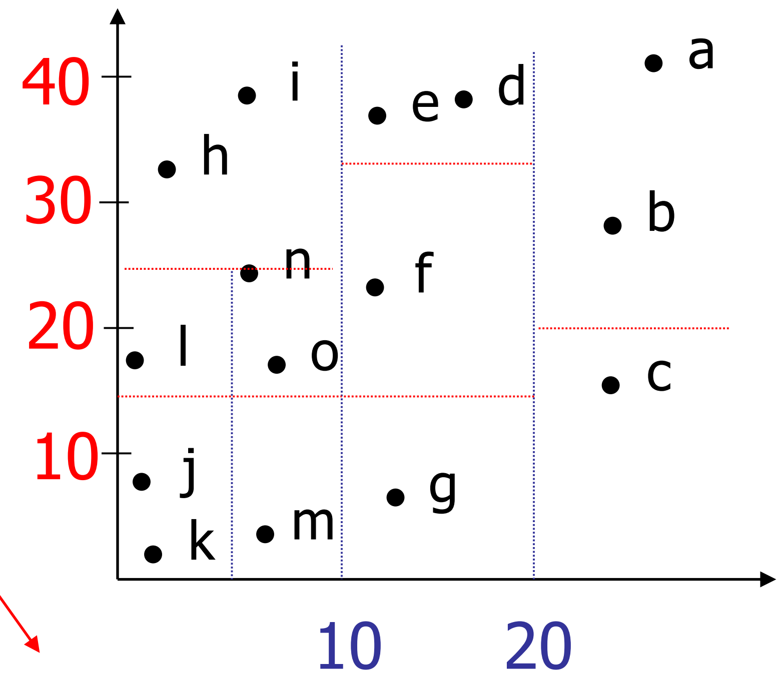
Example



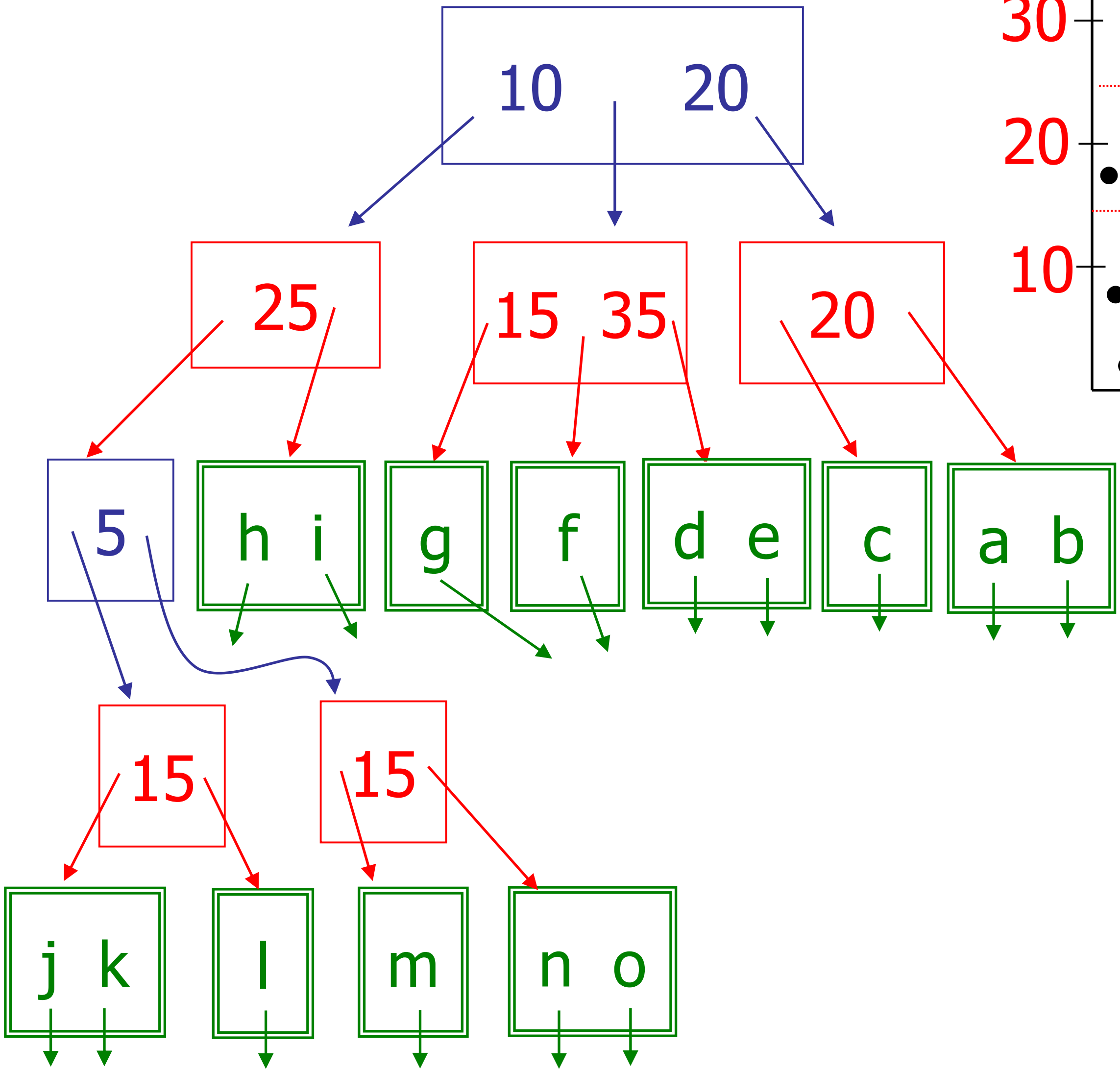


Example

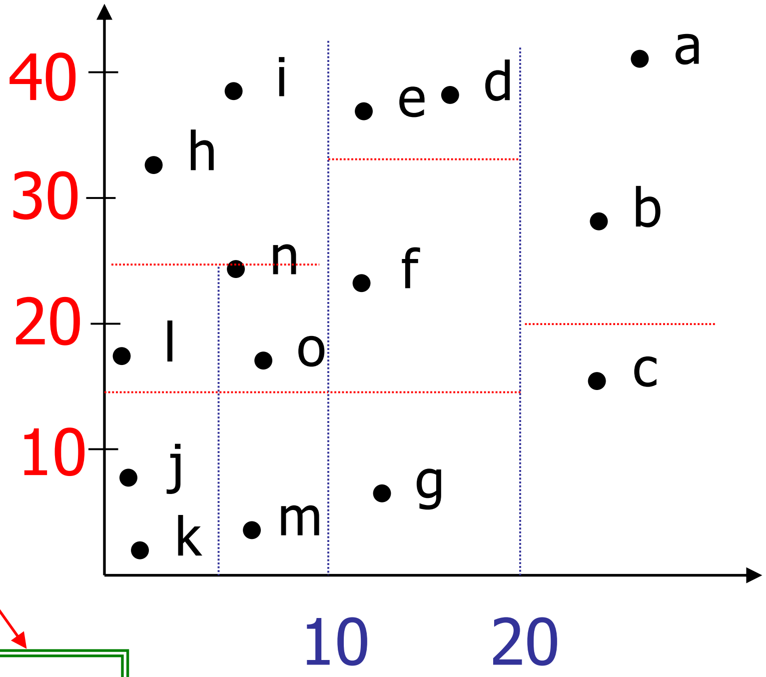




Example

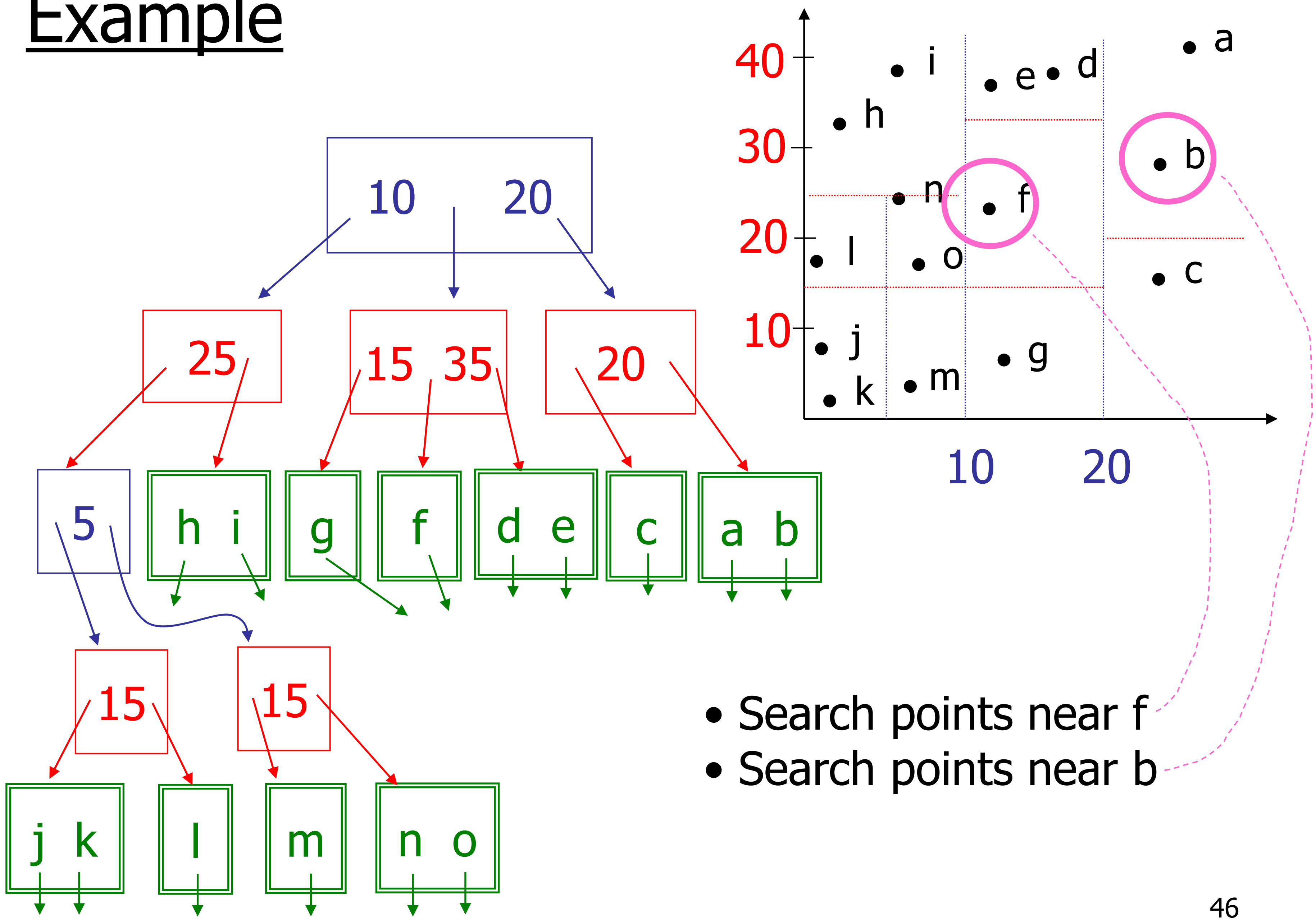


10 20
25
15 35
20
5
h i
g
f
d e
c
a b
15
15
j k
l
m
n o



40
30
20
10
10
20
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o

Example



- Search points near f
- Search points near b

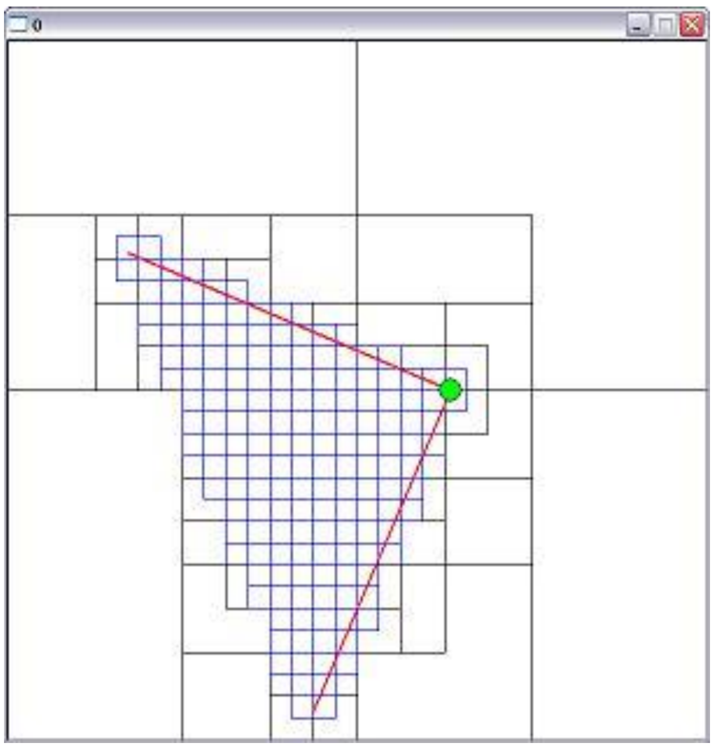
Queries

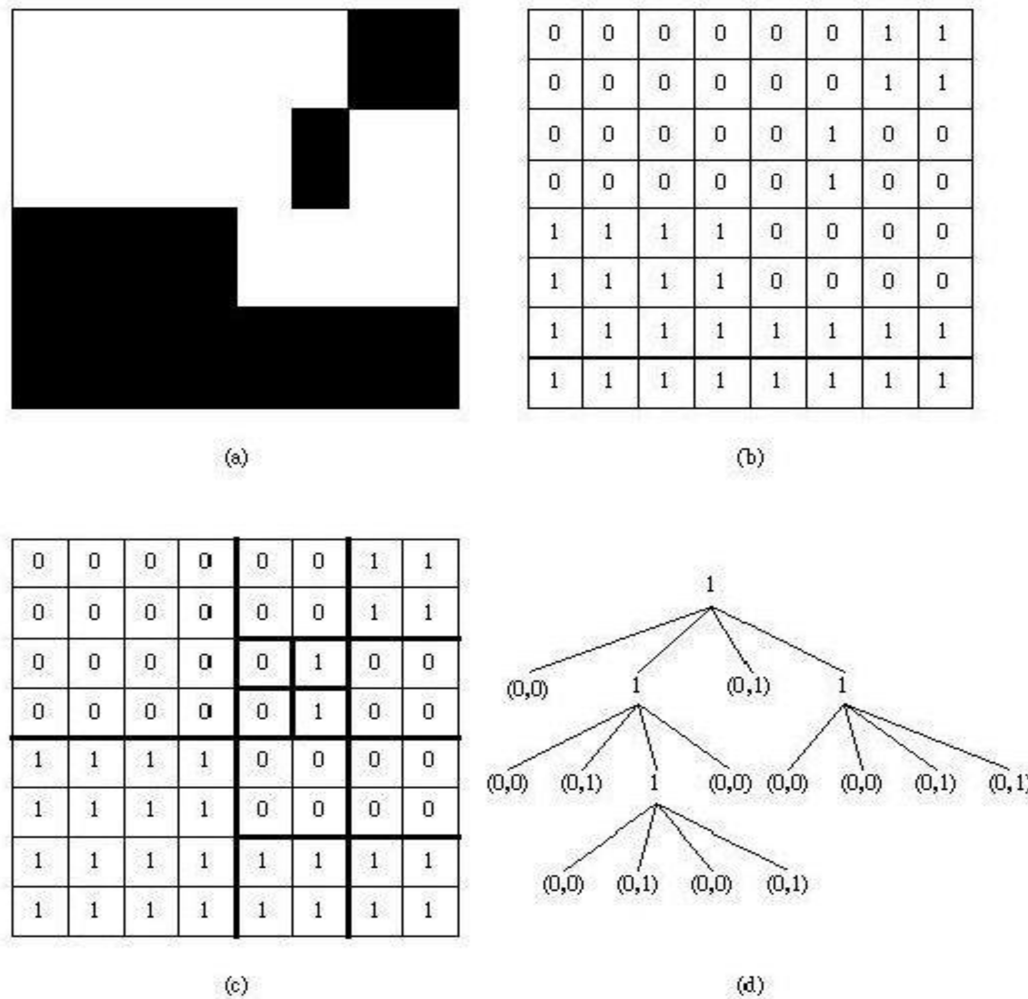
- Find points with Yi > 20
- Find points with Xi < 5
- Find points “close” to i = <12,38>
- Find points “close” to b = <7,24>

Quad trees

- Divides multidimensional space into quadrants and divides the quadrants same way if they have too many points.
- If the number of points in a square
  - fits in a block, it is a leaf node
  - no longer fits in a block, it becomes an interior node, four quadrants are its children.

Quad tree pictures

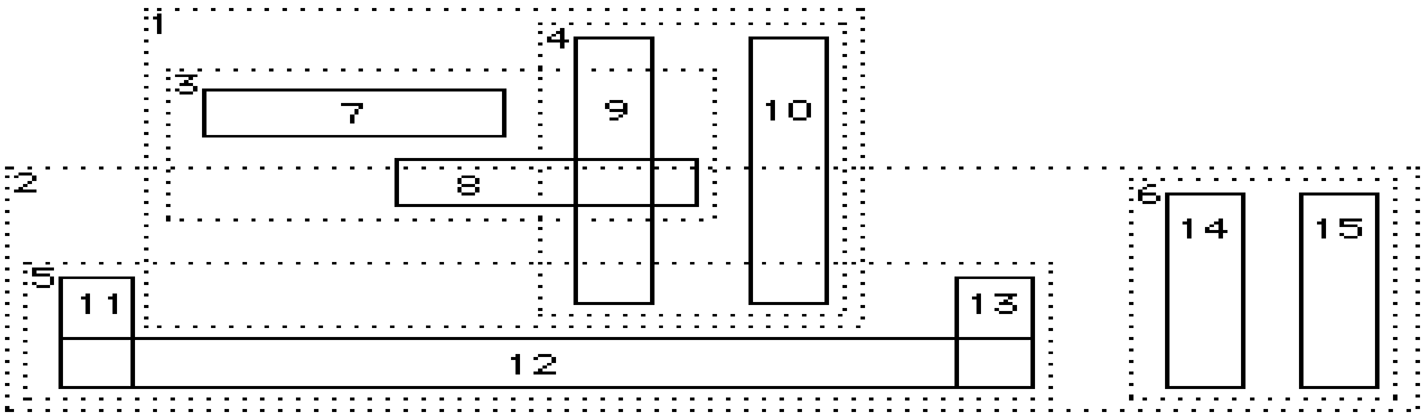




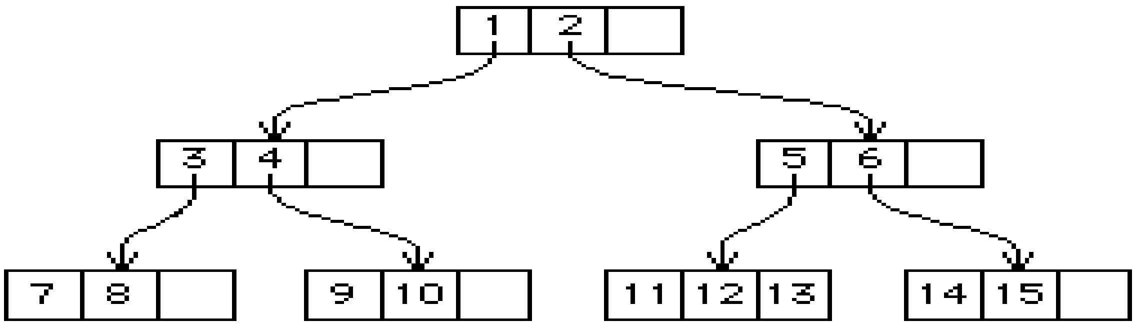
R-tree

- Captures the spirit of B-tree for multidimensional data.
- Represents a collection of regions by grouping them into a hierarchy of larger regions.
- Data is divided into regions.
- Interior node is corresponds to interior region
  - region can be of any shape
    - Rectangular is popular
  - Children corresponds to sub-regions.

R-tree



(a)



(b)

Bitmap indexes

- Assume that records have permanent numbers
- A bit-map index is a collection of bit vectors of length n, one for each value may appear in the field F.
- The vector for value v has 1 in position i if the i’th record has v in field F, and it has 0 there if not.
- Example for F and G fields:
  - (30, foo), (30,bar), (40, baz), (50, foo), (40, bar), (30, baz)
  - Bit index for F, each of 6 bits. For 30, it is 110001, for 40, it is 001010, and for 50, it is 000100.
  - Bit index for G also have three vectors. For foo it is, 100100, for bar it is 010010, and for baz it is 001001.

Bit map indexes: Partial match

- Bit maps allow answering of partial match queries quickly and efficiently.
- Example:
  - Movie(title, year, length, studioName)
  - SELECT title
  - FROM Movie
  - WHERE studioName= ‘Disney’ and Year=1965;
- If we have bitmap for studioName and year, then intersection or AND operation will give the result.
- Bit vectors do not occupy much space.

Bitmap indexes: range queries

- Example: consider the gold jewelry data of twelve points
  - 1 (25,60), 2(45,60), 3(50,75), 4(50,100), 5(50,120), 6(70, 110), 7(85,140), 8(30,260), 9(25,400), 10(45,350), 11(50,275), 12(60,260)
- Age has seven different values
  - 25(100000001000), 30(000000010000), 45(010000000100), 50(001110000010), 60(000000000001), 70(000001000000)
  - 85(000000100000)
- Salary has 10 different values
  - 60(000000000000), 75(001000000000), 100(000100000000)
  - 110(000001000000), 120(000010000000), 140(000000100000)
  - 260(000000010001), 275(000000000010), 350(000000000100)
  - 400(000000001000),

Example continued

- Find the jewelry buyers with an age range 45-55 and salary in the range 100-200
- Find the bit vectors of for the age values in the range and take OR
  - 010000000100 (for 45) and 001110000010 (for 50)
  - Result: 011110000110
- Find the bit vectors of salaries between 100 and 200 thousand.
  - There are four: 100,110,120, and 140, their bitwaise OR is 000111100000
- Take AND of both bit vectors
  - 000110000000
  - Find two records (50,100) and (50,120) are in the range.

Compressed bitmaps

- If number of different values is large, then number of 1 is rare.
- Run-length coding is used
  - Sequence of 0's followed by 1.
  - Example: 000101 is two runs, 3 and 1. the binary representation is 11 and 1. So it is decoded as 111.
- To save space, the bitmap indexes tend to consist of vectors with very few 1's are compressed using run-length coding.

Finding bit vectors

- Use any index technique to find the values.
- From the values to bit vectors.
- B-tree is a good choice.