

# CS525: Advanced Database Organization

## Notes 5: Multi-dimensional indexes

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

[yelmehdwi@iit.edu](mailto:yelmehdwi@iit.edu)

September 25, 2018

Slides: adapted from a course taught by [Shun Yan Cheung, Emory University](#)

# Topics

- Multi-dimensional information and query
- Motivation for Multi-dimensional indexes
- Multi-dimensional index structures
  - Hash like structures
  - Tree like structures
- Bitmap indices

# Recap

- We have studied the following 3 index structures:
  - Sorted indexes
  - $B^+$ -tree indexes
  - Hashing-based indexes
- Common property
  - The search key values are values taken from a one-dimensional space/set

# Multi-dimensional Information

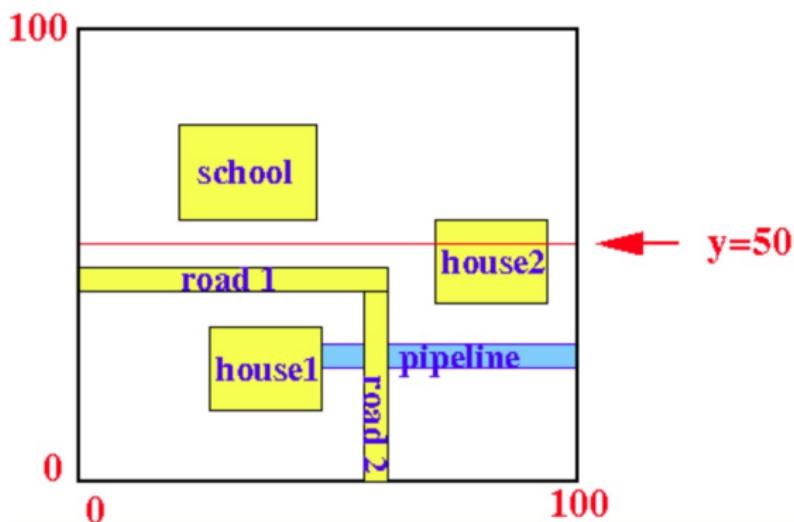
- There are information that are naturally multi-dimensional
  - e.g., Geographic information:
    - Stores objects in a (typically) two-dimensional space.
    - The objects may be points or shapes.
    - Often, these databases are maps, where the stored objects could represent houses, roads, bridges, pipelines, and many other physical objects.

# Multi-dimensional queries

- Partial Match queries
- Range queries
- Nearest neighbor queries
- Where-am-I queries

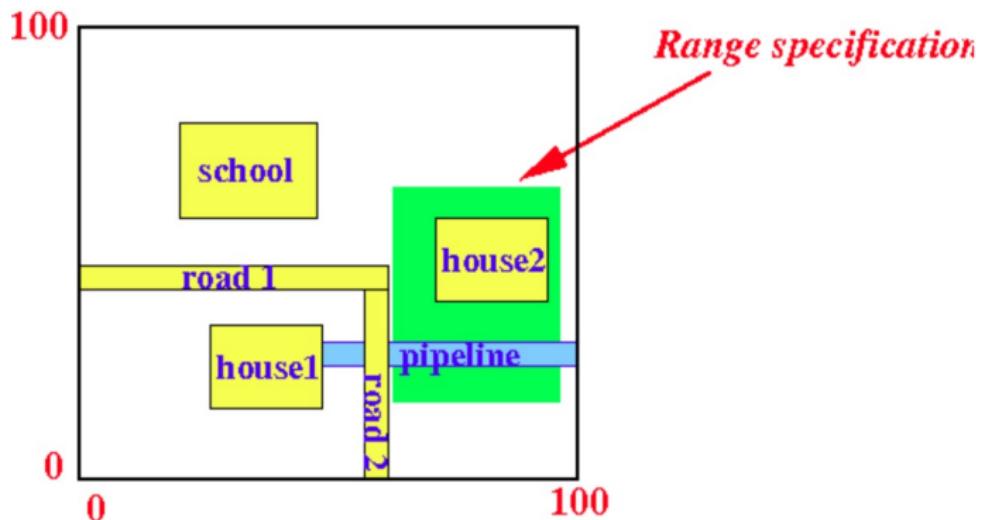
## Partial Match queries

- The query specifies conditions on some dimensions but not on all dimensions
- e.g., Find all points/objects that intersects with  $y = 50$



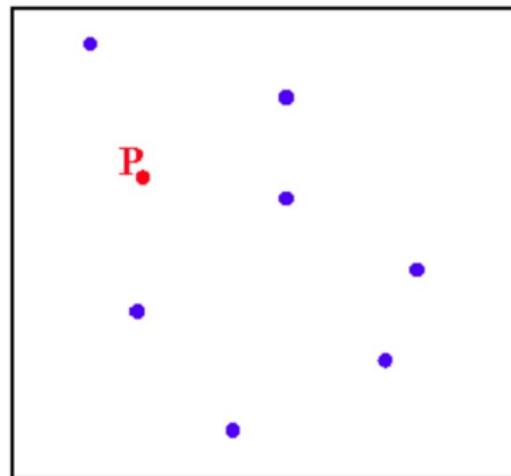
# Range queries

- Find objects that are located either partial or wholly within a certain range
- e.g., Find all objects that have an overlap with the green area:



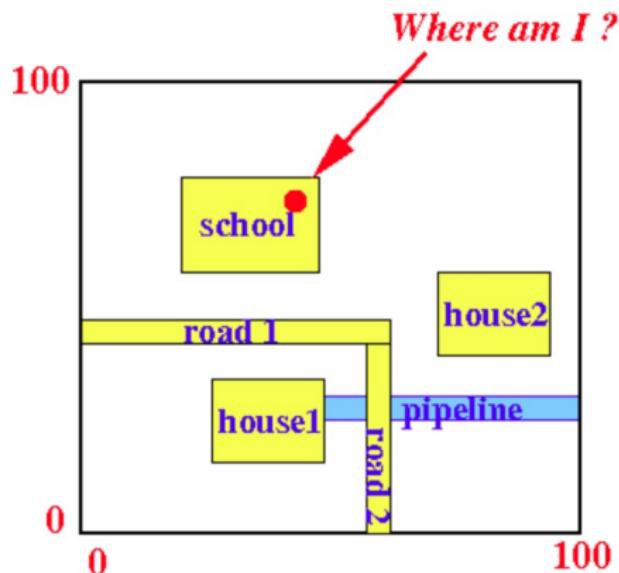
## Nearest neighbor queries

- Find the closest point to a given point.
- Suppose we have a relation containing points on a map
- Each point is stored in the following relation as Point(x,y)
- Find the point that is closest to point P(10,20)



# Where-am-I queries

- Given a location (i.e., coordinate)
- Find the object(s) that contains the location

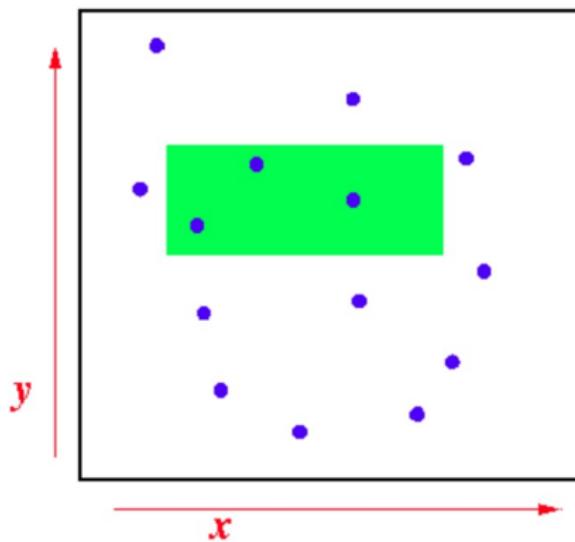


# Motivation for developing multi-dimensional indexes

- Are multi-dimensional indexes necessary?
- Can one-dimensional index technique support multi-dimensional queries efficiently?
- Case Study: Try to process a range query using a  $B^+$ -tree index

# Processing multi-dimensional query using B<sup>+</sup>-tree index

- Database and query description
  - Database: object locations
    - `Object(x,y, other-attributes)`
    - where  $x$  and  $y$  are the coordinates of the object
  - Query
    - Find all objects that lies within a rectangle

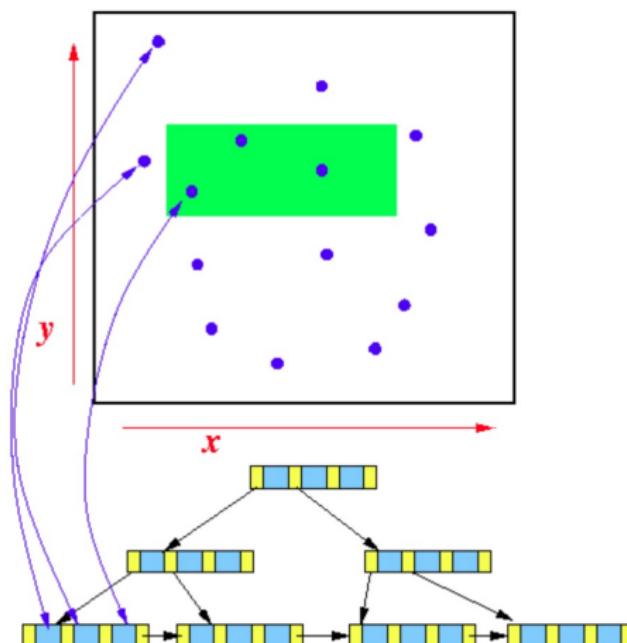


## Processing multi-dimensional query using B<sup>+</sup>-tree index

- Suppose we have B<sup>+</sup>-tree indexes on:
  - The *x*-coordinate attribute of Object and
  - The *y*-coordinate attribute of Object

# Processing multi-dimensional query using B<sup>+</sup>-tree index

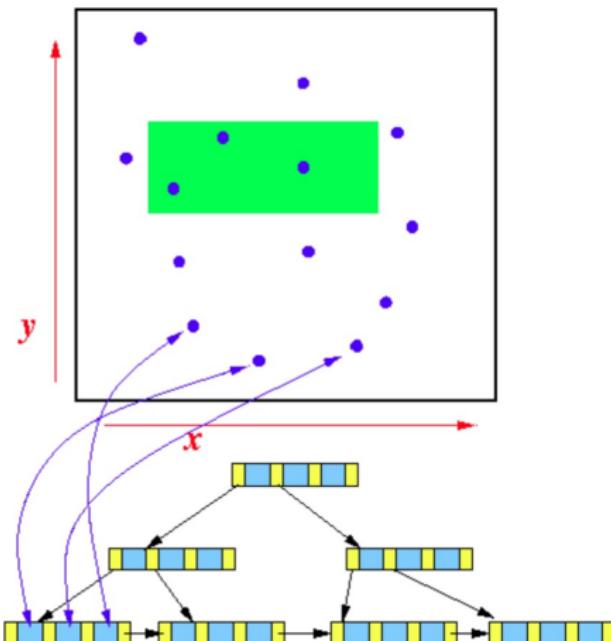
- The B<sup>+</sup>-tree on the **x-coordinate** information looks like this:



- The **point** with the smallest **x-coordinate** value is the **left-most leaf key**

# Processing multi-dimensional query using B<sup>+</sup>-tree index

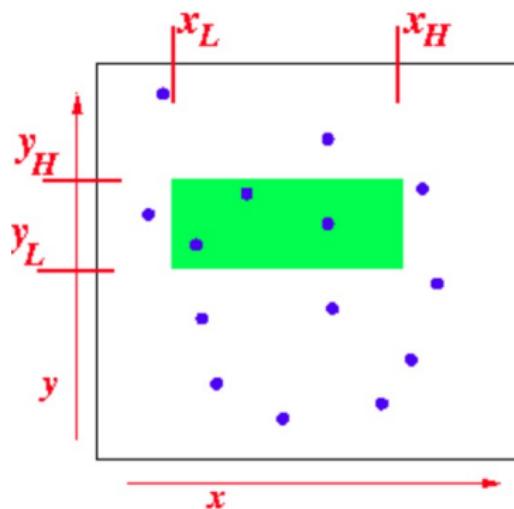
- The B<sup>+</sup>-tree on the **y**-coordinate information looks like this:



- The **point** with the smallest **y**-coordinate value is the left-most leaf key

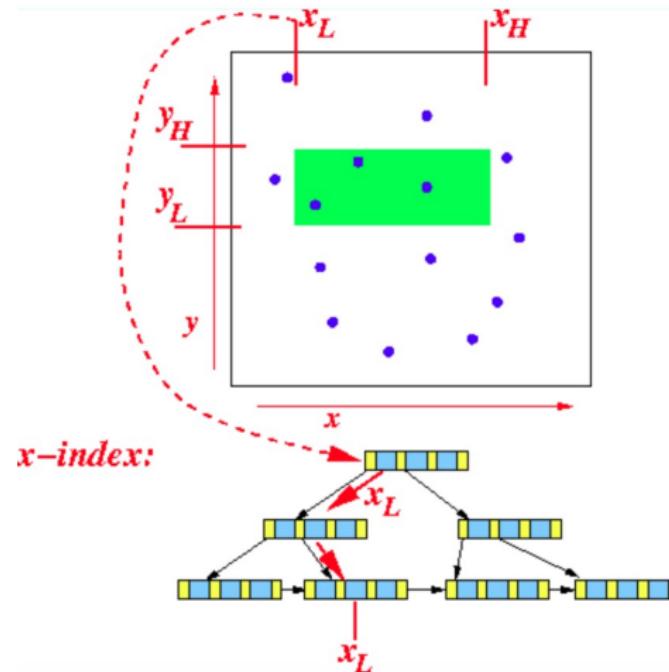
# Processing multi-dimensional query using B<sup>+</sup>-tree index

- Range query:
  - Find **all points** such that:
    - $x_L \leq x \leq x_H$  and
    - $y_L \leq y \leq y_H$



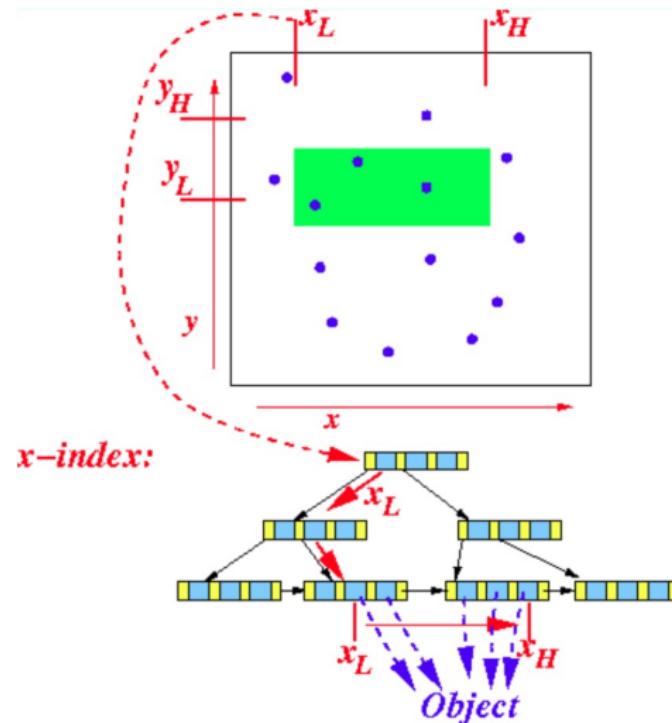
# How to use the $B^+$ -tree indexes to process range query

1. Use the  $x$ - $B^+$ -tree index and find the first value that is  $\geq x_L$



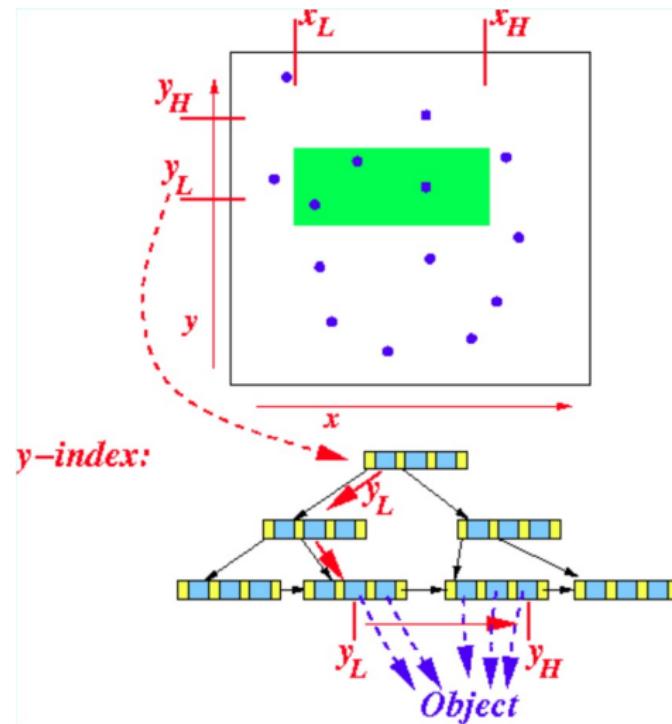
# How to use the B<sup>+</sup>-tree indexes to process range query

- Traverse the **leaf nodes** to find all record pointers for which  $x_L \leq x \leq x_H$



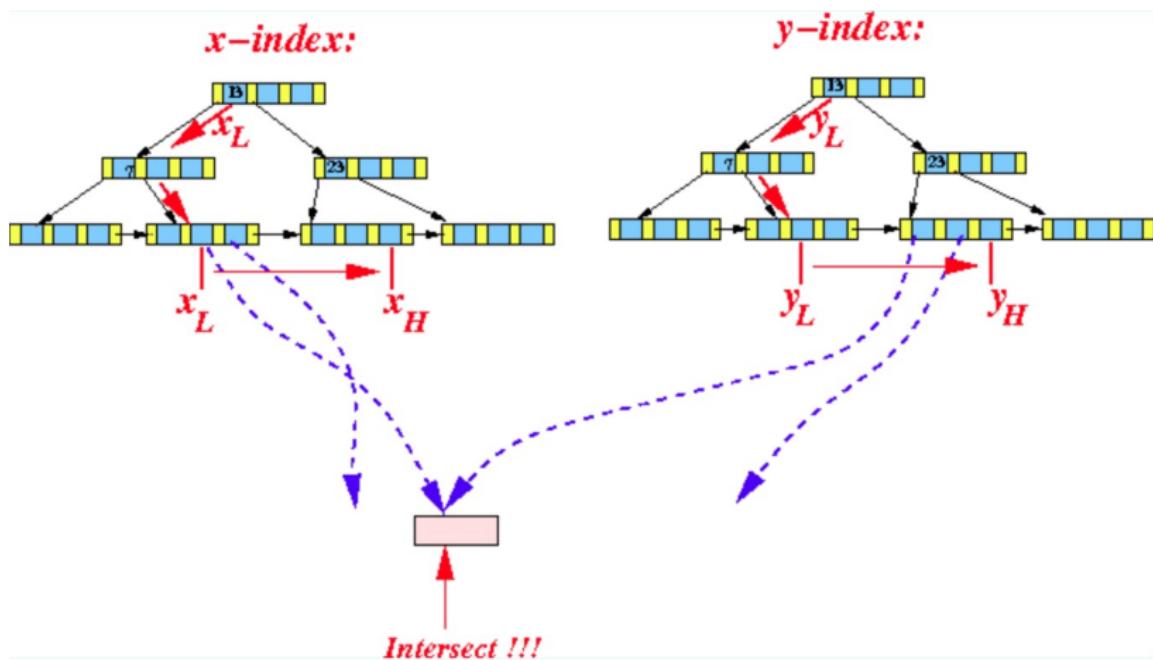
# How to use the $B^+$ -tree indexes to process range query

2. Do the same for the  $y$ -coordinate



# How to use the B<sup>+</sup>-tree indexes to process range query

3. Compute the intersection of the 2 pointer sets

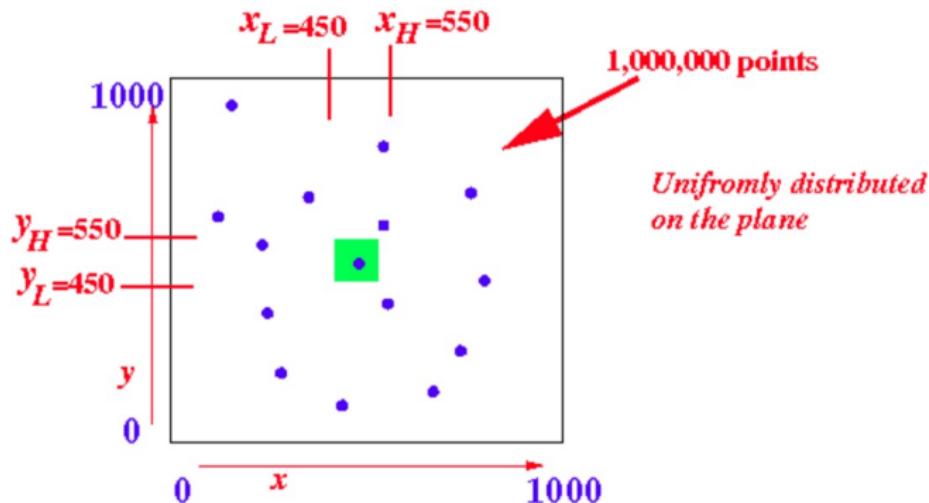


## How to use the B<sup>+</sup>-tree indexes to process range query

4. Retrieve the records using the record pointers in the intersection
  - These records are guaranteed to satisfy:
    - $x_L \leq x \leq x_H$  and
    - $y_L \leq y \leq y_H$
  - This solution is not faster than scanning the entire relation

## Example

- Consider the following situation:

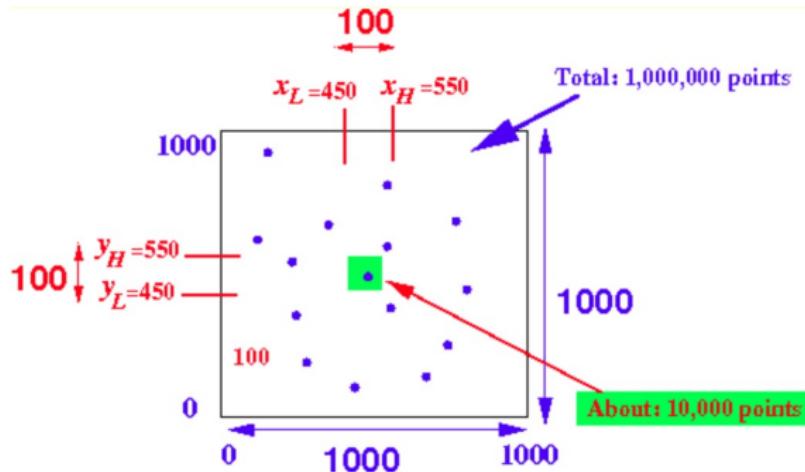


- Some statistics:

- Green area =  $100 \times 100 = 10,000$
- Total area =  $1000 \times 1000 = 1,000,000$
- Green area =  $0.01 \times$  Total area

## Example

- Total # points in area = 1,000,000
- # points in green area  $\cong 0.01 \times 1,000,000 = 10,000$



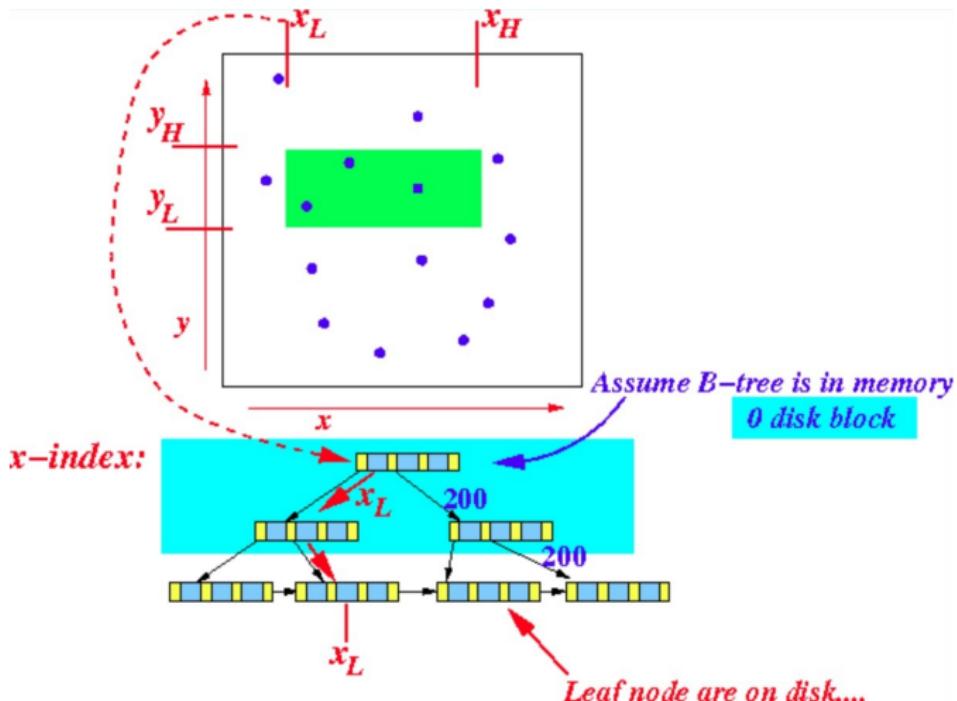
- # points with  $x$ -coordinate in  $[450, 550]$   
 $\cong 0.1 \times 1,000,000 = 100,000$
- # points with  $y$ -coordinate in  $[450, 550]$   
 $\cong 0.1 \times 1,000,000 = 100,000$

## Storage information

- To compute the processing cost = # disk blocks accessed
  - 1 disk block contains 100 points
  - 1 B<sup>+</sup>-tree block (node) contains an average of 200 (key, ptr) pairs

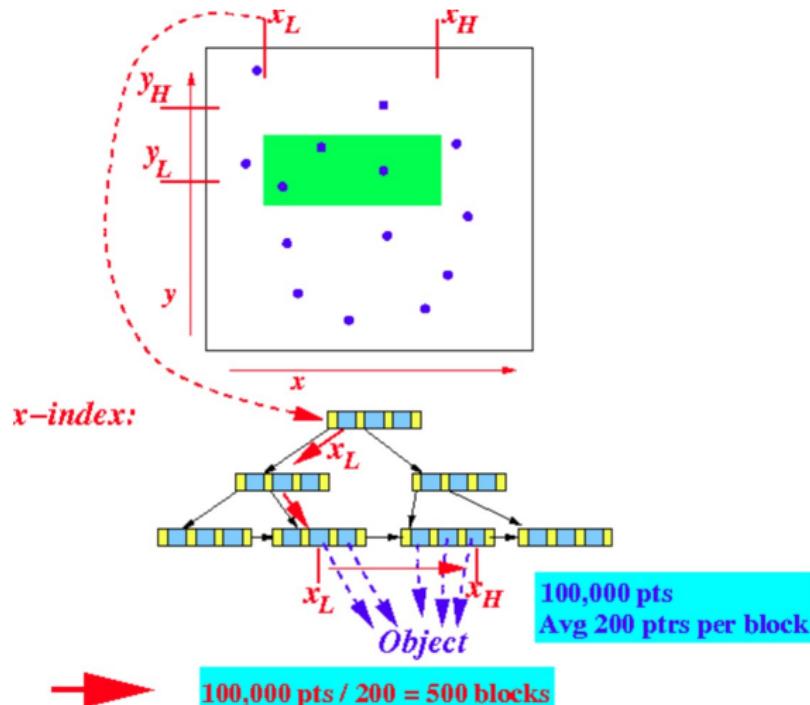
# Compute the Processing Cost

1. Use the  $x$ -B<sup>+</sup>-tree index and find the first value that is  $\geq x_L$



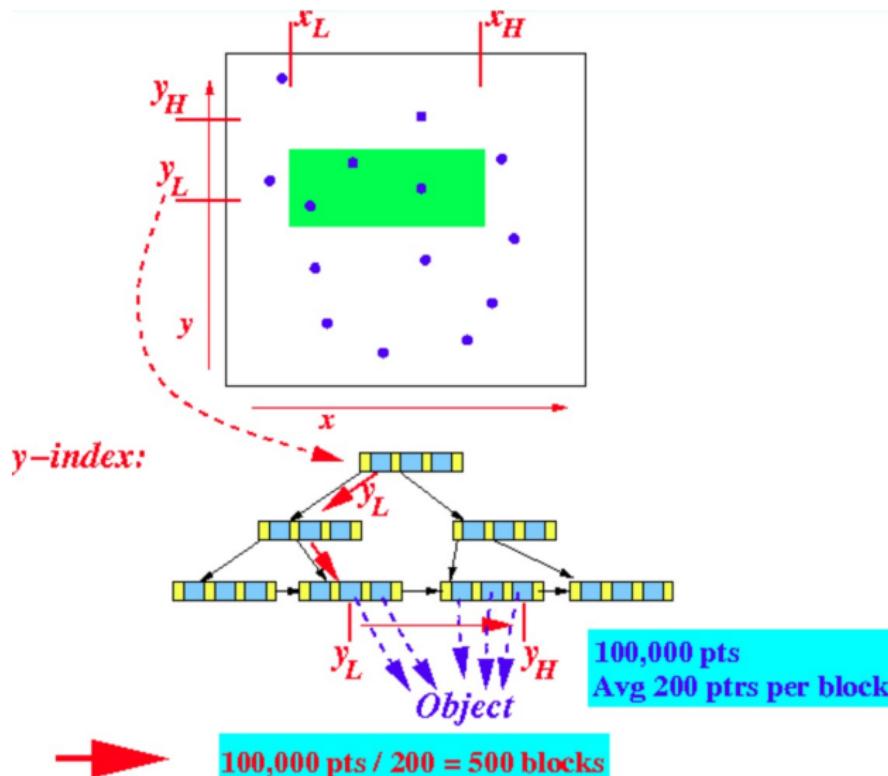
# Compute the Processing Cost

- Traverse the **leaf nodes** to find all record pointers for which  $x_L \leq x \leq x_H$



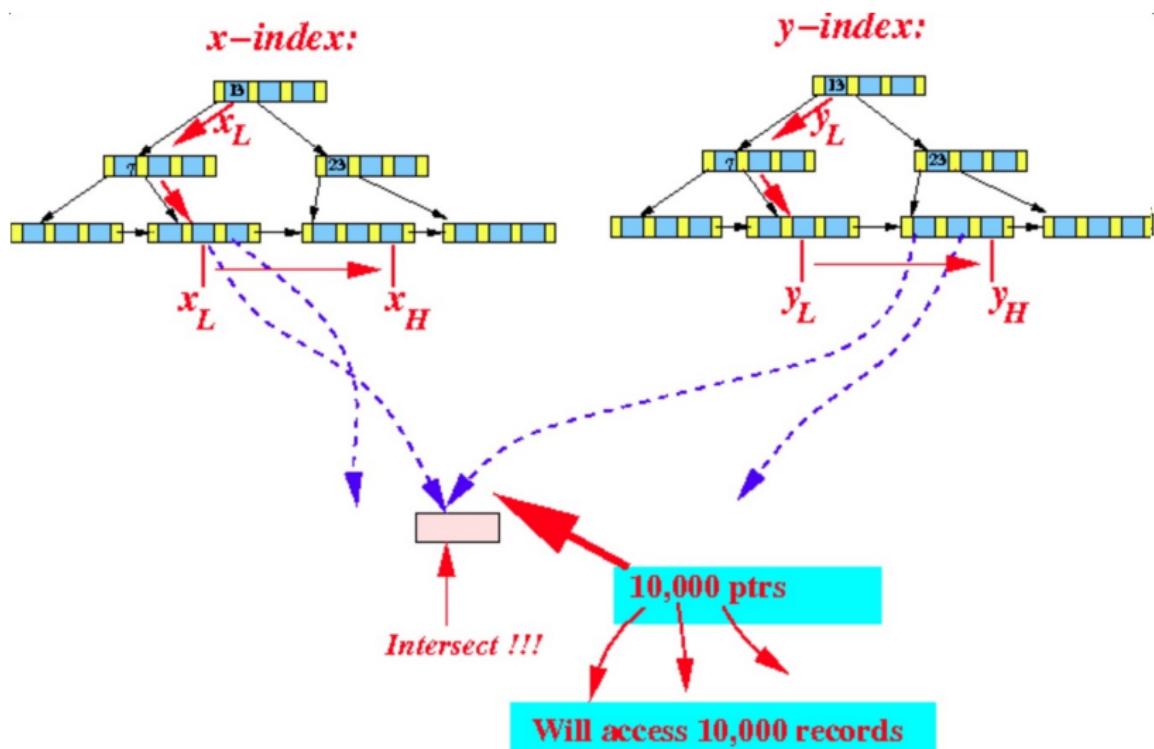
# Compute the Processing Cost

2. Do the same for the *y*-coordinate



# Compute the Processing Cost

3. Compute the intersection of the 2 pointer sets



## Compute the Processing Cost

4. Retrieve the records using the record pointers in the intersection
  - We assume the records are stored randomly (i.e., not ordered by the x or y coordinate)
  - Different records will likely be stored in different blocks
  - Accessing the 10,000 records using the record pointers will result in Accessing 10,000 data blocks
5. Total number of disk blocks accessed:  
 $500 + 500 + 10,000 = 11,000$  disk blocks

## Scan the entire relation

- Now, consider finding the points by scanning the entire relation:
  - There are 1,000,000 points
  - 1 disk block stores 100 points
  - # disk blocks used =  $\frac{1,000,000}{100} = 10,000$  blocks
- So we would need: 10,000 disk blocks accesses
- ⇒ using the B<sup>+</sup>-tree index does not help us improve performance

# Conclusion

- We cannot store geographically “related” data randomly
  - If related geographical data is stored randomly, we will need to access too many data blocks
- ⇒ must store geographically “related” data (i.e.: points that are close to each other) in the same data block
- To support the access to such a data
  - Need a more appropriate index structure for multi-dimensional data

# Multi-dimensional index structures

- Most data structures for supporting queries on multi-dimensional data fall into one of two categories:
  1. Hash like structures
    - Grid files
    - Partitioned Hashing functions
  2. Tree like structures
    - Multiple key indexes
    - kd-trees
    - Quad trees
    - R-trees

# Multi-dimensional index structures

## 1. Hash like structures

- No longer have the advantage that the answer to a query is in exactly one bucket.
- However, each of these schemes limit our search to a subset of the buckets.

## 2. Tree like structures

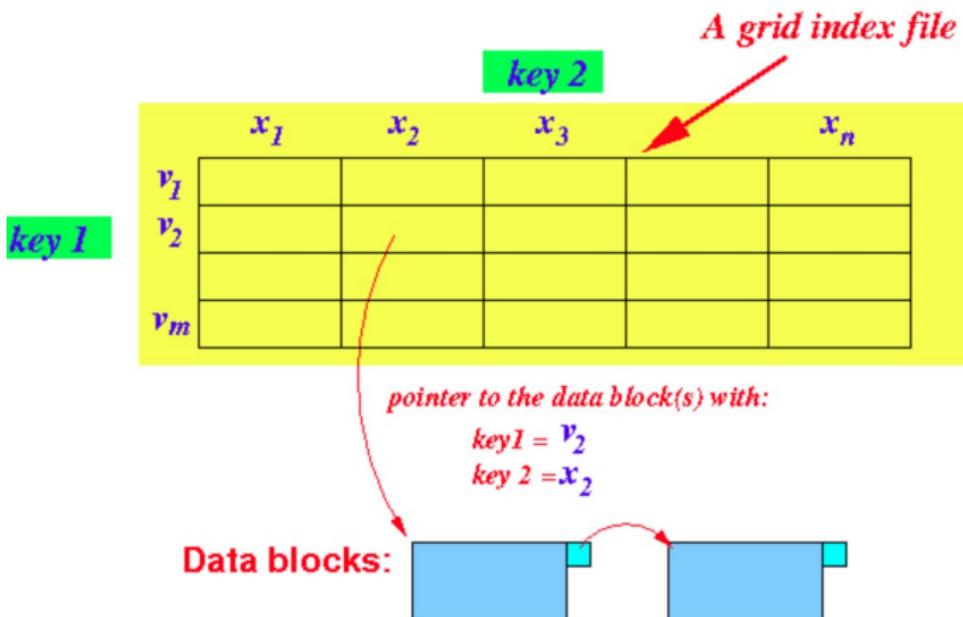
- we give up at least one of these important properties of B-trees:
  - The balance of the tree, where all leaves are at the same level.
  - The correspondence between tree nodes and disk blocks.
  - The speed with which modifications to the data may be performed

## Grid Index

- It doesn't "hash" values along the dimensions, but rather partitions the dimensions by sorting the values along that dimension
- Partition multi-dimensional space with a grid
- In each dimension, grid lines partition space into stripes
- Intersections of stripes from different dimensions define regions
- The number of grid lines in different dimensions may vary.
- Spacings between adjacent grid lines may also vary.
- Each region corresponds to a bucket.
- Attribute values for record determine region and therefore bucket

## Grid Index

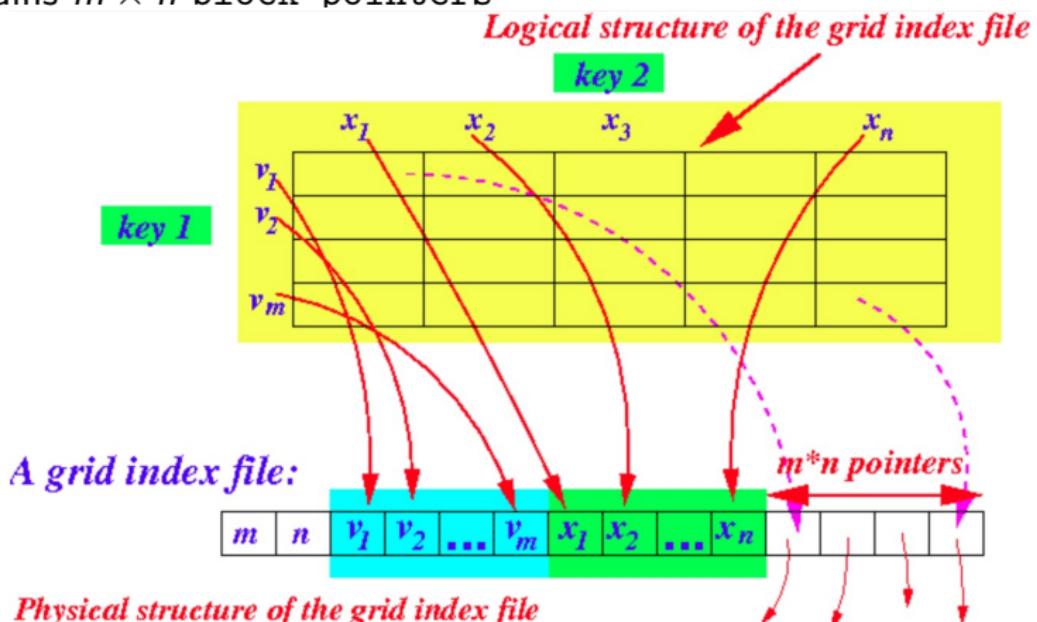
- Grid index file: an index that is organized into a 2-dimensional structure



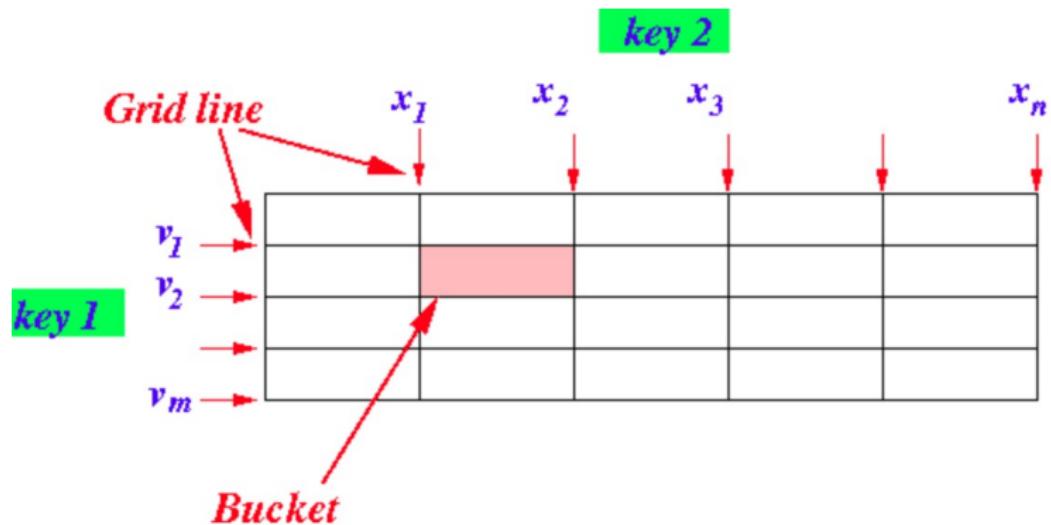
- Note: Geographically “related” data (i.e.: points that are close to each other) are stored in the same data block

# Storage Structure of Grid Index File

- 1) Stores the size parameters  $m$  and  $n$  of the grid
- 2) Stores the buckets of the grid
  - $v_1, v_2, \dots, v_m$
  - $x_1, x_2, \dots, x_n$
- 3) contains  $m \times n$  block pointers



# Buckets and Grid lines

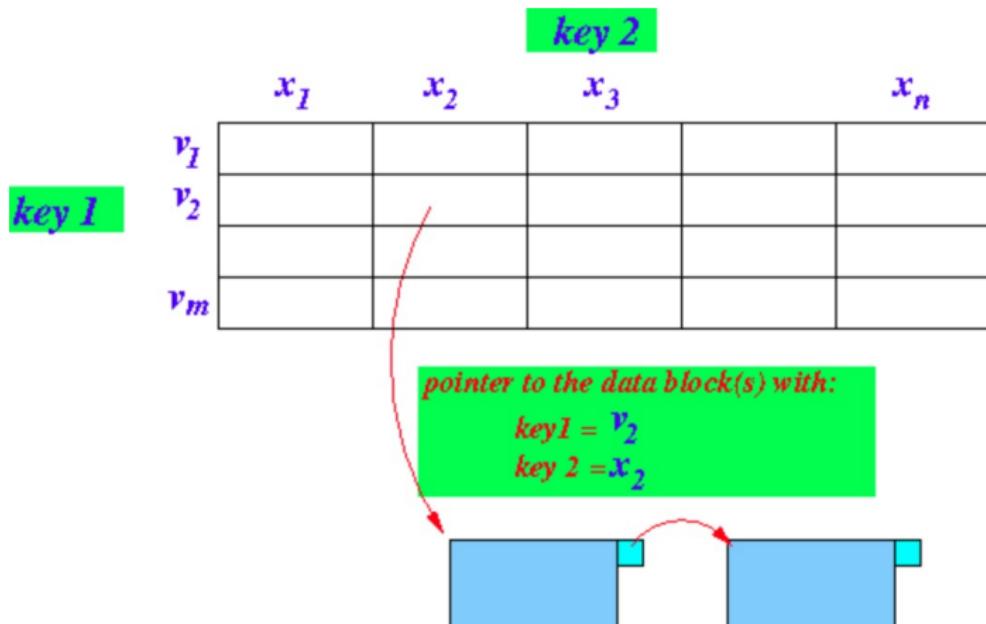


## Interpreting the grid lines

- You can interpret the values:
  - $v_1, v_2, \dots, v_m$
  - $x_1, x_2, \dots, x_n$

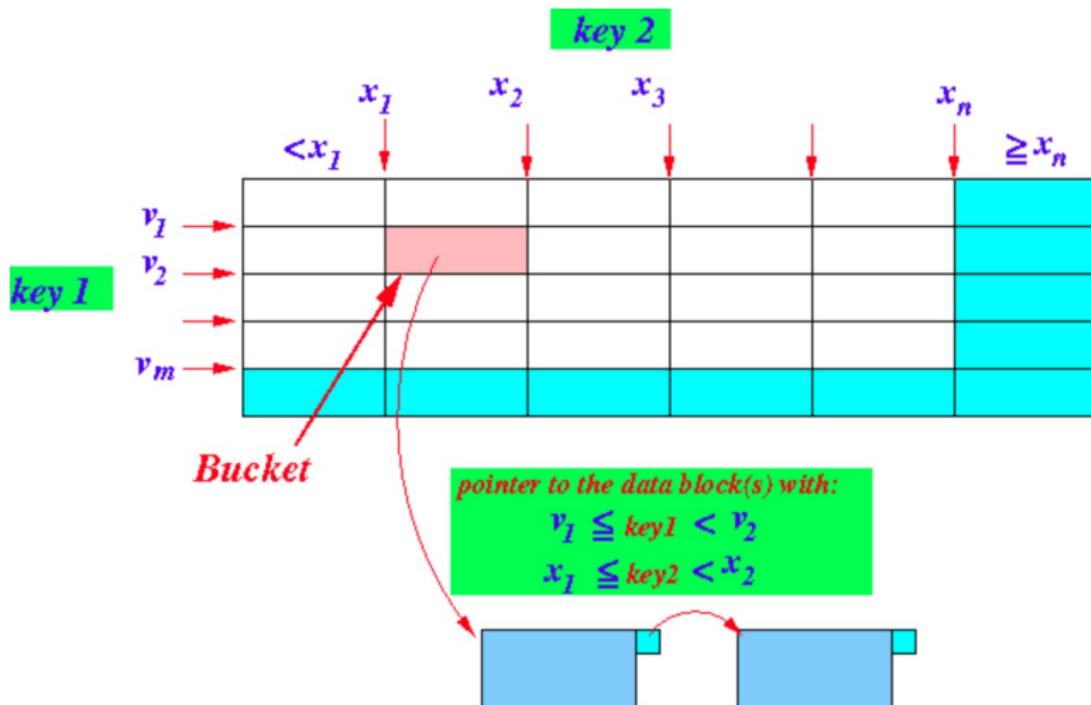
- 1) As individual points
- 2) As intervals

## Interpreting the grid lines: Point interpretation



- The grid lines represent discrete values
- With  $n$  grid lines you will have  $n$  index points

## Interpreting the grid lines: Interval interpretation



- The grid lines represents end points of intervals
- With  $n$  grid lines you will have  $n + 1$  intervals

# Example of a Grid index file

- Data on people who buy jewelry:

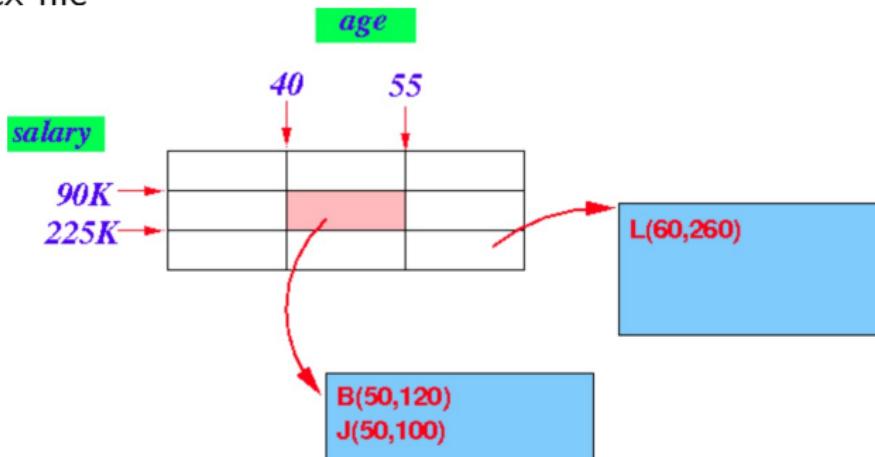
(age, salary (in \$1,000))

A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

- Ranges

Age:	0-40	40-55	$\geq$ 55+
Salary:	0-90K	90K-225K	$\geq$ 225K+

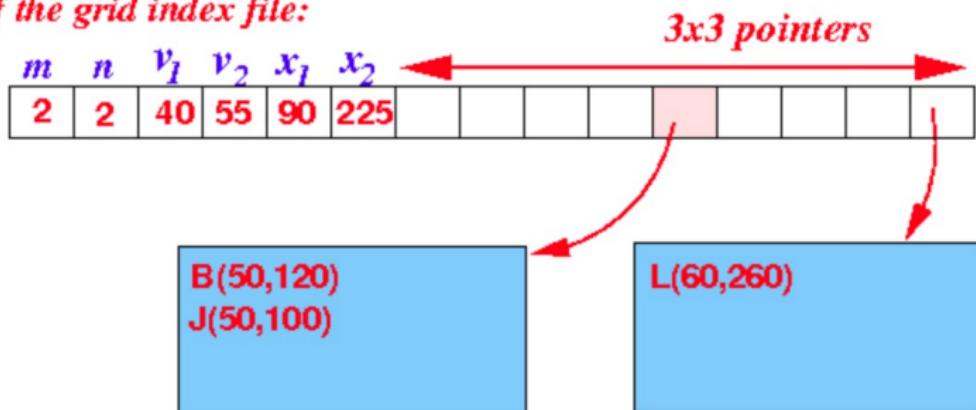
- Grid index file



## Example of a Grid index file

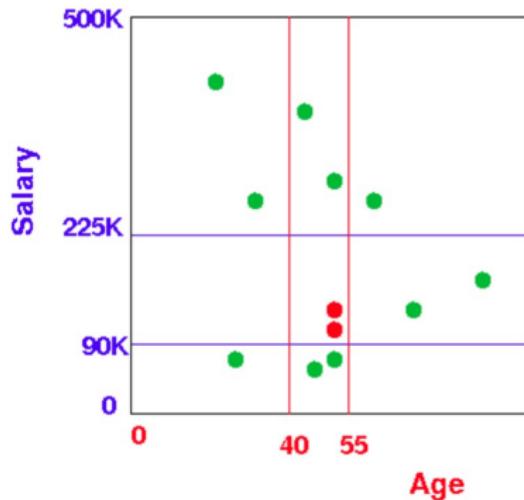
- How the grid index file is stored:

### *Storage of the grid index file:*



## Example of a Grid index file

- The text book use the following method to represent the index file



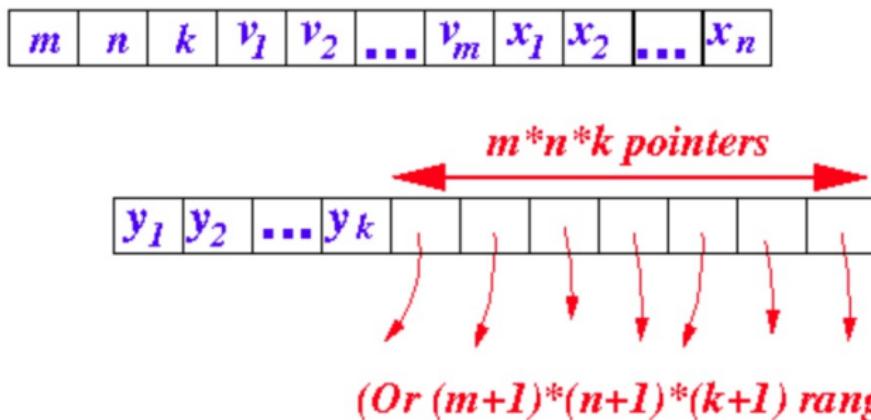
- For the following data set

(age, salary (in \$1,000))

A(25,6)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

## Generalization to higher dimensions

A 3-dimensional grid index file:



## Lookup a search key

- **Given:** search key (Age = 50, Salary = 100)
- How to find this record
  - To locate the proper bucket for a point, we need to know, for each dimension, the list of values at which the grid lines occur.
  - Look at each component of the point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.

# Insert a new record in a Grid Index file

---

## Algorithm 1 insert( record )

---

- 1: Lookup (record.SearchKey)
  - 2: Let  $b$  = the last bucket block
  - 3: **if**  $b$  has room for record **then**
  - 4:     Insert record in block  $b$
  - 5: **else**
  - 6:     Allocate an overflow block for bucket
  - 7:     Link overflow block to  $b$
  - 8:     Insert record in overflow bucket block
  - 9: **end if**
-

## Performance Analysis: lookup/insert a search key

- Assumption: The grid index file can be stored in memory
- Lookup performance
  - 0 block access to obtain the bucket block pointer
  - 1 block access to obtain the data block (that contains the record)
  - If there are overflow blocks, need to access a few more (overflow) blocks

## Performance Analysis: lookup/insert a search key

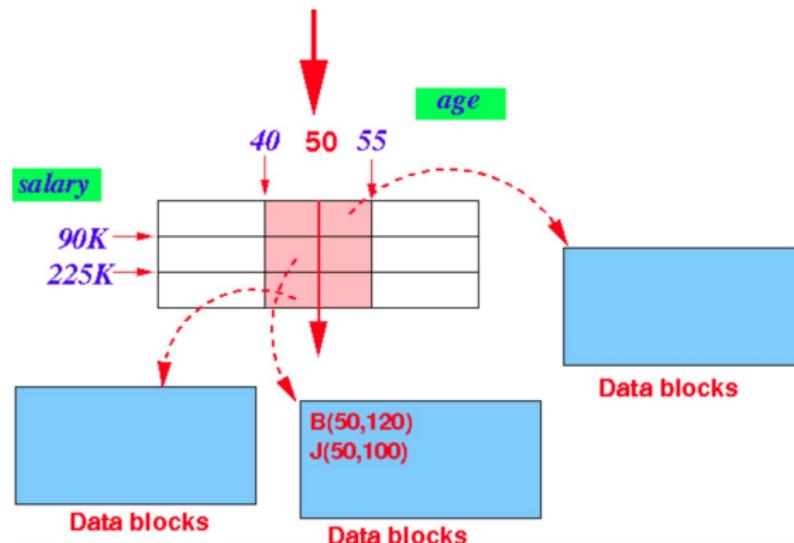
- Assumption: The grid index file can be stored in memory
- Insert performance
  - In addition to the lookup cost
  - 1 more block write operation to update the bucket block
  - If overflow, need to update the overflow link in the bucket and write an overflow block

## Using a grid index in multi-dimensional queries

- Performance of Grid index for the commonly used multi-dimensional queries
- Assumption: The grid index file can be stored entirely in memory

# 1) Partial Match queries

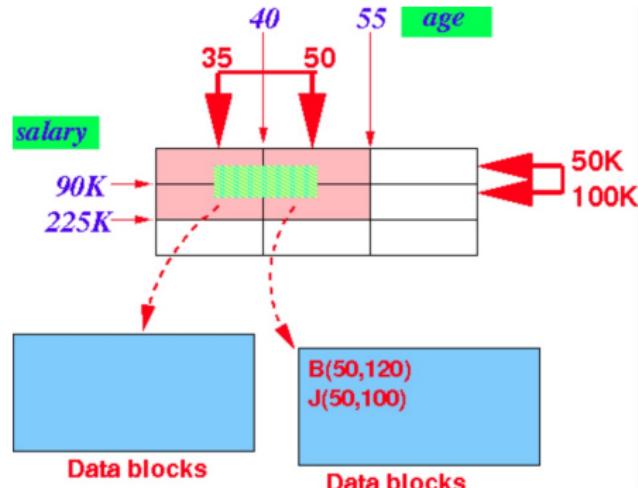
- The query specifies conditions on some dimensions but not on all dimensions
- Find all jewelry purchases by people with age = 50



- You will access  $m$  disk blocks ( $m$  is some dimension of the grid)

## 2) Range queries

- Find objects that are located either partial or wholly within a certain range
- Find all jewelry purchases by people whose  $35 \leq \text{age} \leq 50$ ,  $50K \leq \text{salary} \leq 100K$



- In this example, we must access 4 disk blocks

### 3) Nearest neighbor queries

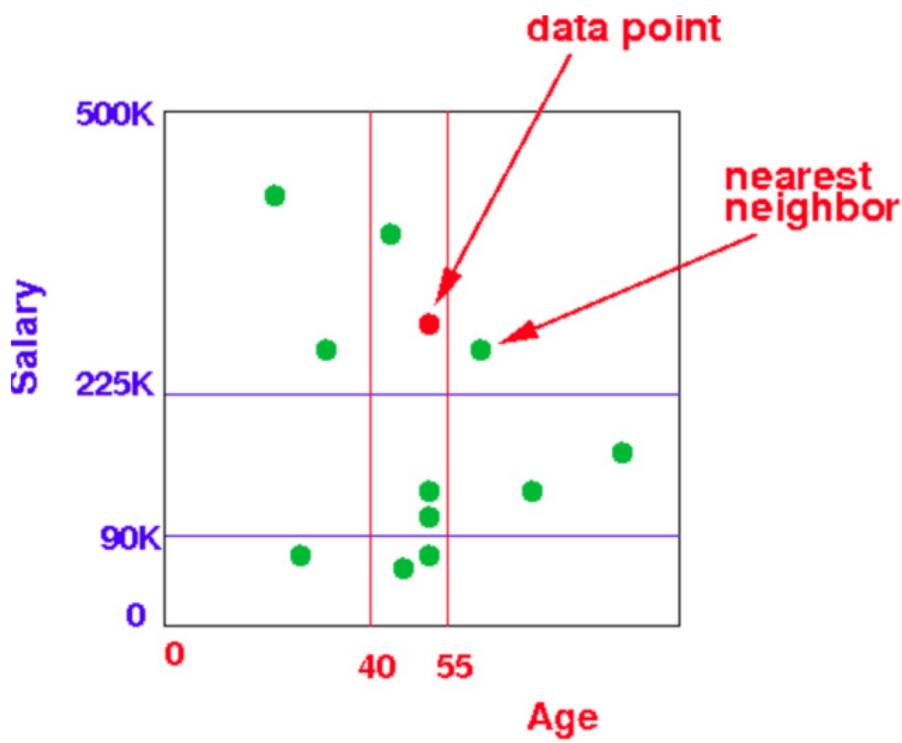
- Suppose we have a relation containing points on a map
- Each point is stored in the following relation: Point(x,y)
- Given a point  $p_0(10,20)$
- Query: Find the point that is closest to point  $p_0(10,20)$

```
SELECT * FROM Point P
/* P is the closest point to p0 */
WHERE NOT EXISTS
    (SELECT * FROM Point Q
     /* A closer point Q cannot exist ! */
     WHERE "distance(Q,p0)<distance(P,p0)" )

SELECT * FROM Point P
/* P is the closest point to p0 */
WHERE NOT EXISTS
    (SELECT * FROM Point Q
     /* A closer point Q cannot exist ! */
     WHERE (Q.x-10)^2+(Q.y-20)^2<(P.x-10)^2+(P.y-20)^2)
```

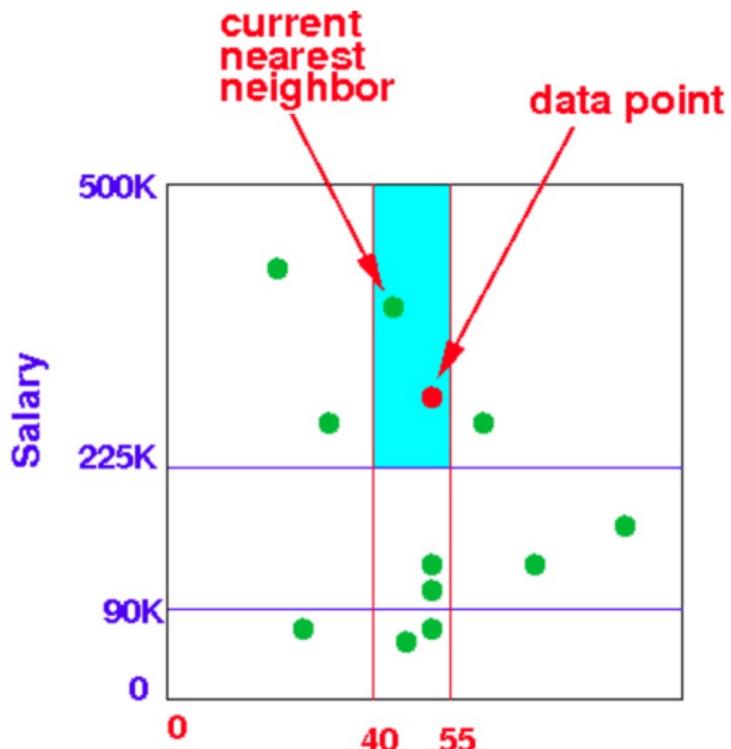
### 3) Nearest neighbor queries

- Find the nearest neighbor of a data point



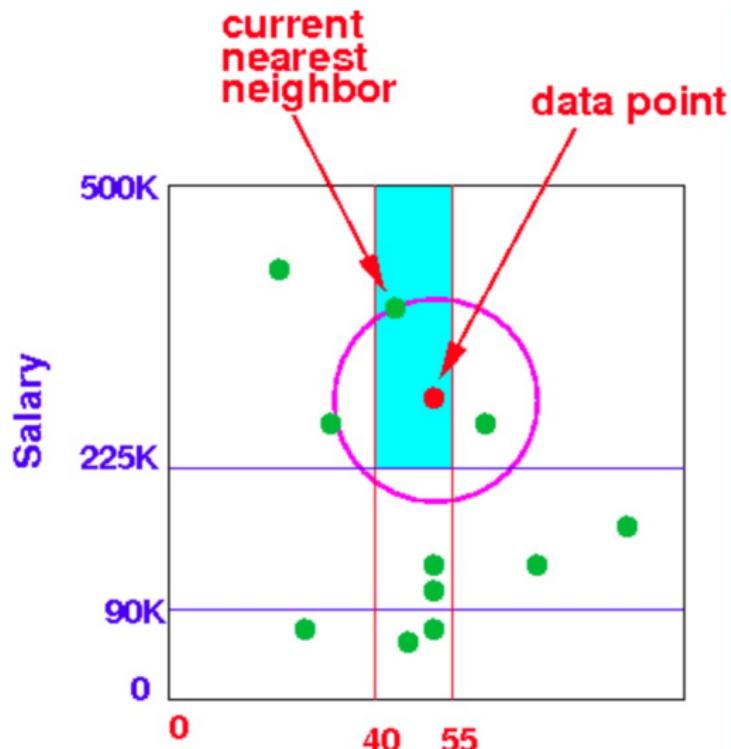
### 3) Nearest neighbor queries

- Start by finding the nearest neighbor in the bucket that contains the data point



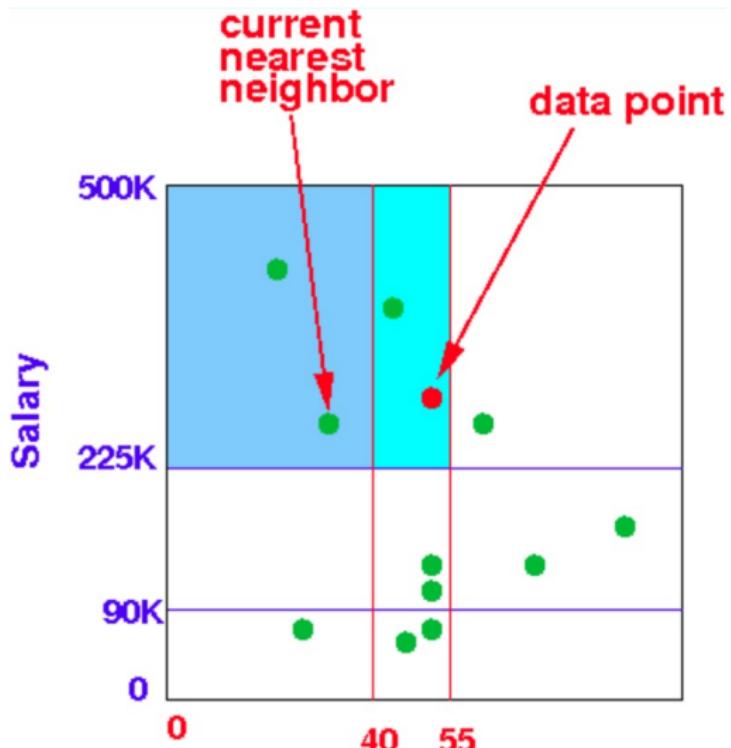
### 3) Nearest neighbor queries

- This distance will limit the block where you need to search to all blocks that intersect with this circle:



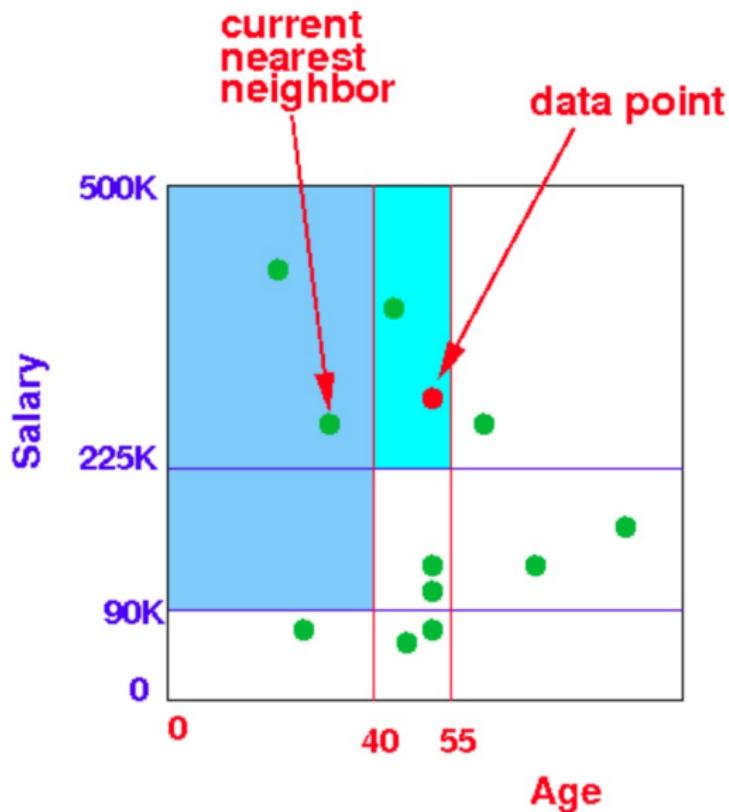
### 3) Nearest neighbor queries

- Expand the search region in an adjacent bucket that contained within the circle:



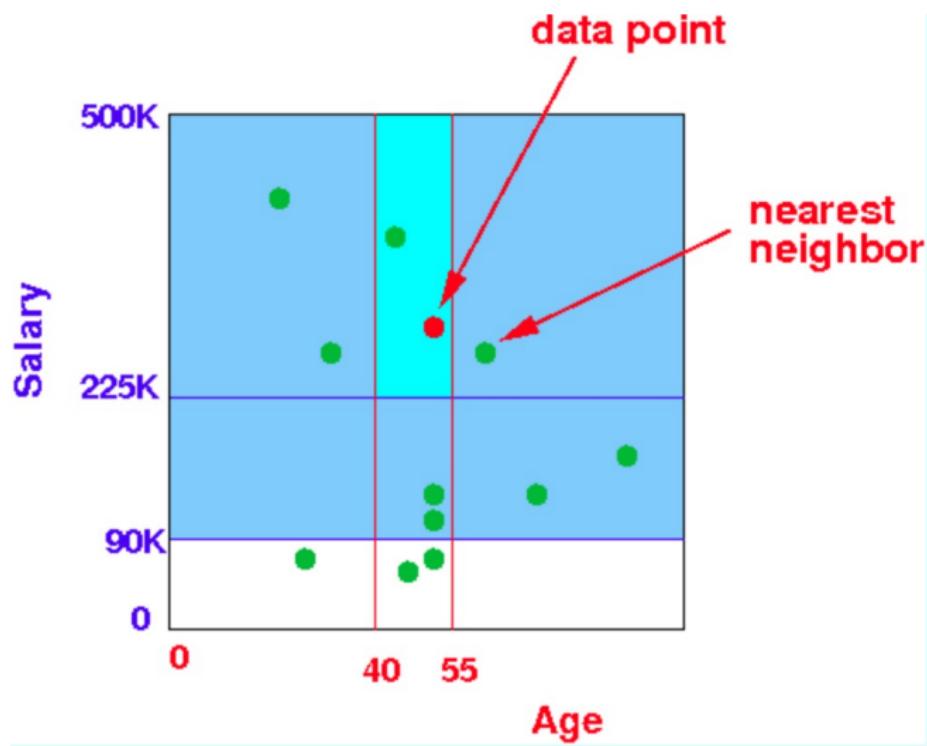
### 3) Nearest neighbor queries

- And so forth



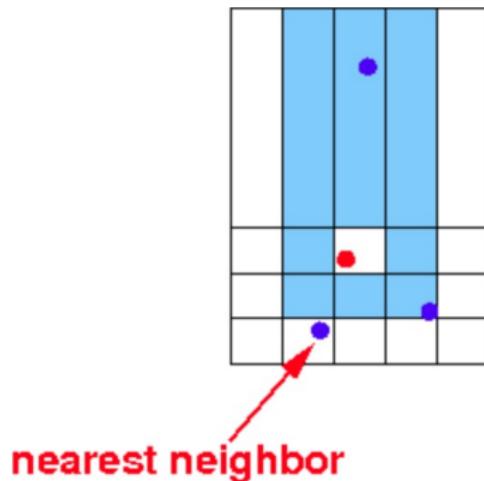
### 3) Nearest neighbor queries

- And so forth



### 3) Nearest neighbor queries

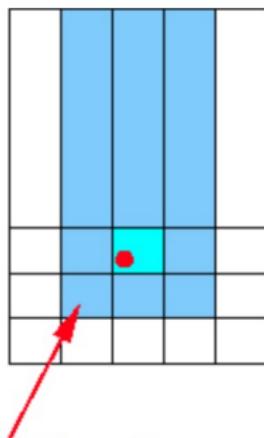
- Note: You may need to expand the search range beyond the adjacent regions



- The nearest neighbor is outside the adjacent regions
- You must use the current nearest neighbor and the grid lines to decide whether you need to expand the range of the search

### 3) Nearest neighbor queries: Performance

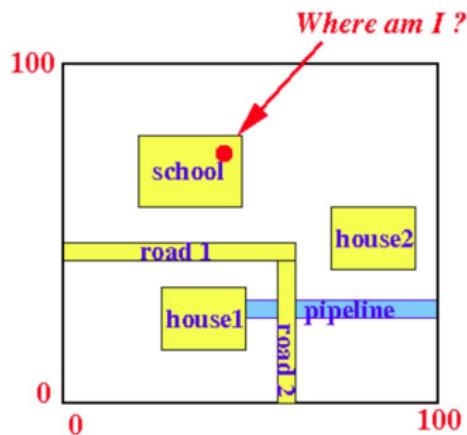
- The expanding range search will access on average 9 data blocks (in a 2-dimensional grid index)



nearest neighbor is usually found in this region

## 4) Where-am-I queries

- Given a location (i.e., coordinate)
- Find the object(s) that contains the location



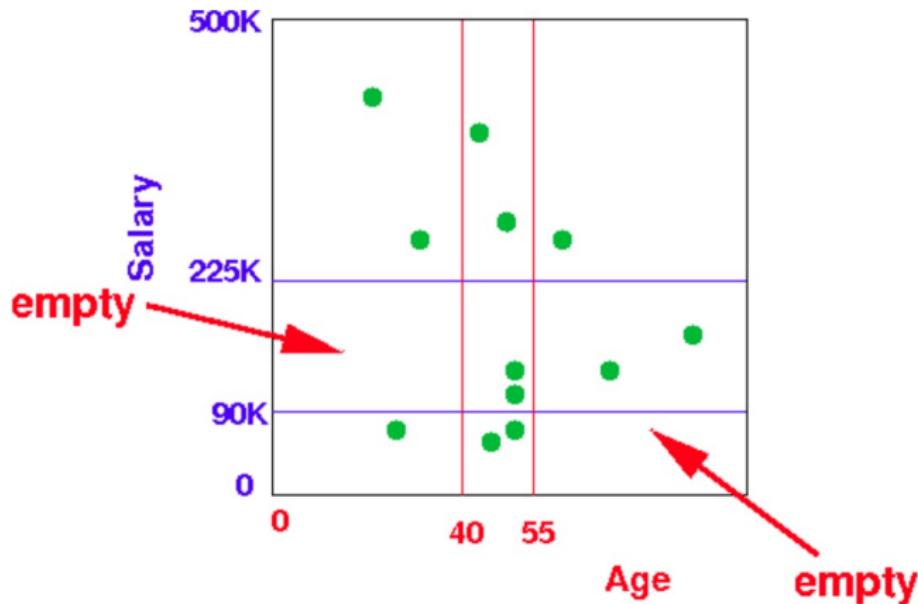
- Grid index cannot represent objects (can only represent points)
- ⇒ Grid Index cannot handle Where-am-I type of queries
- The only kind of index that can handle Where-am-I queries is the R-tree (Region-tree) (Discussed later)

## Grid Index: Summary

- + Good for multiple-key search
- Space, management overhead (nothing is free)
- Need partitioning ranges that evenly split keys

## Grid Index

- A major problem with Grid Index files is Poor occupancy rate at many grid buckets



- Especially when you have 3 or more dimensions. You will have many buckets that are empty.

# Multi-dimensional index structures

- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

# Partitioned Hashing

- Traditional hashing



- Problem with traditional hashing

- If the key is composite

$$K = (x, y)$$



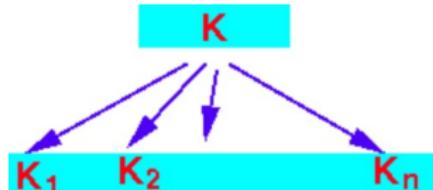
- and some component of the key is not known



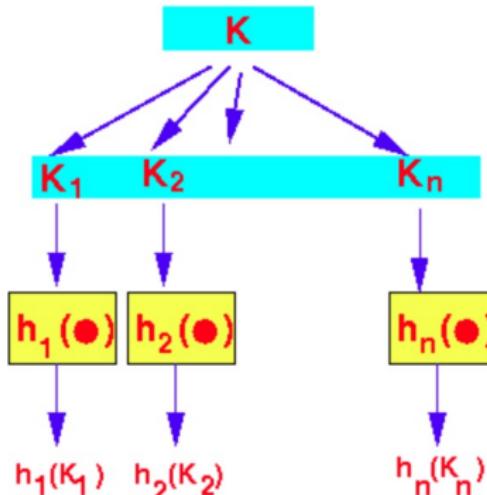
- we cannot compute a meaningful hash value at all

# Partitioned Hashing

- Partitioned Hashing
  - The key is a composite:

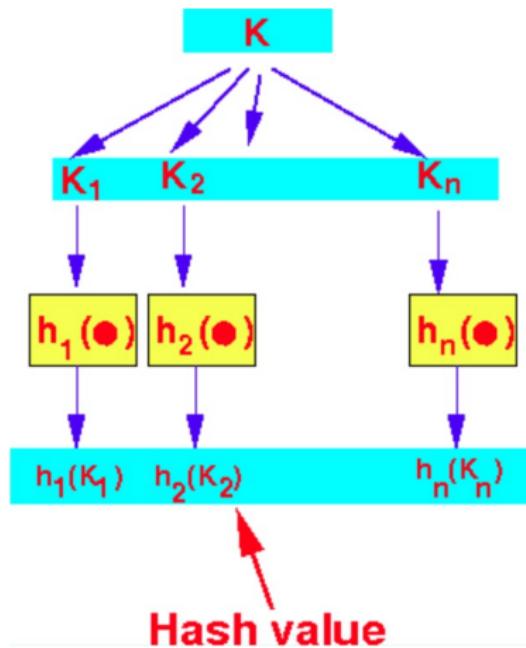


- Use  $n$  hash functions, one function on one component



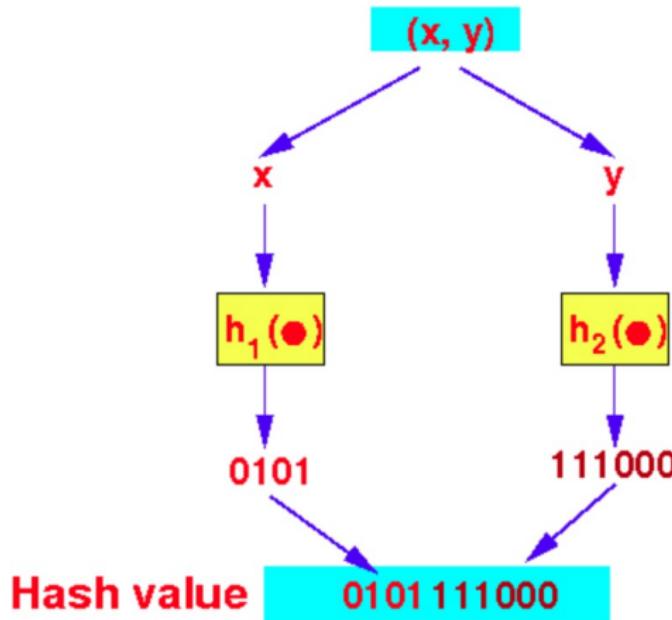
# Partitioned Hashing

- Partitioned Hashing
  - The hash value is the concatenation of the individual hash function values



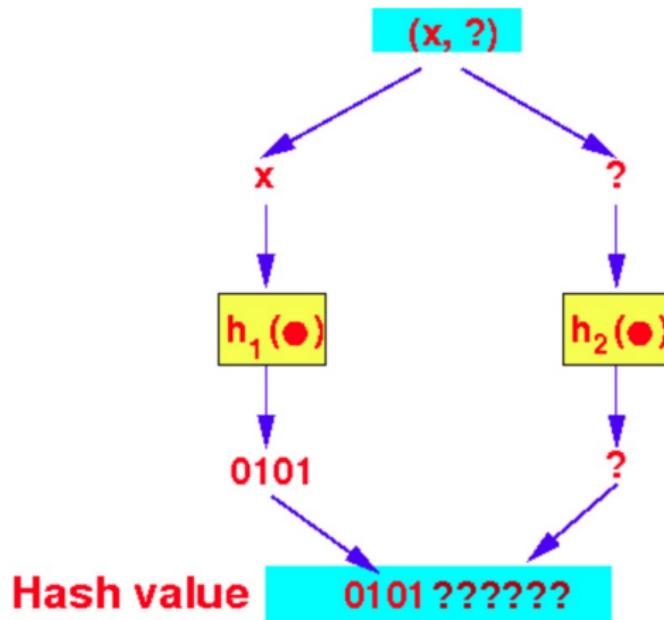
## Partitioned Hashing: Example

- Assume: a hash table with 10-bit bucket numbers (1024 buckets)
- Devote four bits to attribute  $x$  and remaining six bits to attribute  $y$ .



# Advantage of Partitioned Hashing

- Partitioned Hashing can generate a meaningful hash value for incomplete keys



# Partitioned Hashing: A complete example

- Data on people who buy jewelry

(age, salary (in \$1,000))

A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

- Given hash functions

$$\text{Age: } h_1(\text{ age }) = \text{ age \% 2}$$

$$\text{Salary: } h_2(\text{ salary }) = \text{ salary \% 4}$$

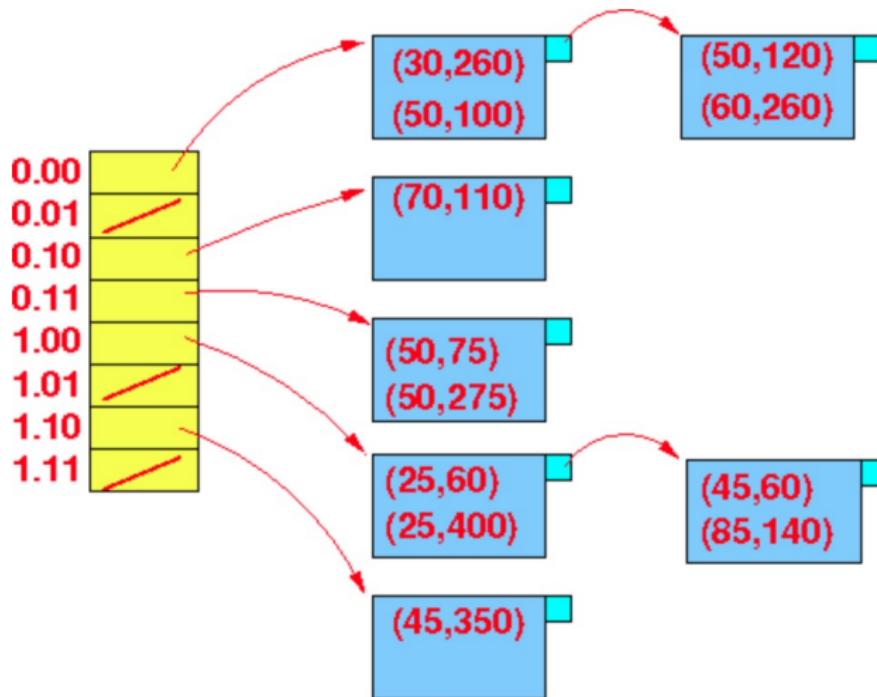
- Some Hash Function values

A(25,60)	D(45,60)	G(50,75)	J(50,100)
V	V	V	V
100	100	011	000

B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

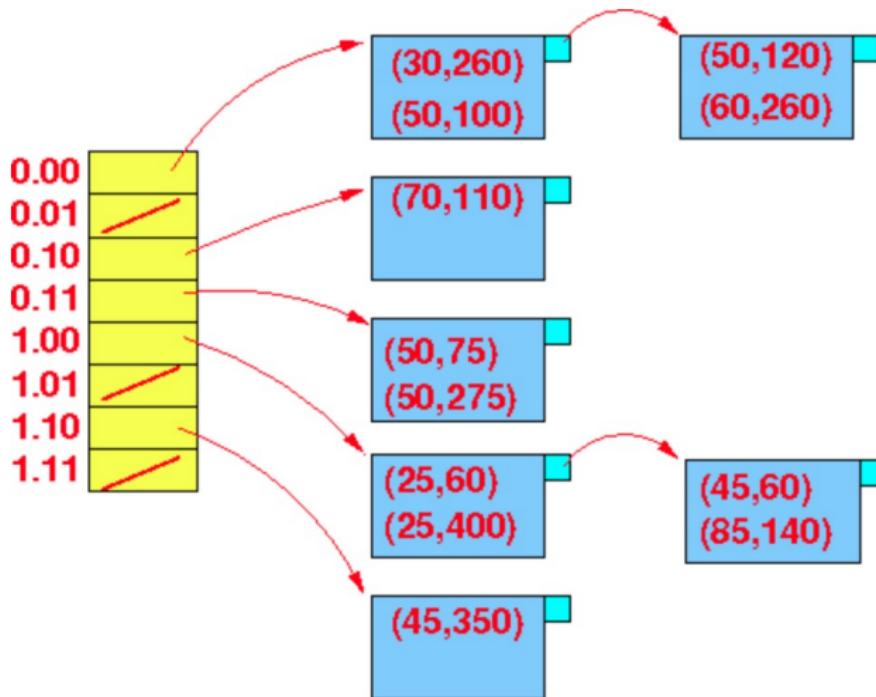
# Partitioned Hashing: A complete example

- The Partitioned Hash index



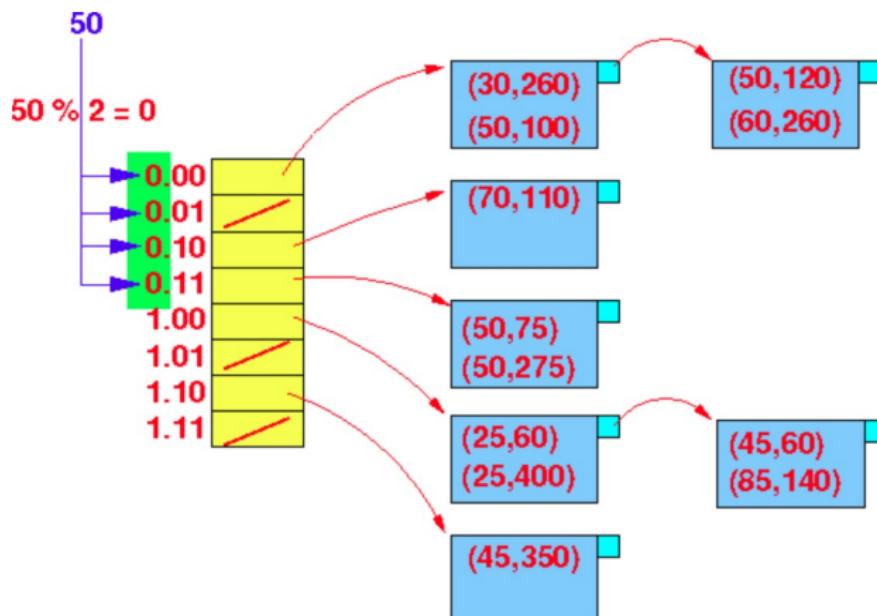
# Using a Partitioned Hashing

- The Partitioned Hash index



# 1) Partial Match queries

- Find people with age = 50



- Age = 50 will hash to the hash value Hash(age) = 0xy
- Start at bucket 000 and scan to bucket 011

## 2) Range queries

- Find objects that are located either partial or wholly within a certain range
- Find people such that:  $35 \leq \text{age} \leq 50$ ,  $50K \leq \text{salary} \leq 100K$

## 2) Range queries

- a) Hash **all** values inside the range

```
hash(35, 50K)    --> block pointer 1  
hash(36, 50K)    --> block pointer 2  
....  
hash(50, 50K)
```

And so on:

(35, 55K) (36, 55K), .... (50, 55K)

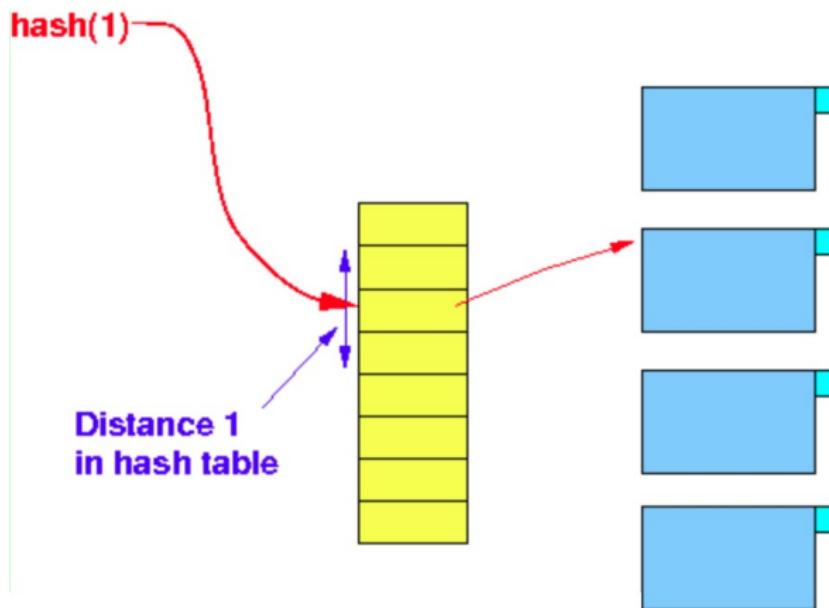
...

(35, 100K) (36, 100K), .... (50, 100K)

- Note: the block pointers can have duplicates
  - b) Collect all the buckets (eliminate duplicate block pointers)
  - c) Access all (unique) buckets (disk blocks)
- 
- ⇒ Hashing is not appropriate for range type queries

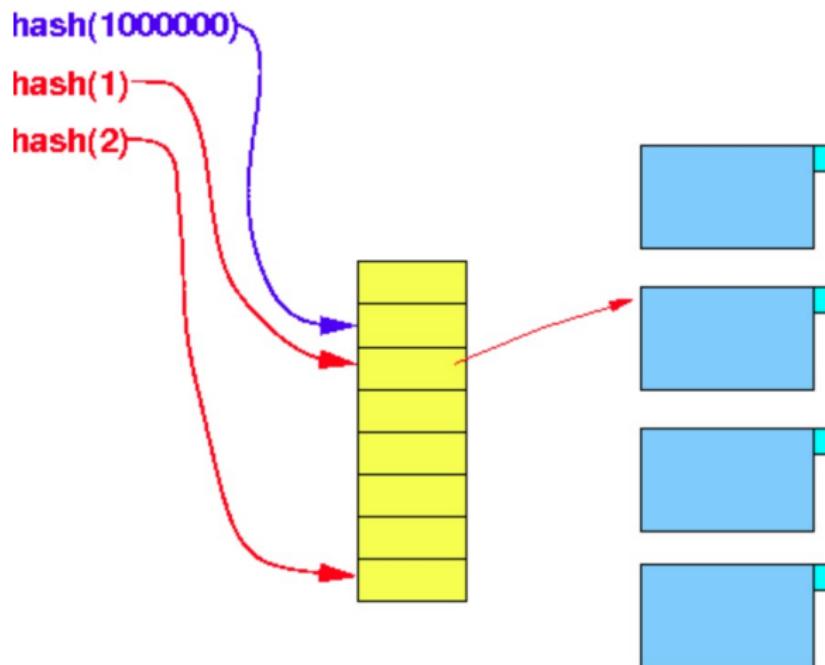
### 3) Nearest neighbor queries

- Hashing is completely useless for nearest neighbor type queries
- Because: There is no notion of distance in the hash function
- Example: find records that with distance  $\leq 1$  to search key = 1
  - We hash the search key 1



### 3) Nearest neighbor queries

- However, we cannot use the distance in the hash table to locate “nearby” objects (records)



- The value 2 is near the value 1, but may get hash very far away

## Property of hashing:

- Closeness of bucket indexes has nothing to do with real distance between data points (because hashing computes a random number)

## 4) Where-am-I queries

- Hashing is also not useful here either
- Because hashing provide no information on distance

## Advantage of Partitioned Hashing

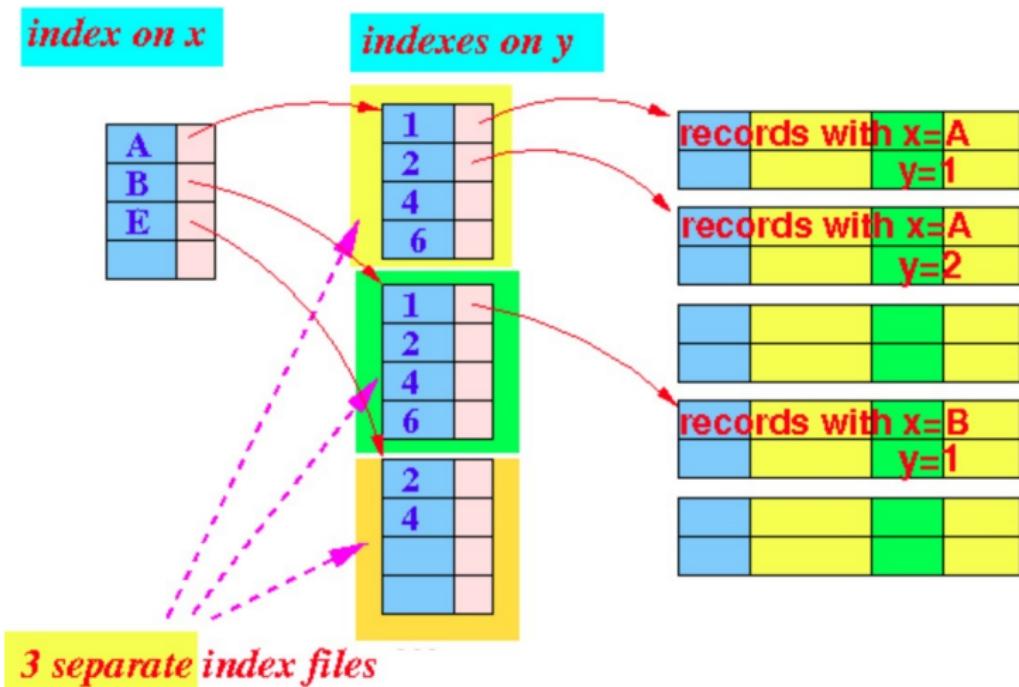
- Good hash functions will randomize the records
- ⇒ Partitioned hashing will achieve good occupancy rate per bucket

# Multi-dimensional index structures

- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

# Multiple-key index

- special case of a multilevel index using different types of search keys in each level



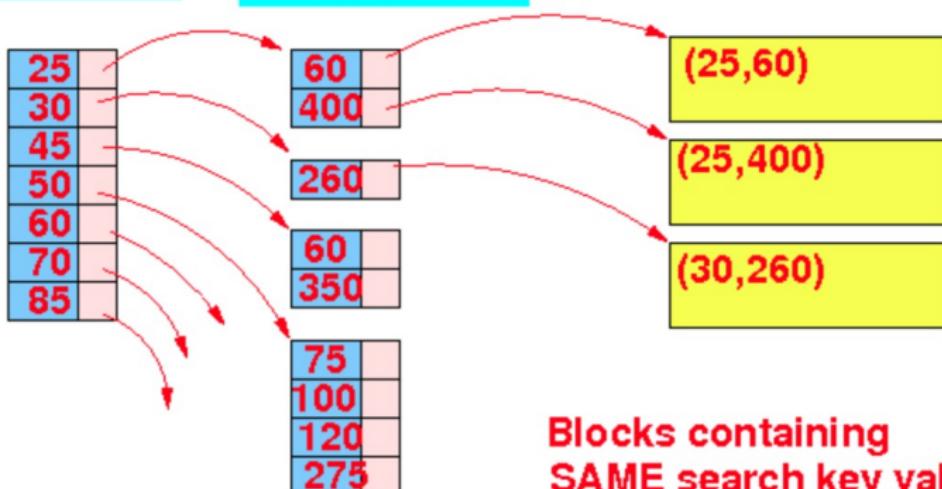
# Multiple-key index: Example

- Data on people who buy jewelry

(age, salary (in \$1,000))			
A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

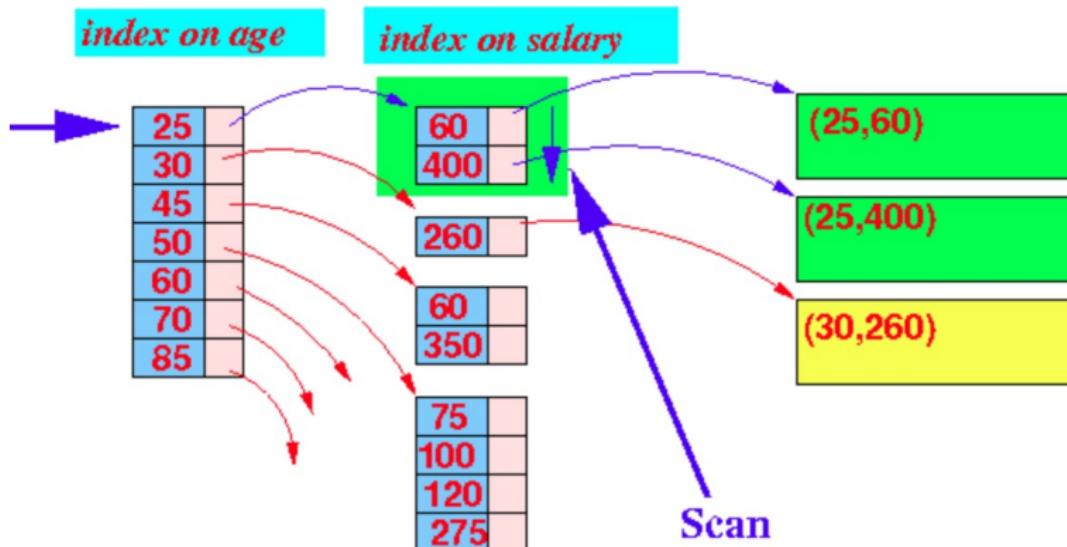
- A multiple-key index on keys (age, salary)

*index on age*      *index on salary*



## Using a Multiple-key index: 1) Partial Match queries

- Find all people with age = 25



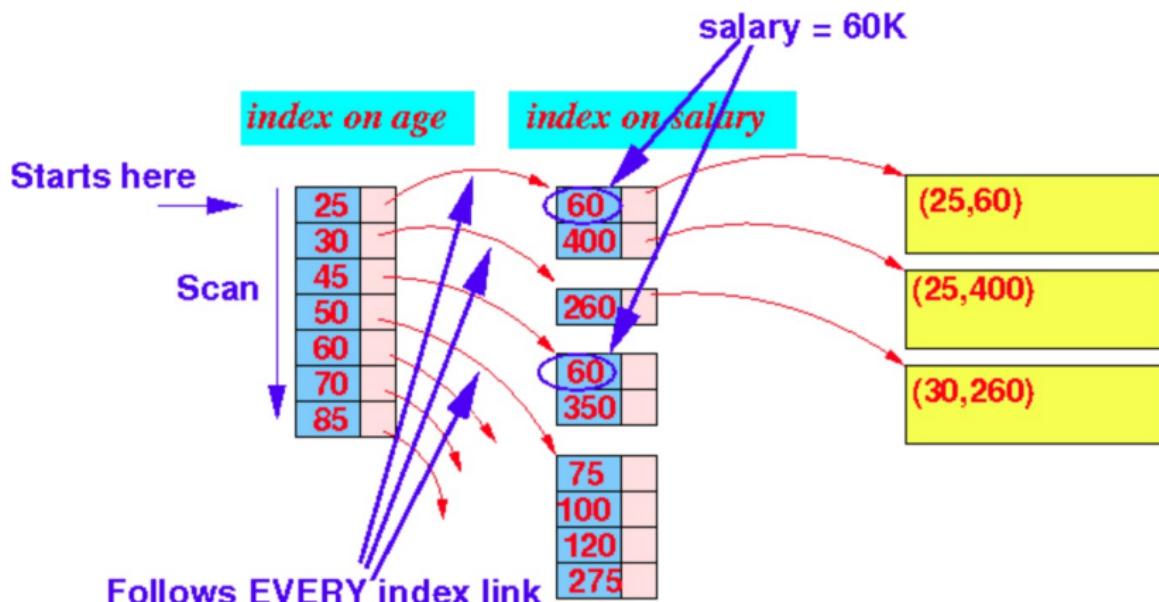
- Use the index on age to find the index block(s) for age = 25
- Then, scan all entries in the salary index file (list of blocks) indexed by age= 25 to find the records

## 1) Partial Match queries

- Multiple-key index for partial match query will only be useful when the first dimension is given
- We cannot use multiple-key index to process the following query efficiently:
  - Find all people who earn \$60,000 who buy jewelry

# 1) Partial Match queries

- Find all people who earn \$60,000 who buy jewelry. We will need to scan the first index



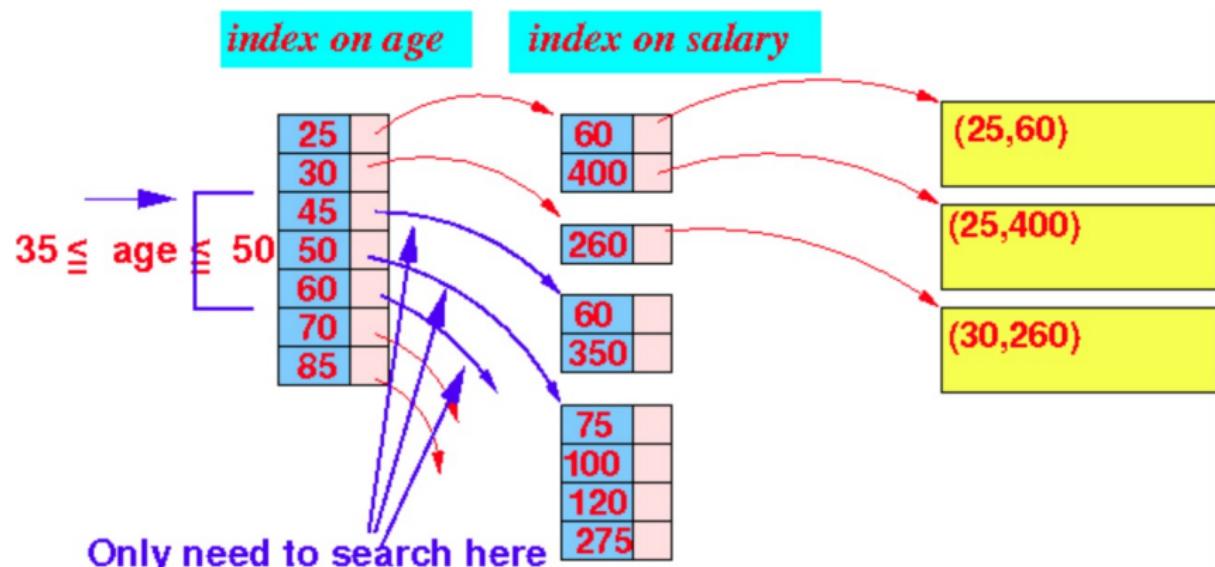
- Result: many disk accesses

## 2) Range queries

- Find objects that are located either partial or wholly within a certain range
- Find people such that:  $35 \leq \text{age} \leq 50$ ,  $50K \leq \text{salary} \leq 100K$

## 2) Range queries

- Use the range of age to find all of the subindexes that might contain answer



- Only need to search a limited number of lower level index files

### 3) Nearest neighbor queries

- The multiple key index can help in the processing of Nearest neighbor queries
- BUT: It involves a complicated expanding range search algorithm in “nearby branches” of the index tree

## 4) Where-am-I queries

- Multiple-key index are not used in Where-am-I queries

# Multi-dimensional index structures

- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

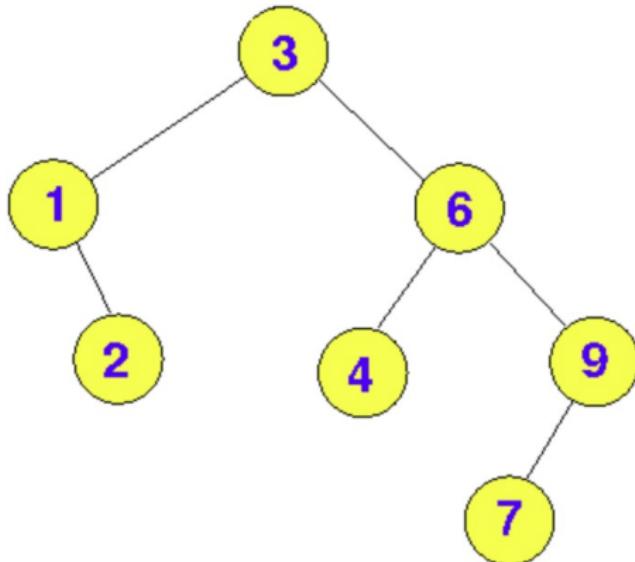
## kd (k-dimensional) tree:

- The kd-tree as a main memory data structure
- Adaptation of the kd-tree for disk storage

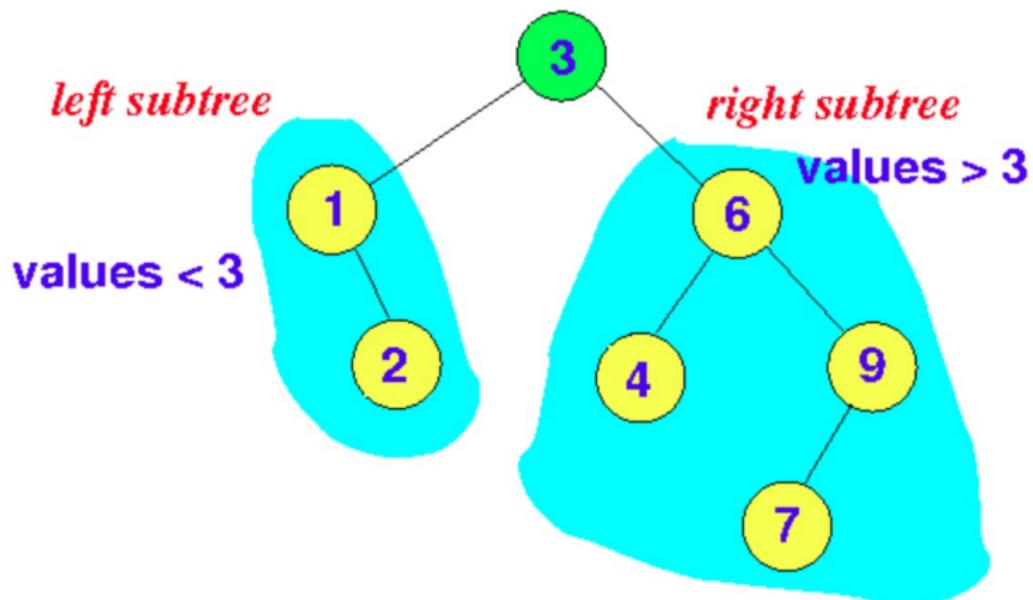
## Review: Binary Search Tree

- Binary Search Tree (BST) is a binary tree where
  - The values in the nodes in the left subtree of the node  $x$  in the tree has a smaller value than  $x$
  - The values in the nodes in the right subtree of the node  $x$  in the tree has a greater value than  $x$
- Notice the above property holds for every node in the binary tree

## Review: Binary Search Tree: Example



## Review: Binary Search Tree: Example

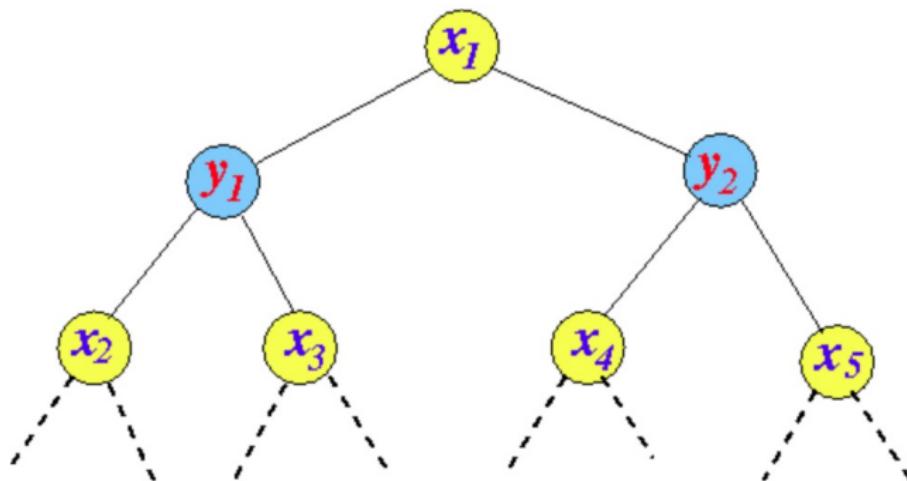


# The kd-tree

- The kd-tree is a generalization of the classic Binary Search Tree (BST)
- The search key used at different levels belongs to a different dimension (domain)
- The dimensions at different levels will wrap around (i.e., circulate)

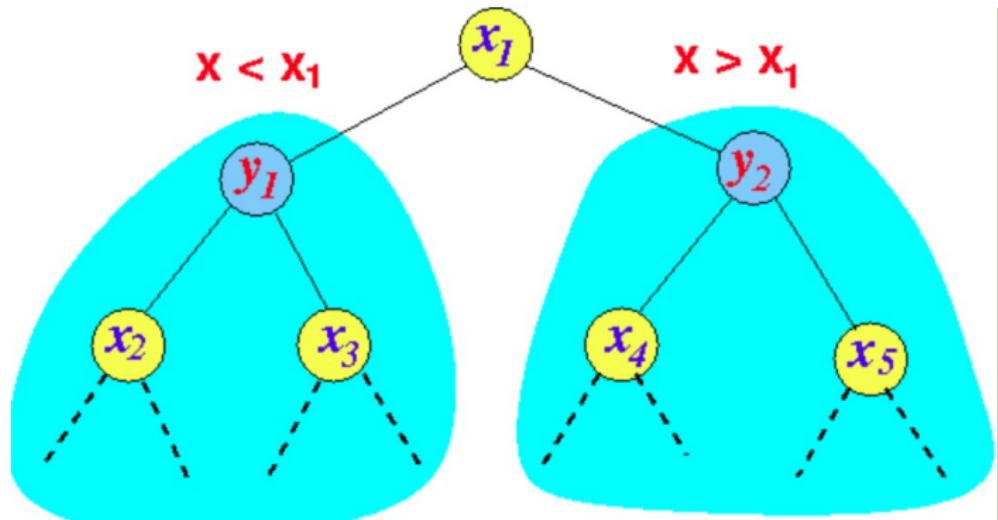
## Example: a 2-dimensional kd-tree

- 2 dimensions: x and y



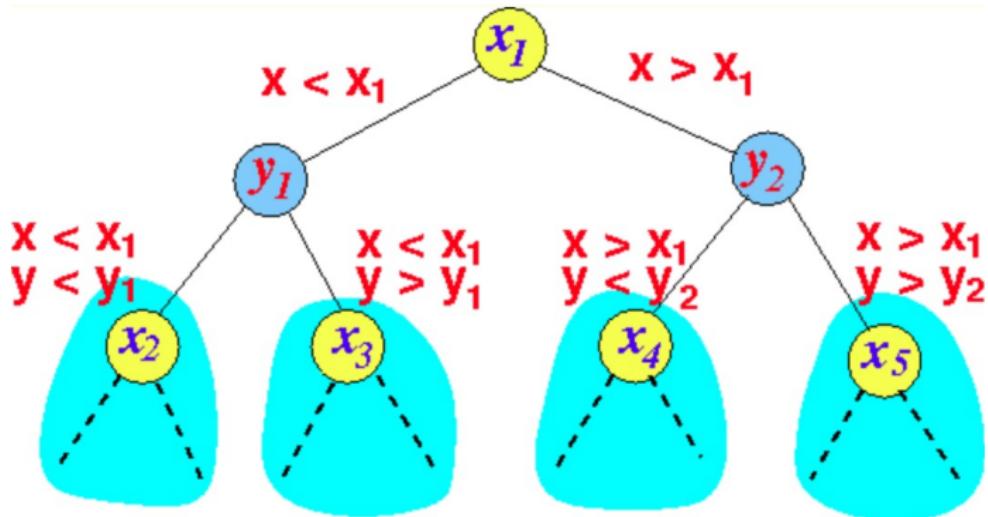
# Properties

- Subtrees of  $x_1$  must satisfy this property



# Properties

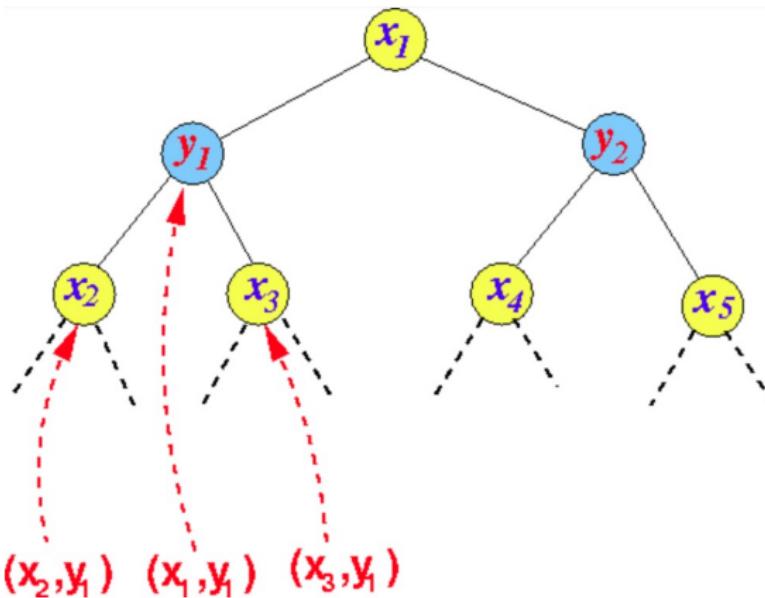
- Subtrees of  $y_1$  and  $y_2$  must satisfy this property



- And so on (for every level of the kd-tree)

## Classical kd-tree

- The actual record (data) are stored in every node (search key) of the kd-tree



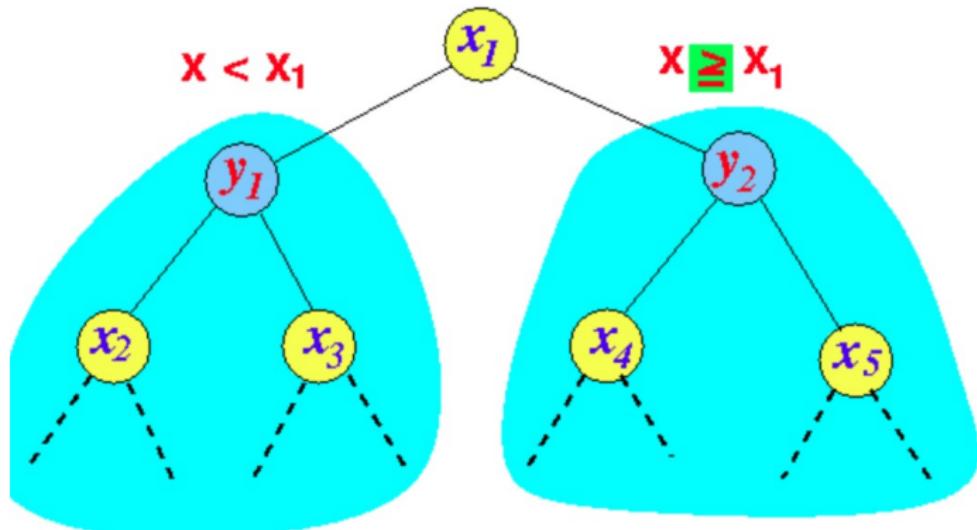
- The node  $y_1$  contains the data (record) for  $(x_1, y_1)$
- The node  $x_2$  contains the data (record) for  $(x_2, y_1)$
- And so on

## Modifications to the kd-tree for storage on disk

- Interior nodes do not store data
- Interior node only stores
  - Attribute name (i.e.:X or Y)
  - Dividing value (i.e.:  $x_1$  or  $y_4$ ) of the attribute
  - Pointers to the (2) children nodes

## Modifications to the kd-tree for storage on disk

- Dividing line is “moved” a little bit



- The equality is included in right branch of the kd-tree
- Each leaf node of the modified kd-tree is one (1) data block

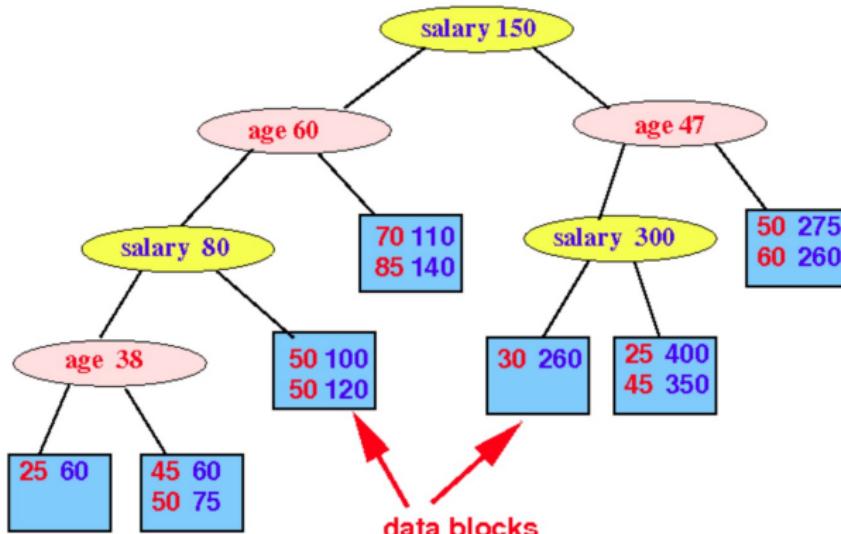
## Example kd-tree

- Data on people who buy jewelry

(age, salary (in \$1,000))

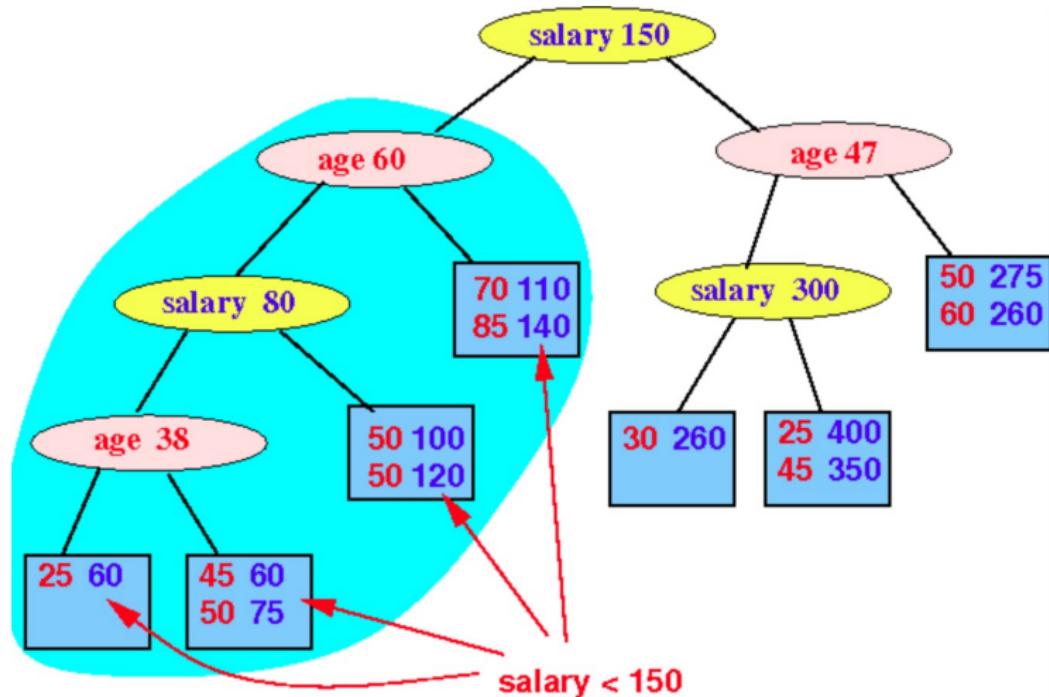
A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

- A kd-tree for the data:



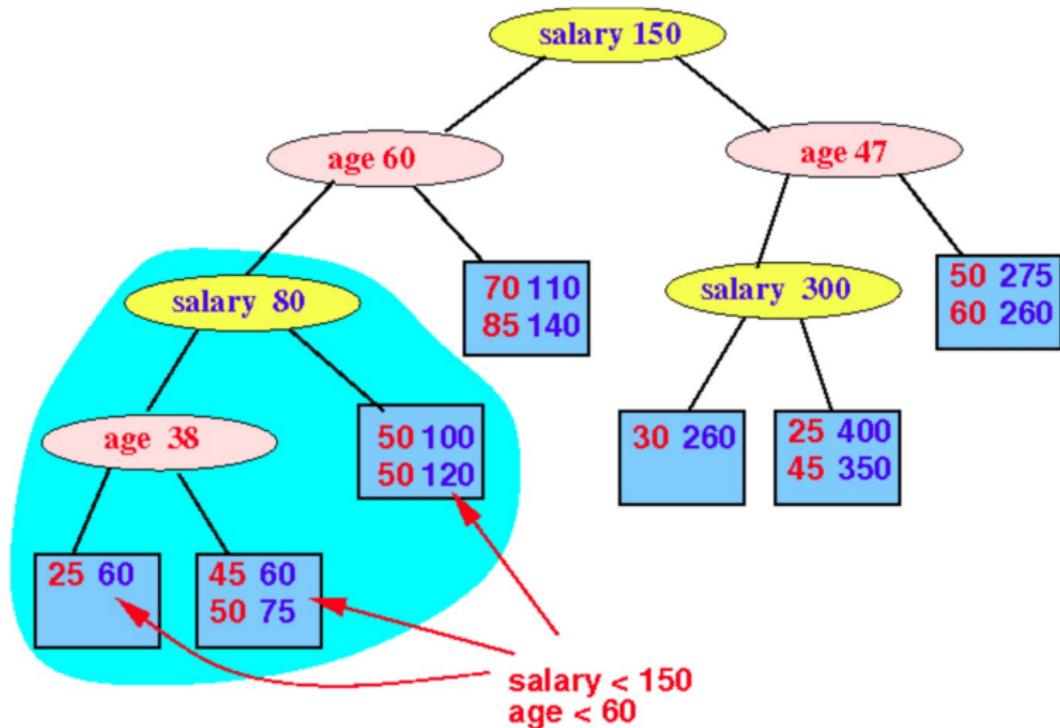
## Example kd-tree

- Behold the structural properties of the kd-tree
  - This left (shaded) subtree has salary search key values < 150 (for salary)



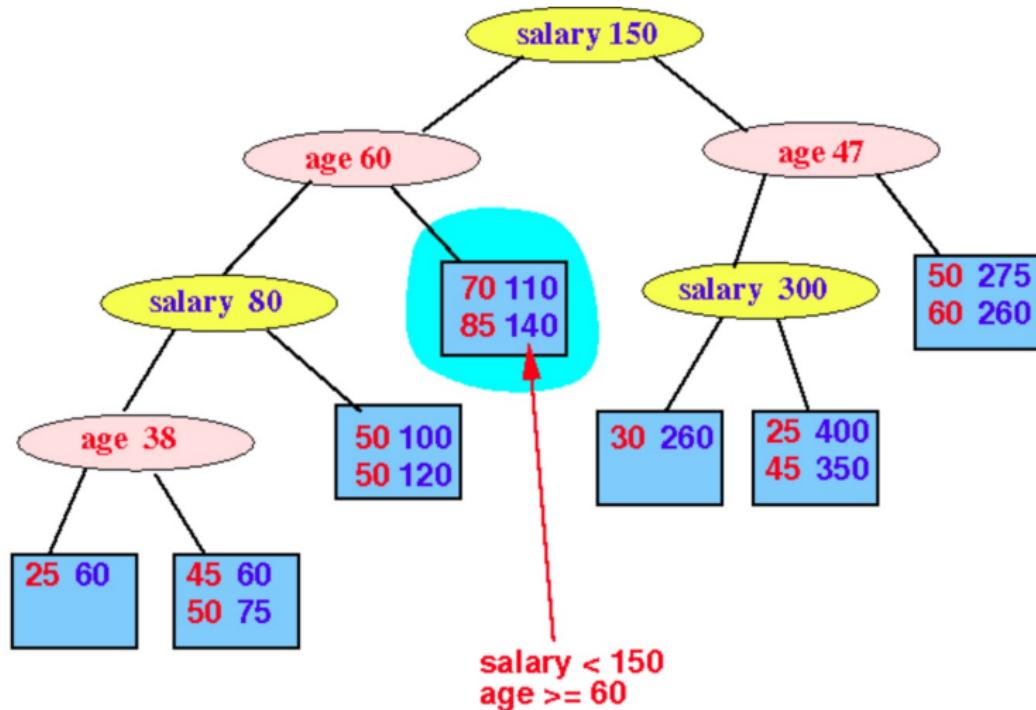
## Example kd-tree

- This left subtree has salary < 150 and age < 60



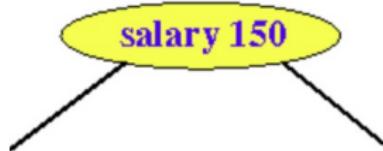
## Example kd-tree

- This right subtree has salary  $< 150$  and age  $\geq 60$

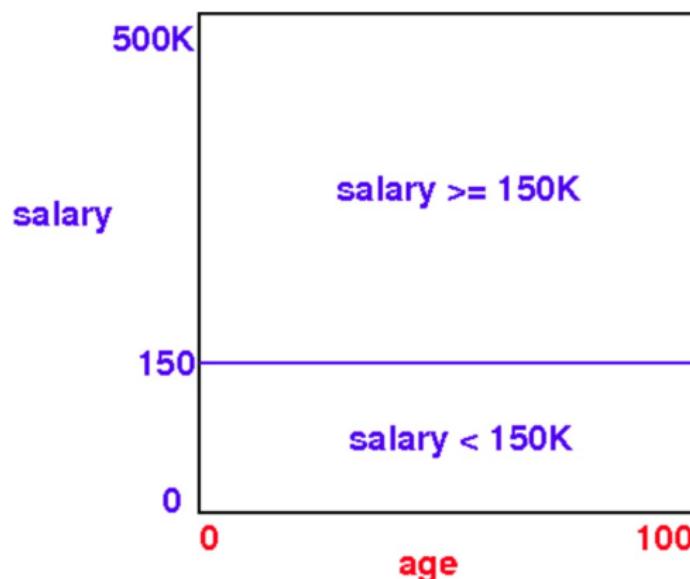


## How a kd-tree partitions the data space

- The root node

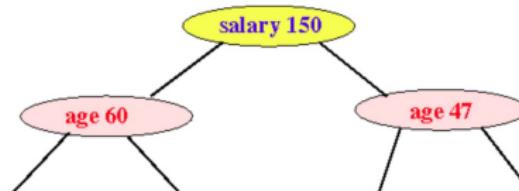


- partitions the data space in half

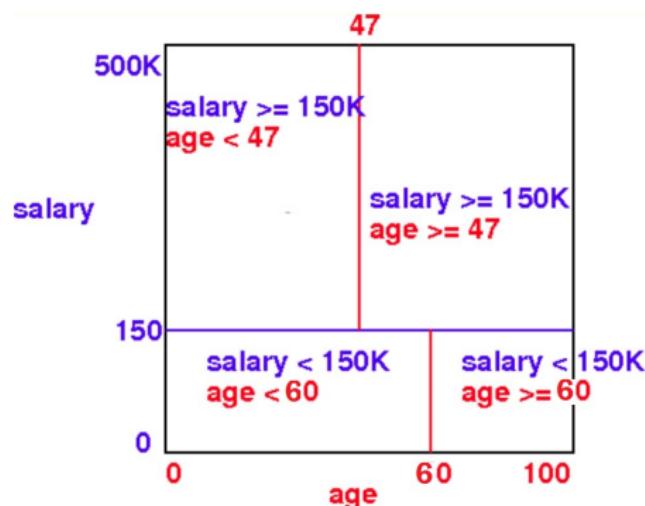


## How a kd-tree partitions the data space

- The age nodes at level 2

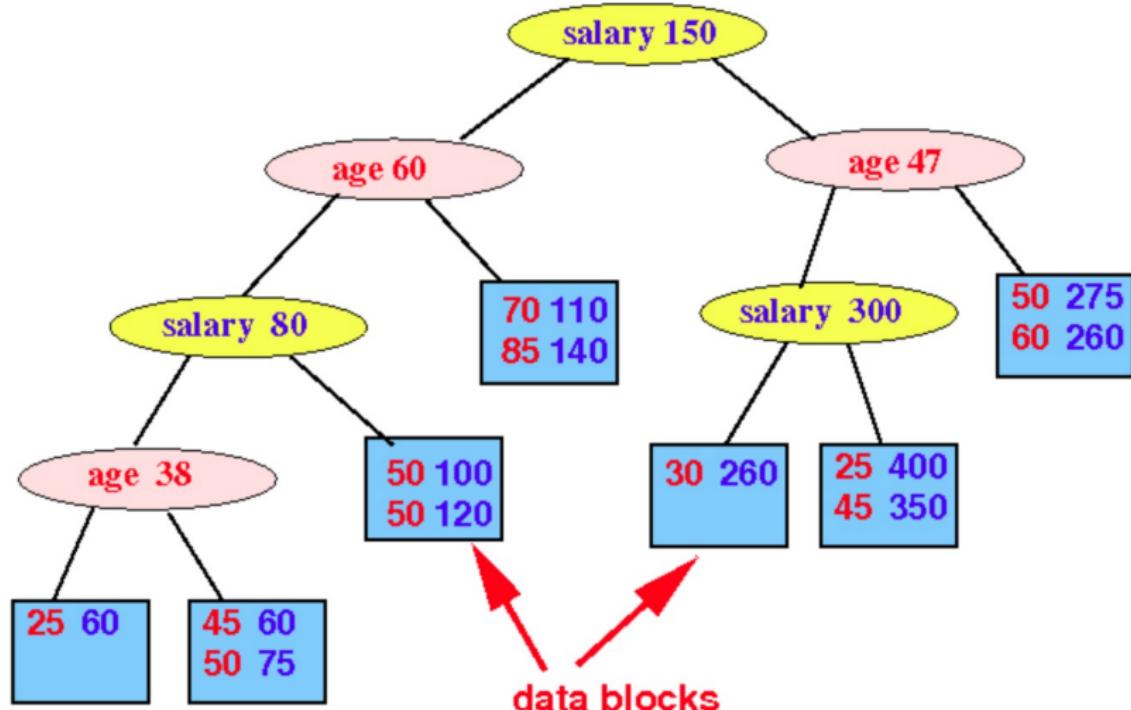


- partitions each sub-space in half



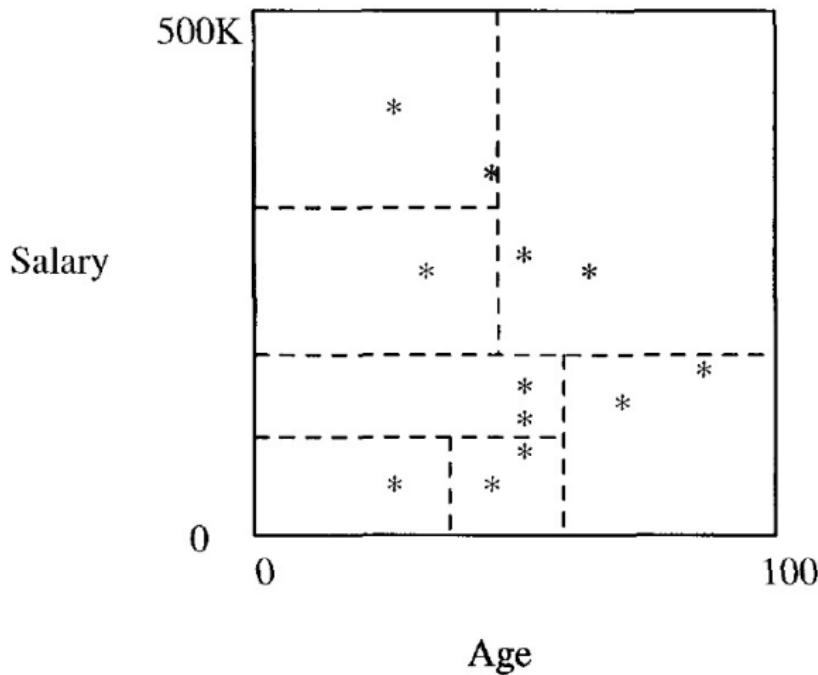
# How a kd-tree partitions the data space

- This kd-tree



## How a kd-tree partitions the data space

- will divide the data space up as follows



# Using kd-tree for common multi-dim queries

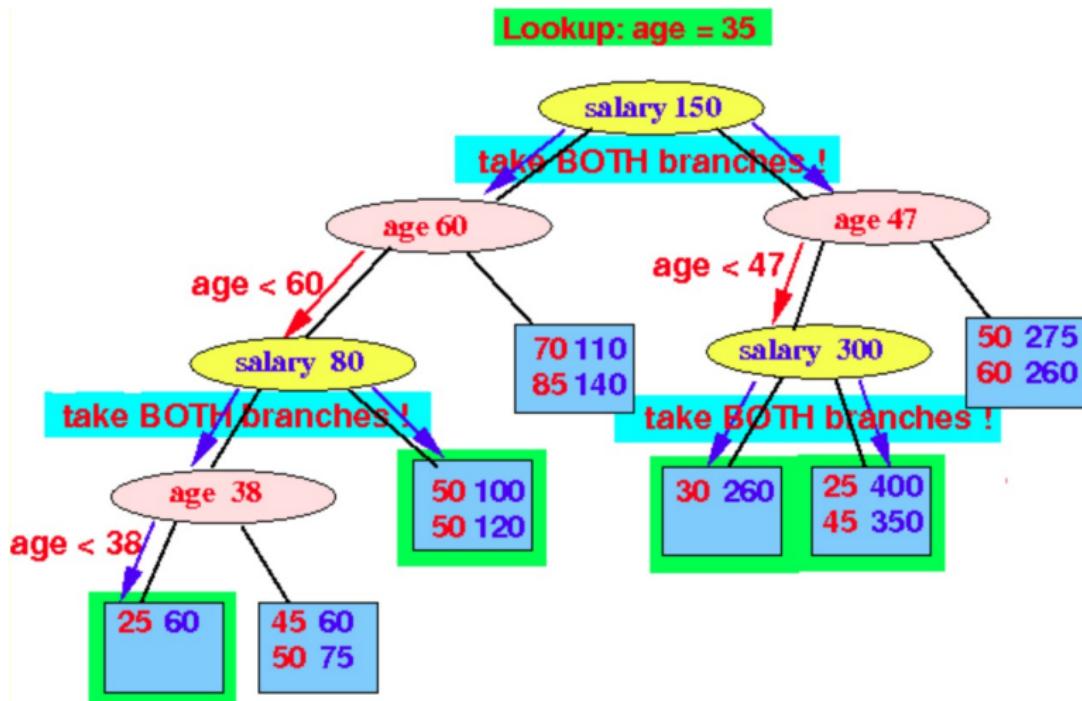
## 1) Partial Match queries

- Search Algorithm

- For a dimension for which the search value is given (specified)
  - Take the (one) branch of the subtree for the search value
- For a dimension for which the search value is not given (not specified)
  - Take both branches of the subtree

# 1) Partial Match queries: Example

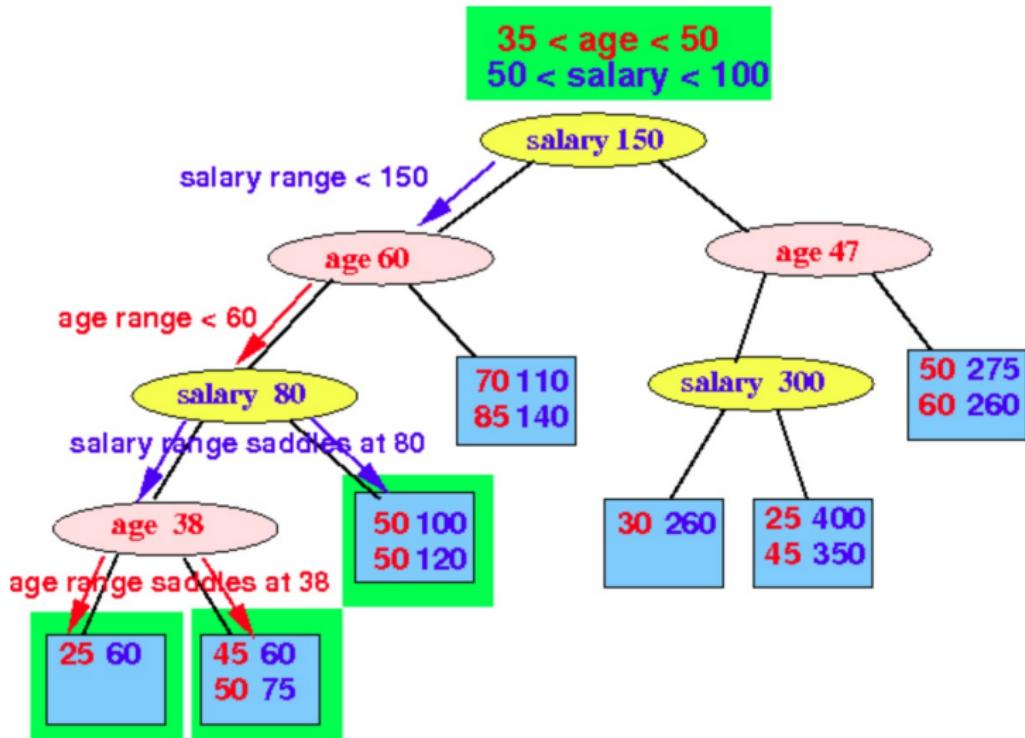
- Find all person with age = 35



## 2) Range queries

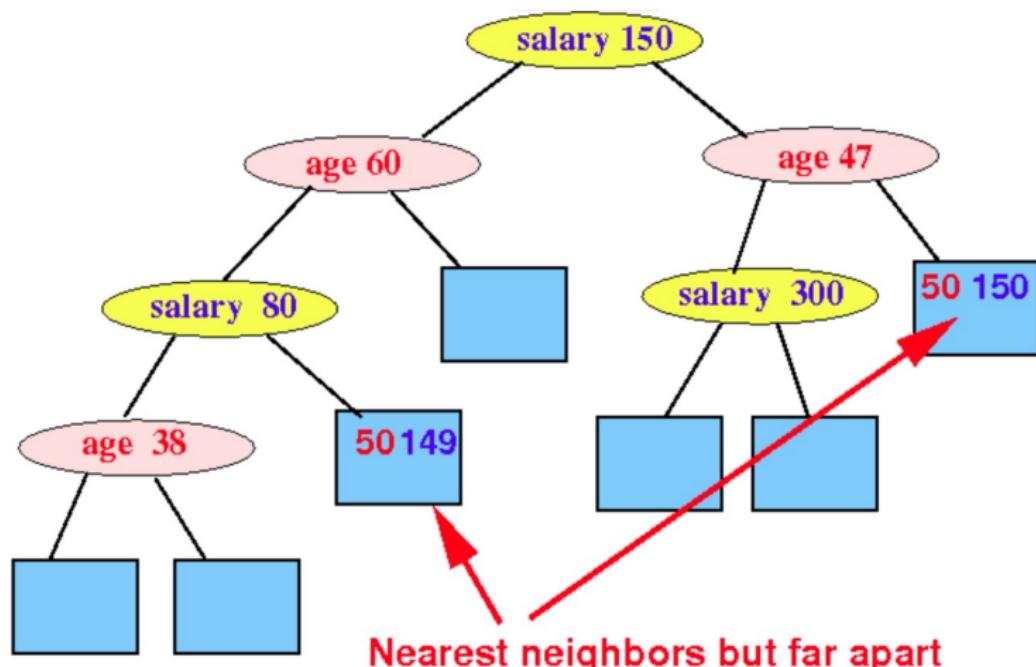
- Search Algorithm
  - For the search range is completely contained by the left subtree, then
    - Take only the left branch of the subtree for the search value
  - For the search range is completely contained by the right subtree, then
    - Take only the right branch of the subtree for the search value
  - Otherwise (the search range straddles at the search value)
    - Search both subtrees

## 2) Range queries: Example



### 3) Nearest neighbor queries

- Not easy to find the nearest neighbor using a kd-tree index



- It requires up and down traversal/search in the kd-tree

## 4) Where-am-I queries

- Not applicable
  - kd-tree can only stores points
  - Cannot store objects

- July 24<sup>th</sup>, 2018

# Multi-dimensional index structures

- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

# The Quad-tree

- An index structure that divides a search space in half (exactly) in every dimension
- Structure of a quad-tree node
  - A quad-tree node contains the following
    - 1 search key value for each dimension
    - $2^n$  child node pointers ( $n$  way split)
    - One parent node pointer (except for the root node)
  - The child node pointers will point to every possible combination of  $\leq$  and  $>$  relationships with the search key values

## Quad-tree on common multi-dimensional queries

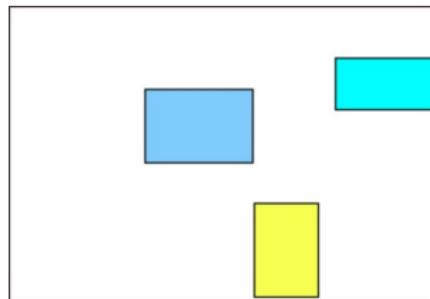
- A quad-tree is similar to a kd-tree
- The techniques discussed in the kd-tree applies to the Quad-tree

# Multi-dimensional index structures

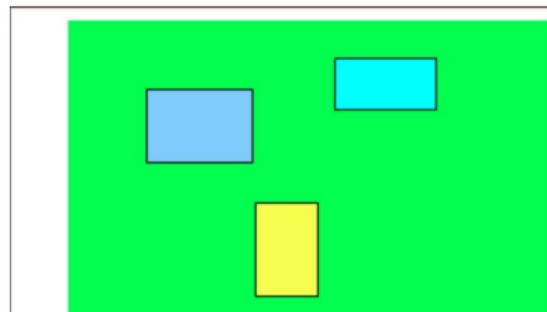
- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

# The R-tree (Region-tree)

- Bounding Box
  - a rectangle that contains a group of objects
- Example: given a group of objects

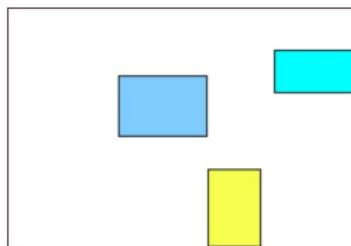


- The Bounding Box for this group of objects

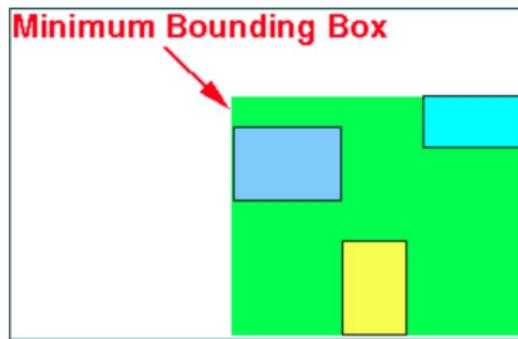


# The R-tree (Region-tree)

- Minimum Bounding Box (MBB)
  - the smallest rectangle that contains a group of objects
- Example: given a group of objects

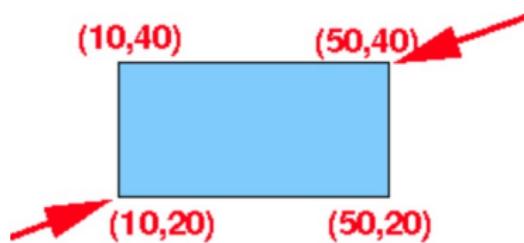


- The Minimum Bounding Box for this group of objects



## The R-tree (Region-tree)

- **Note:** A rectangle can be represented as follows
  - coordinate of the lower left corner
  - coordinate of the upper right corner
- Example: Rectangle:  $((10,20), (50,40))$

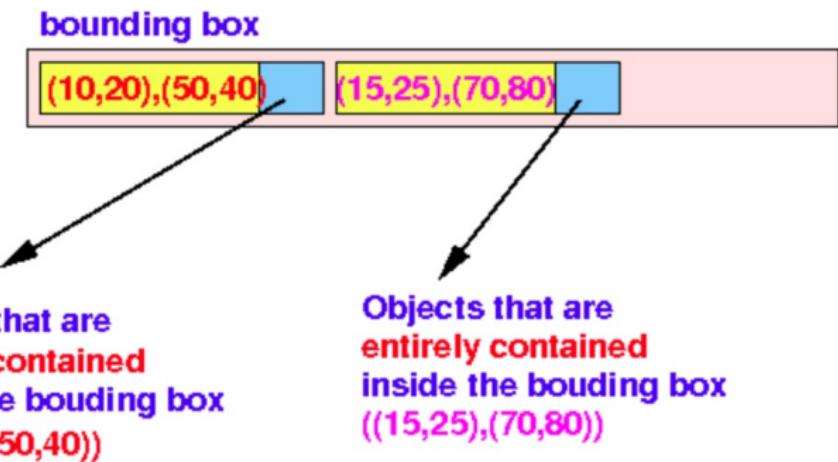


# The R-tree (Region-tree)

- **R-Tree:** an index tree-structure derived from the B-tree that uses bounding boxes as search keys
- The internal nodes contains a number of entries of the following format
  - **(bounding box, child node pointer)**
  - Example:  $((10,20), (50,40)), \text{ptr1}$
- The leaf nodes contains a number of entries of the following format:
  - **(min bounding box, object pointer)**
  - Example:  $((10,20), (50,40)), \text{house-ptr}$

# Property of a R-tree

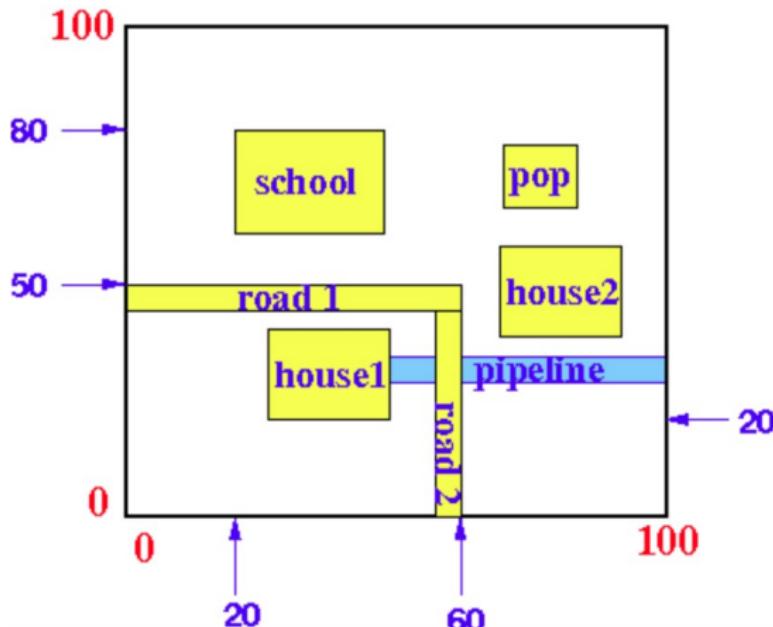
- An internal node of the R-tree has the following structure



- The subtree indexed by the bounding box will contain
  - Only objects that are contained within the given bounding box

## R-tree: Example

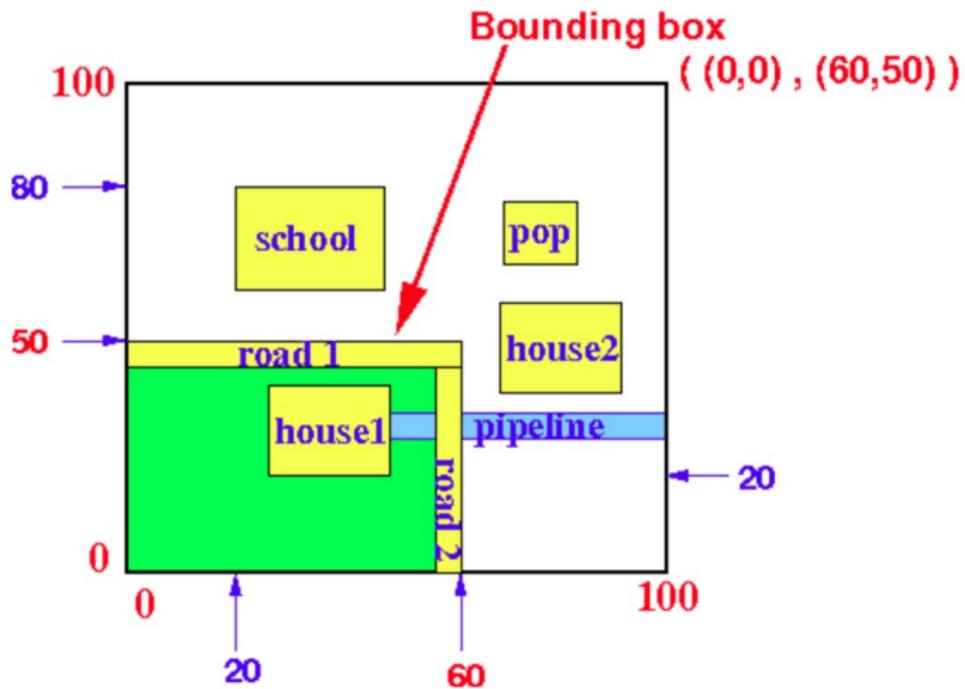
- Objects that we want to represent



- There are 7 objects
  - school, pop (point of presence), house1, house2, road1, road2, pipeline

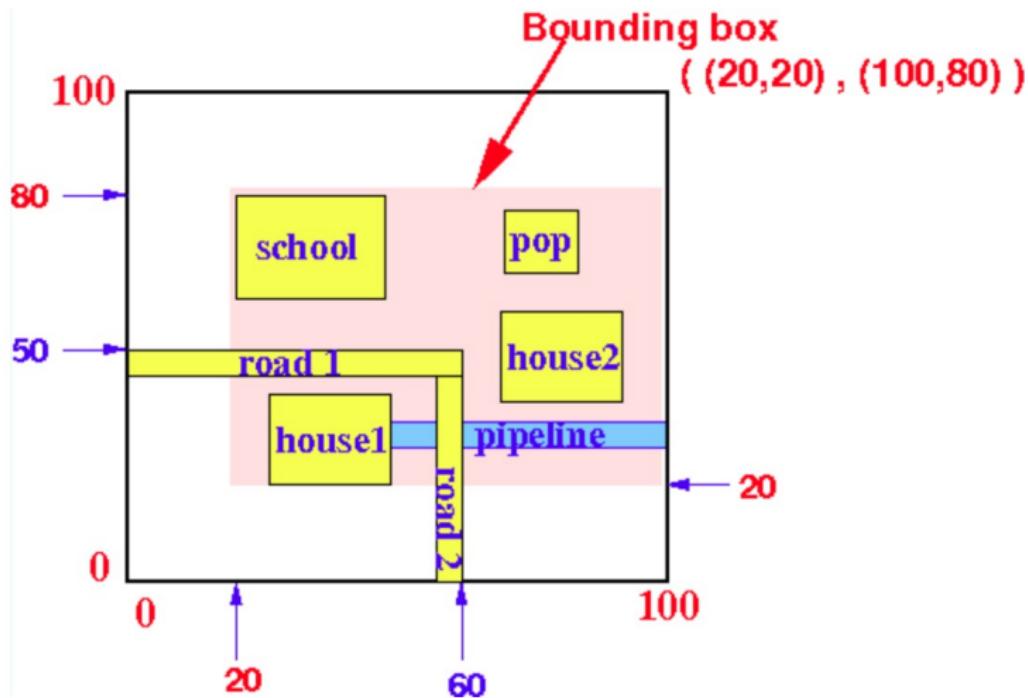
## R-tree: Example

- The 3 objects house1, road1 and road2 are completely enclosed by the bounding box  $((0,0), (60,50))$



## R-tree: Example

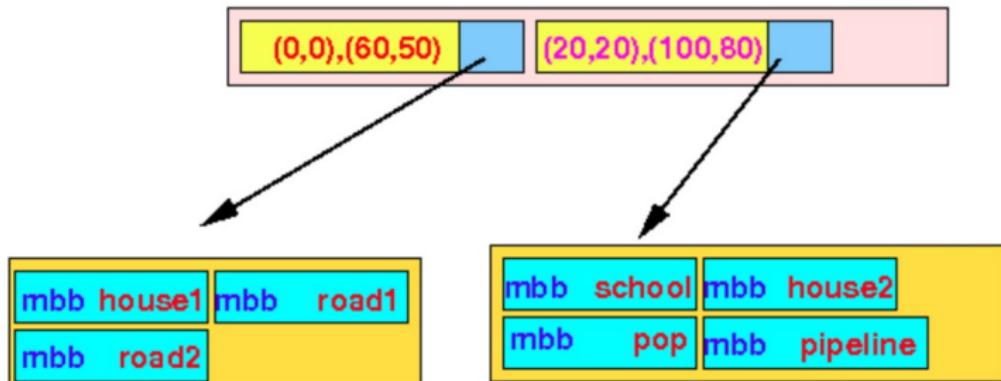
- The objects school, pop , house2 and pipeline are completely enclosed by the bounding box  $((20,20), (100,80))$



## R-tree: Example

- The R-tree that uses the previous bounding boxes

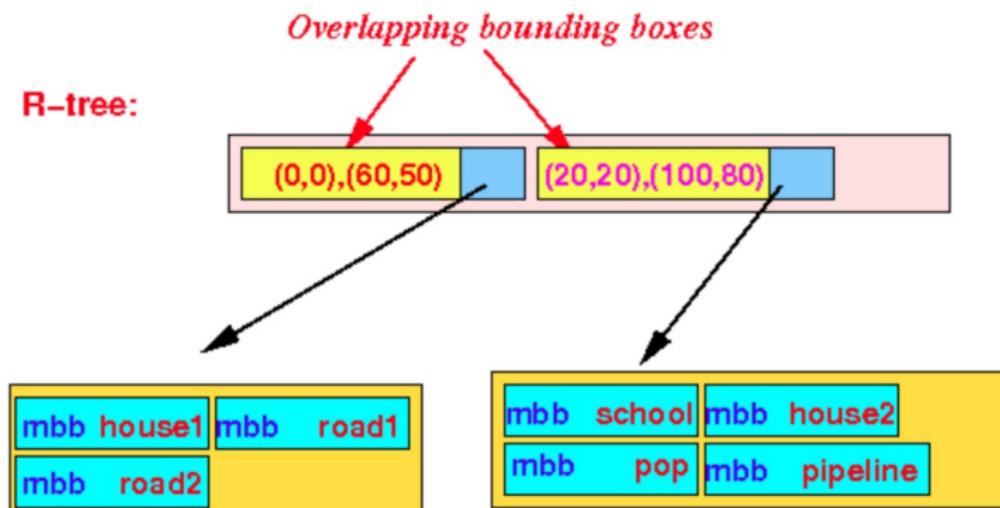
**R-tree:**



- The minimum bounding box (mbb) field for different objects are different

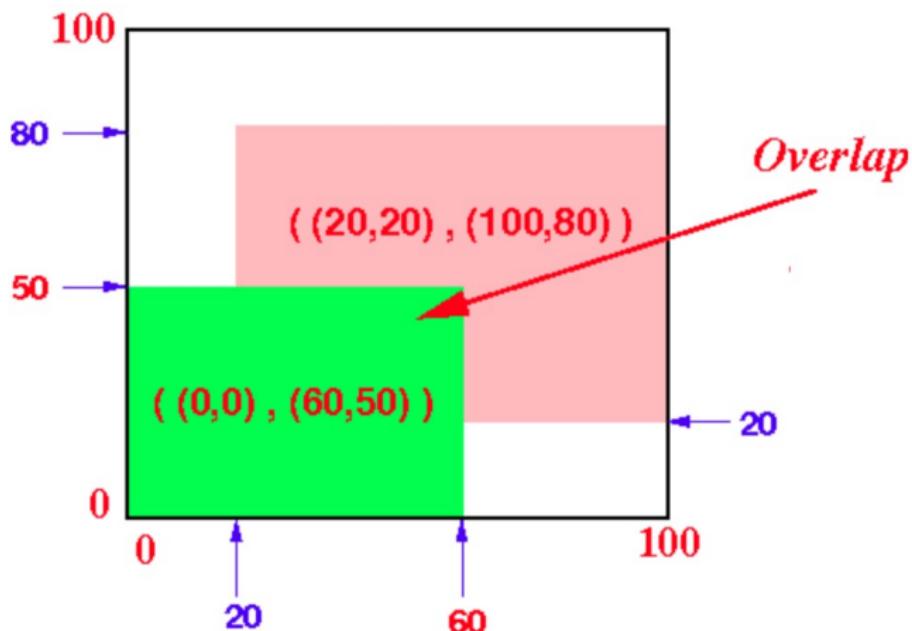
# Overlapping Bounding boxes in R-tree

- The bounding boxes used in the internal R-tree nodes can overlap
- Example



## Overlapping Bounding boxes in R-tree

- You can see the overlap clearly



## Lookup operation in the R-tree

- Lookup algorithm for a point in an R-tree
  - Search Algorithm for a Point( $x, y$ )
    - The search algorithm is recursive
    - The search starts at the root node of the R-tree

# Search algorithm for a point P(x,y)

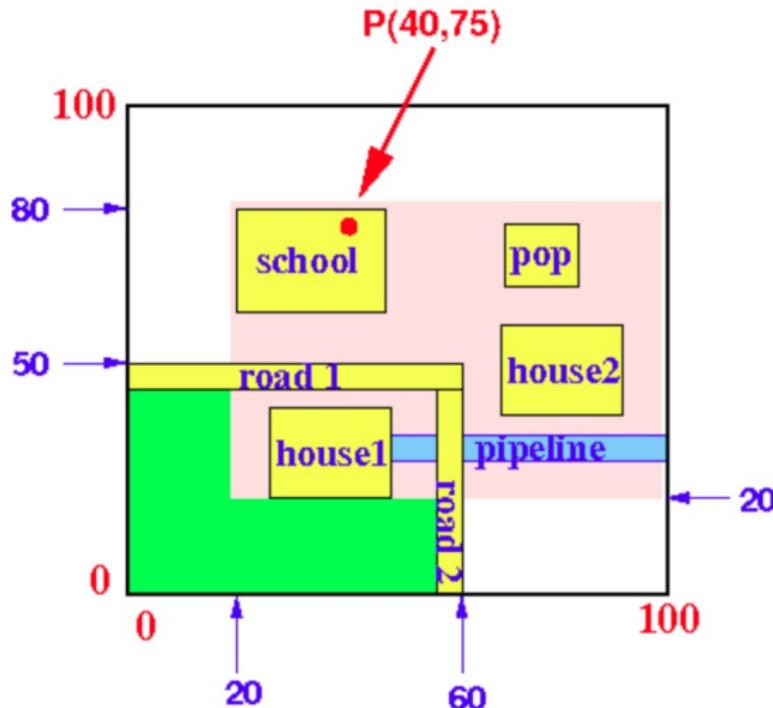
---

## Algorithm 2 Lookup((x, y), n, result)

```
1: // n = current node of the search in the R-tree
2: if ( n == internal node ) then
3:   for each entry (BB, childptr) in internal node n do
4:     // Look in subtree if (x,y) is inside bounding box
5:     if (x,y) ∈ BB then
6:       Lookup((x,y), childptr, result)
7:     end if
8:   end for
9: else
10:   //n is a leaf node
11:   for ( each object Ob in node n) do
12:     if (x,y) ∈ MBB(Ob) then
13:       Add Ob to result // Object Ob contains point (x,y)
14:     end if
15:   end for
16: end if
```

## Lookup operation in the R-tree: Example

- Find the object(s) that contain the point  $P(40, 75)$

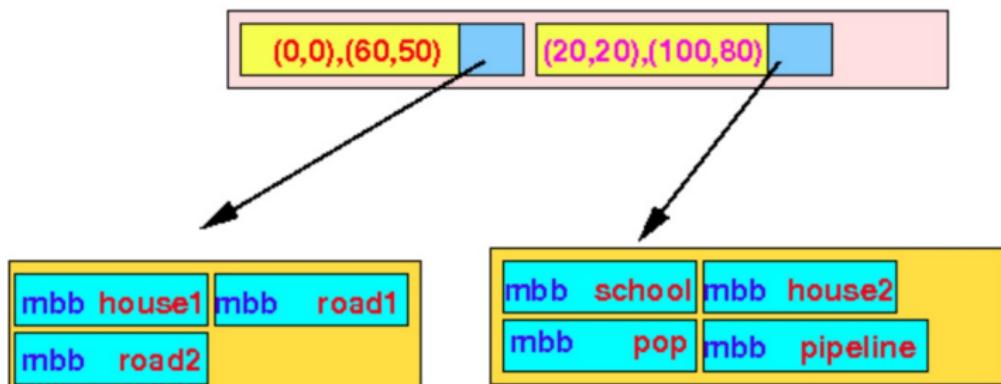


# Lookup operation in the R-tree: Example

- Execution of the search algorithm:

- Input:

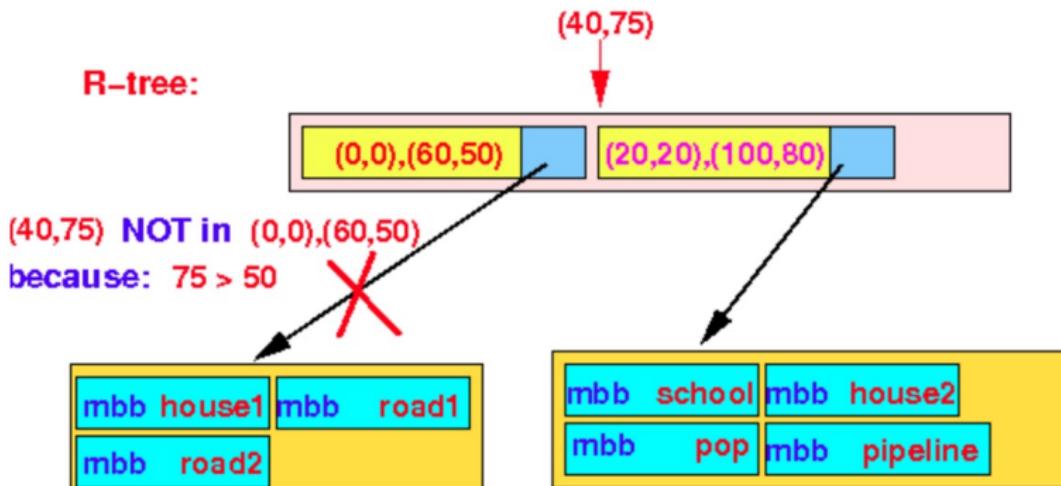
R-tree:



- Point  $(40,70)$

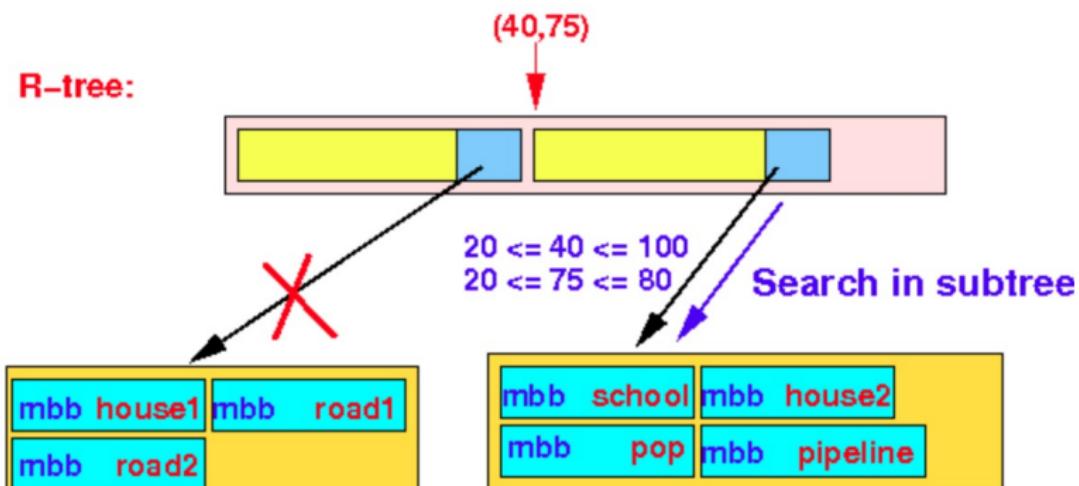
# Lookup operation in the R-tree: Example

- Execution of the search algorithm:
  - Check in first Bounding Box
    - point P(40,75)  $\notin$  bounding box ((0,0),(60,50))
  - Skip the first subtree



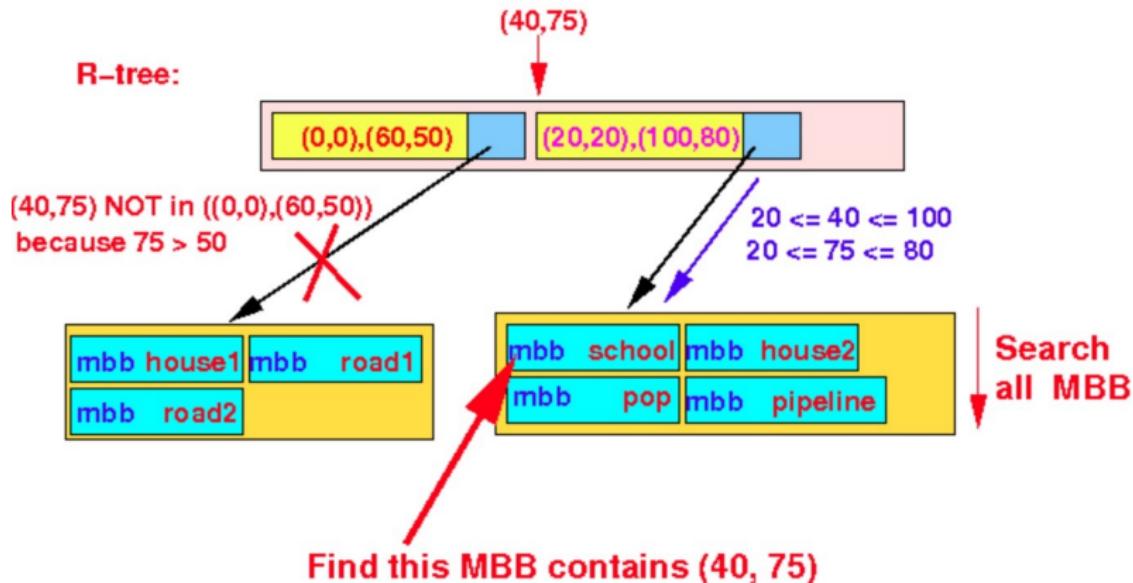
# Lookup operation in the R-tree: Example

- Execution of the search algorithm:
  - Check in second Bounding Box
    - point  $P(40,75) \in$  bounding box  $((20,20), (100,80))$
  - The search algorithm will recurse and search the  $2^{nd}$  subtree



## Lookup operation in the R-tree: Example

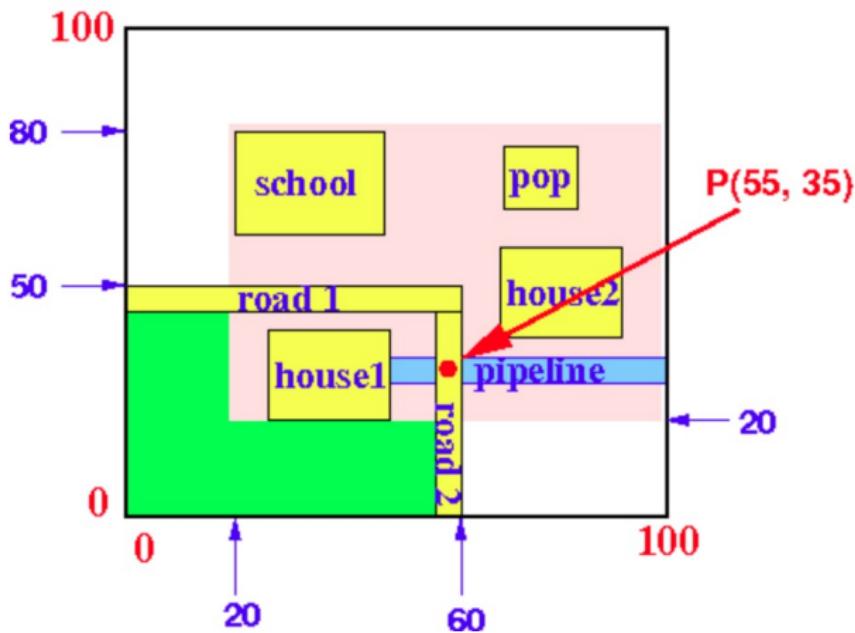
- Execution of the search algorithm:
  - We arrived at a leaf node, so we search the MBBs in the leaf node



- We find that the school object contains the point

## Lookup operation in the R-tree: Example 2

- Find the object(s) that contain the point  $P(55, 35)$

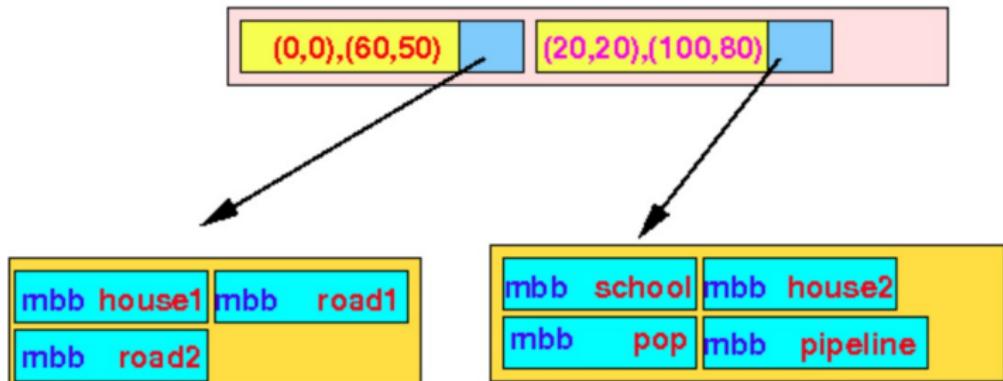


# Lookup operation in the R-tree: Example

- Execution of the search algorithm:

- Input:

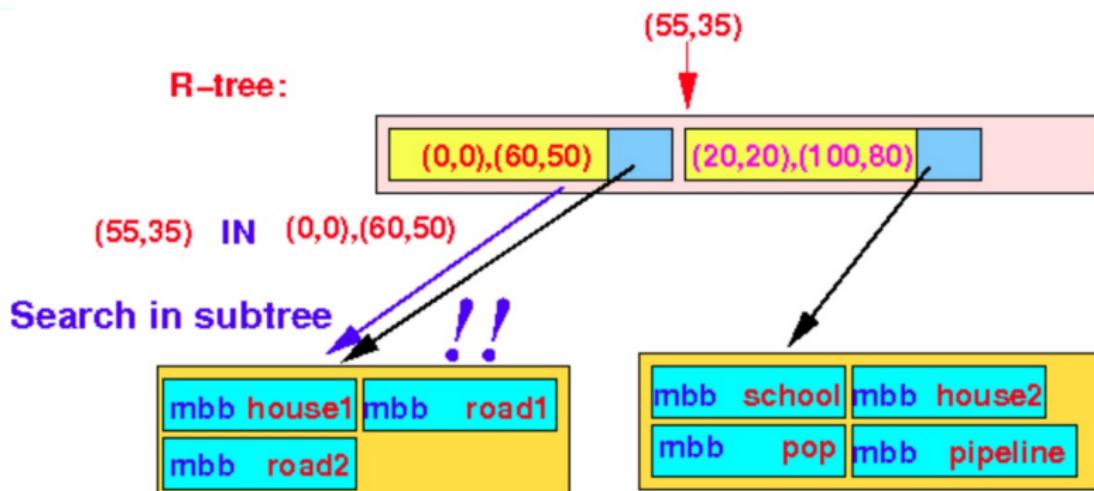
R-tree:



- Point (55,35)

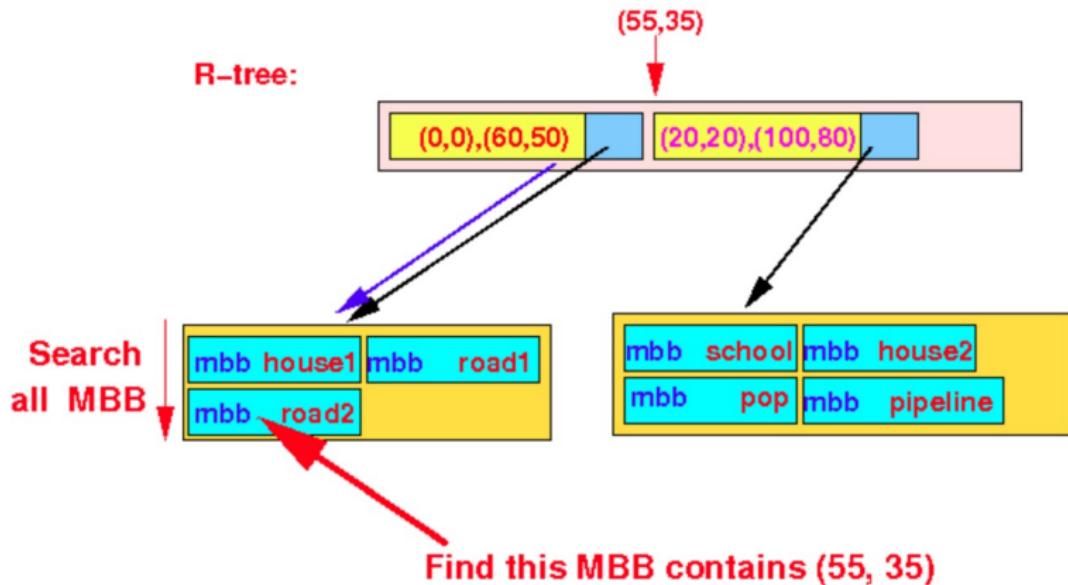
## Lookup operation in the R-tree: Example 2

- Execution of the search algorithm:
  - Check in first Bounding Box
    - point  $P(55,35) \in$  bounding box  $((0,0), (60,50))$
  - The search algorithm will recurse and search the 1<sup>st</sup> subtree



## Lookup operation in the R-tree: Example 2

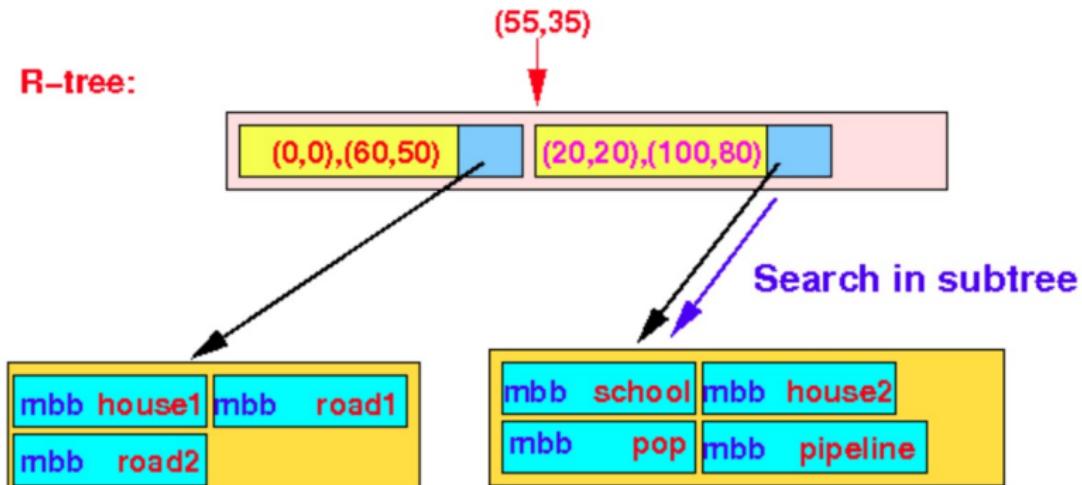
- Execution of the search algorithm:
  - We arrived at a leaf node, so we search the MBBs in the leaf node



- We find that the road2 object contains the point

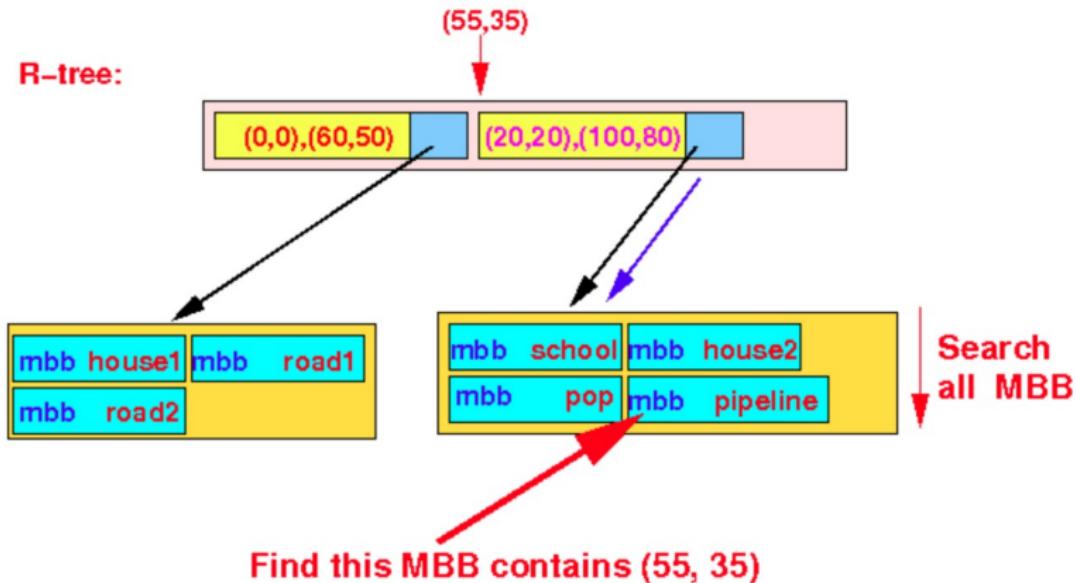
## Lookup operation in the R-tree: Example 2

- Execution of the search algorithm:
  - Check in second Bounding Box
    - point P(55,35)  $\in$  bounding box ((20,20), (100,80))
  - The search algorithm will also recurse and search the 2<sup>nd</sup> subtree



## Lookup operation in the R-tree: Example 2

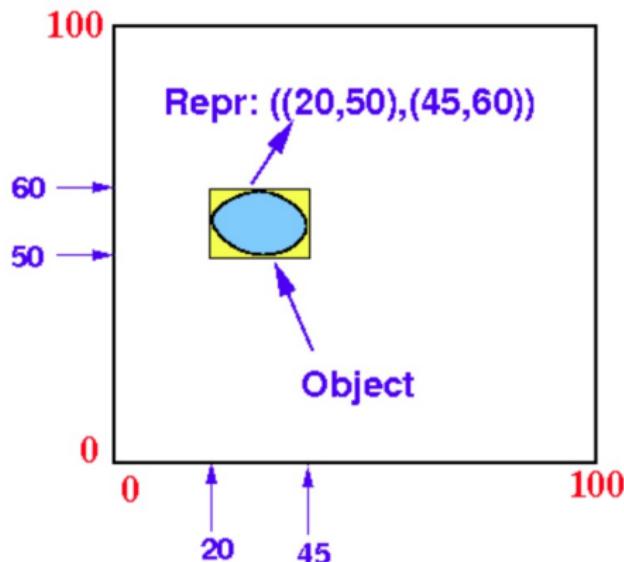
- Execution of the search algorithm:
  - We arrived at a leaf node, so we search the MBBs in the leaf node



- We find that the pipeline object contains the point

## Lookup algorithm for an object (rectangle) in the R-tree

- Simplification: We represent an object by its Minimum Bounding Box
- Example



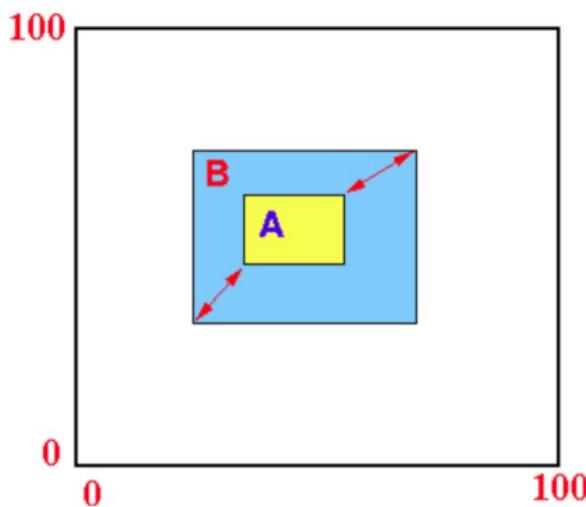
# Lookup algorithm for an object (rectangle) in the R-tree

- Containment relationship of bounding boxes
  - A bounding box A is contained in a bounding box B iff

$$x_{LL}(B) \leq x_{LL}(A) \text{ and } y_{LL}(B) \leq y_{LL}(A)$$

$$x_{UR}(A) \leq x_{UR}(B) \text{ and } y_{UR}(A) \leq y_{UR}(B)$$

- Graphically:



## Search algorithm for objects in an R-tree

- Is similar to the search algorithm for points using the containment relationship for bounding boxes
- Algorithm to find rectangle object is called using
  - `Lookup(Obj, n, result)`: Look for object that contain rectangle Obj
    - Obj: representation of a rectangle object
    - n: current node of search
    - result: output

# Search algorithm for objects in an R-tree

## Algorithm 3 Lookup( Obj, n, result )

```
1: // n = current node of the search in the R-tree
2: if ( n == internal node ) then
3:   for each entry (BB, childptr) in internal node n do
4:     // Look in subtree if Obj is inside bounding box
5:     if Obj ⊆ BB then
6:       Lookup( Obj, childptr, result )
7:     end if
8:   end for
9: else
10:   //n is a leaf node
11:   for ( each object Ob in node n) do
12:     if MBB(Obj) == MBB(Ob) then
13:       Add Ob to result // Object Ob contains Obj
14:     return // Obj found, done
15:   end if
16: end for
```

## Insert into the R-tree

- Insert algorithm (to insert an object into a leaf node of the R-tree)
- The insert algorithm is called using
  - $\text{Insert}((\text{MBB}, \text{ObjID}), \text{root})$ 
    - MBB: Min. Bounding Box of object
    - ObjID: Object ID (e.g.: record pointer)
    - root: root node of the R-tree

# Insert Algorithm

---

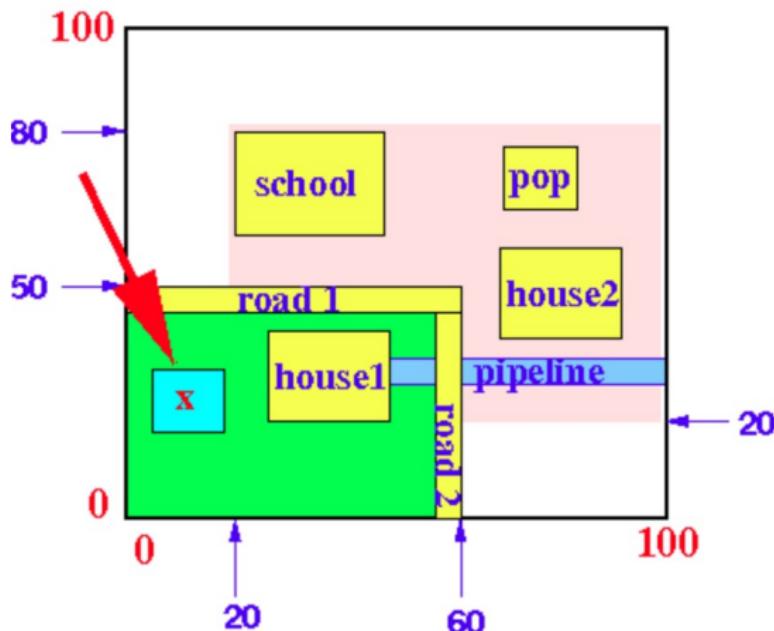
## Algorithm 4 Insert( (mbb,ObjID), n)

```
1: if ( n == internal node ) then
2:   for each entry (BB, childptr) in node n ) do
3:     // Insert the object in subtree of the first BB that contains the mbb of the inserted object
4:     if mbb of inserted object  $\subseteq$  BB then
5:       //Insert object in this subtree
6:       Insert((mbb,ObjID),childptr) // The recursion WILL have inserted the object
7:       return // Terminates the for-loop
8:     end if
9:   end for
10:  // We ONLY reach here when EVERY BB in n not contain object. We enlarge one of the BB so it can store object
11:  Find a BB in n such that enlarging BB to contain the new object will add the least amount of area to BB
12:  Update this BB in n to the Enlarged BB
13:  // Now the enlarged BB contains the object
14:  Insert((mbb,ObjID), childptr) // Insert in subtree
15:  return // Done
16: else if leaf node has space to hold object then
17:   // We found the leaf node to hold the object
18:   insert(mbb, ObjID) in the leaf node
19: else
20:   Split the objects in the leaf node into set1 and set2
21:   Find the bounding box BB1 for set1 of the objects and bounding box BB2 for set2 of the objects
22:   Replace the parent's (BB, ptr) by (BB1, set1) and (BB2, set2)
23:   // This step can cause the parent node to overflow. We will need to split the parent node into 2 internal nodes.
24:   // You will use Insert procedure into an internal node
25: end if
```

---

## Insert example 1: easy case (leaf node has space for object)

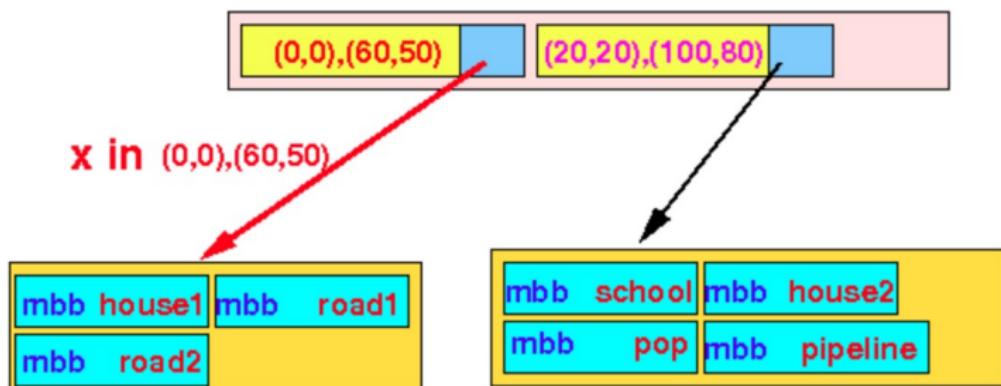
- Insert the object  $x$  into the R-tree



# Insert example 1: Steps of the Insert Algorithm

- The first BB contains x

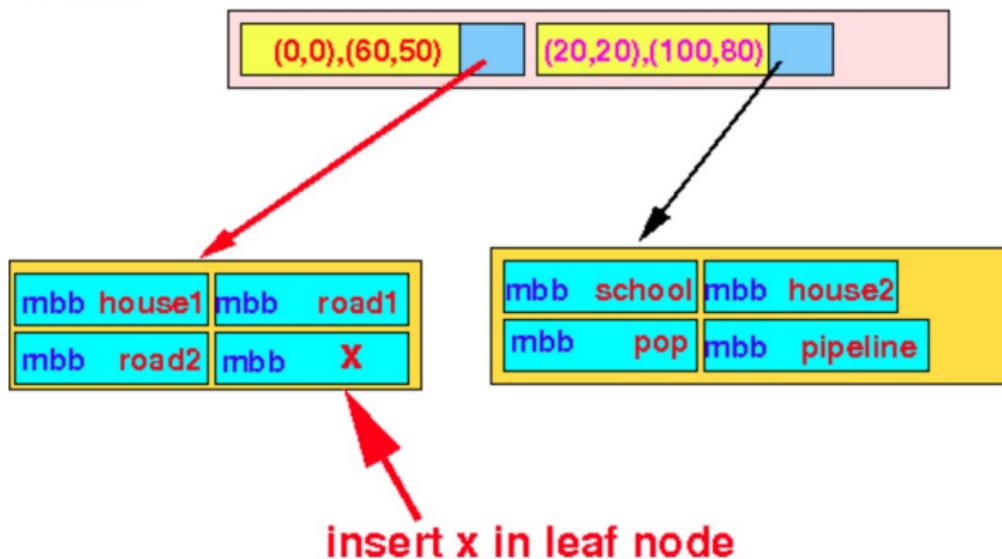
R-tree:



## Insert example 1: Steps of the Insert Algorithm

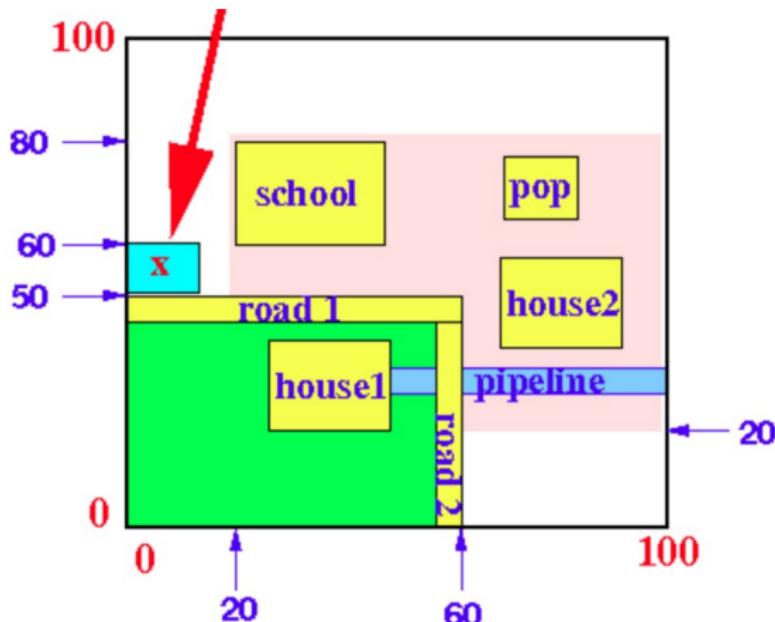
- Because there is space in the leaf node, we insert the object x there

R-tree:



## Insert example 2: (augmenting the bounding box)

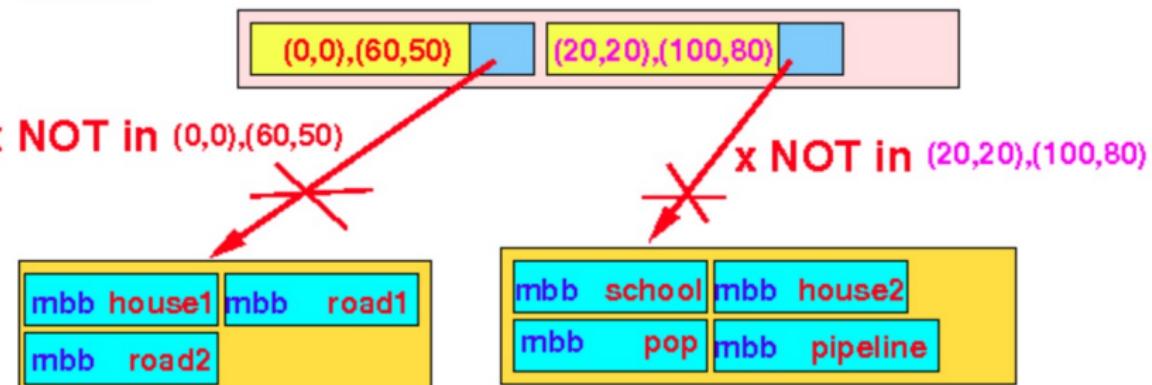
- Insert the object  $x$  into the R-tree



## Insert example 2: Steps of the Insert Algorithm

- Find a bounding box that contains  $x$

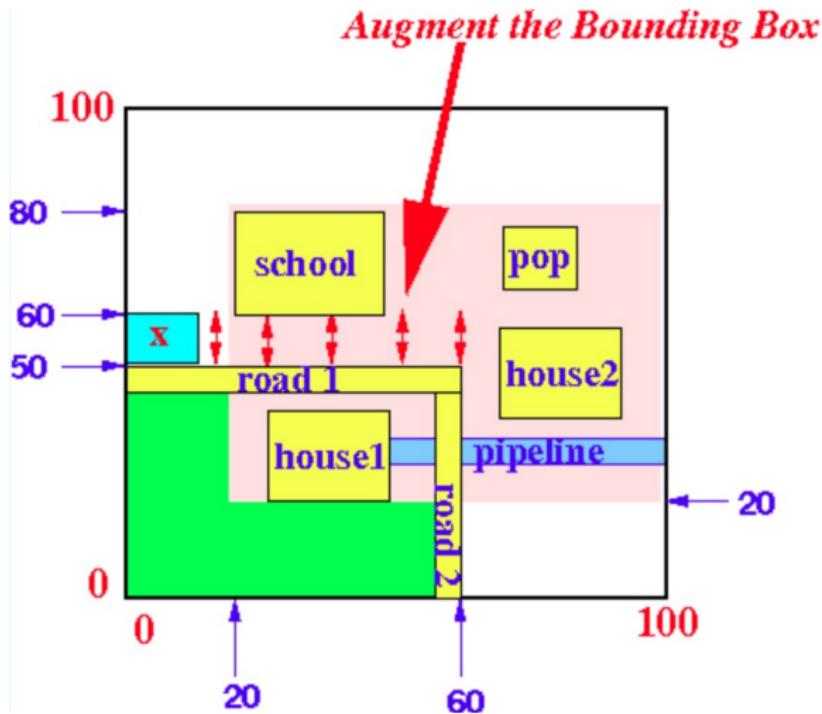
R-tree:



- We cannot find any BB that contains the object  $x$ , therefore Augment one of the bounding boxes

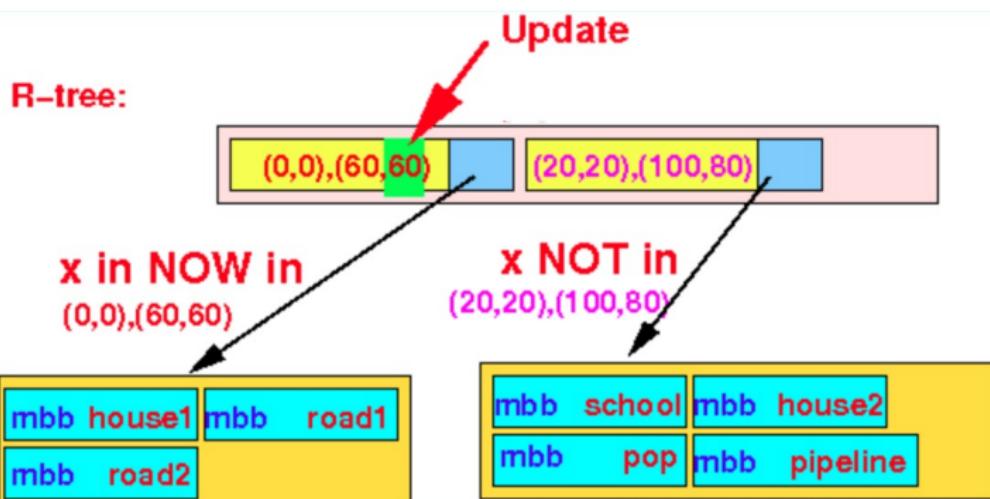
## Insert example 2: Steps of the Insert Algorithm

- We can augment the first BB to  $((0,0), (60,60))$



## Insert example 2: Steps of the Insert Algorithm

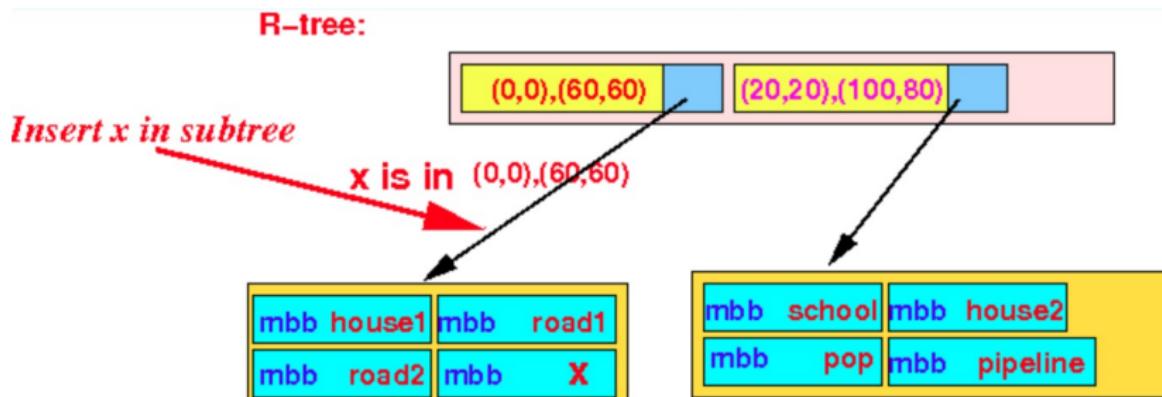
- so that the new will include the object x



There is no need to update the BB in the parent node

## Insert example 2: Steps of the Insert Algorithm

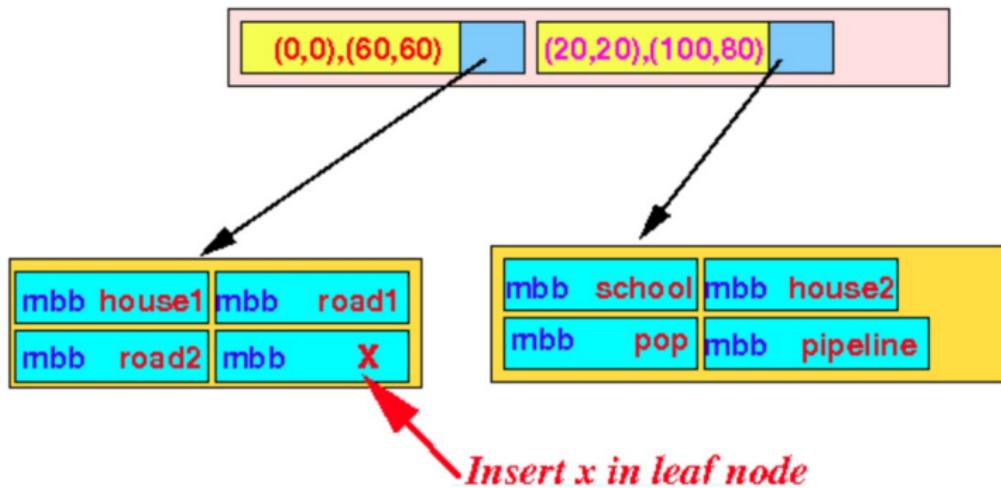
- After augmenting, we can insert the object  $x$  in the subtree



## Insert example 2: Steps of the Insert Algorithm

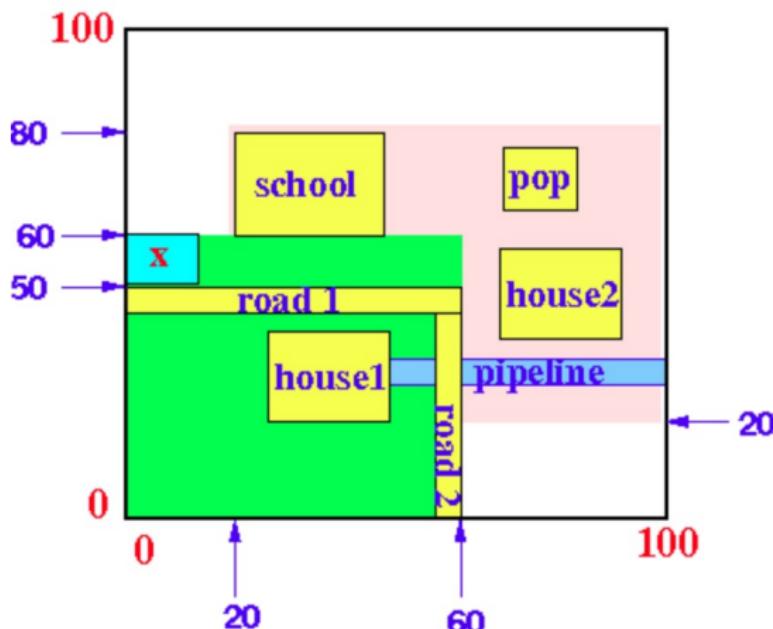
- Which will insert x in this leaf node

R-tree:



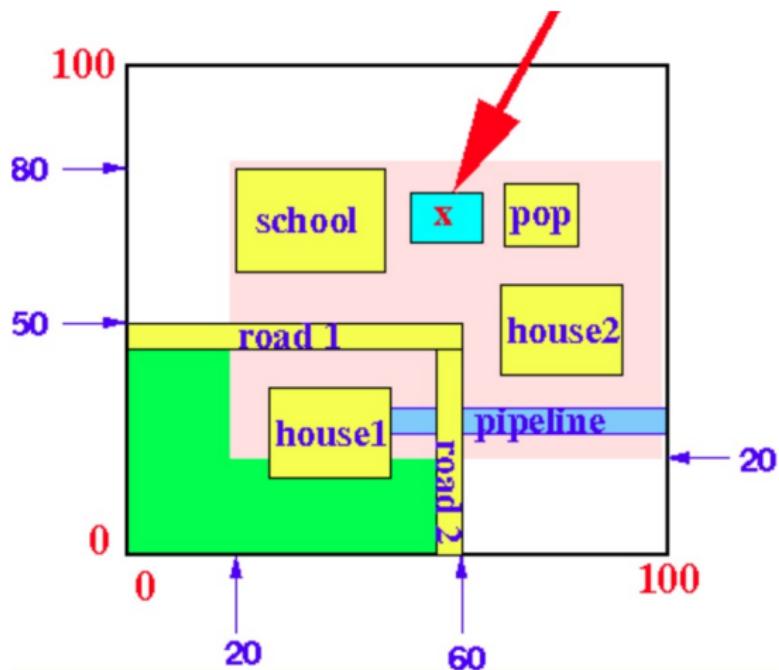
## Insert example 2: Steps of the Insert Algorithm

- Which represents the following result



## Insert example 3: (split a leaf node)

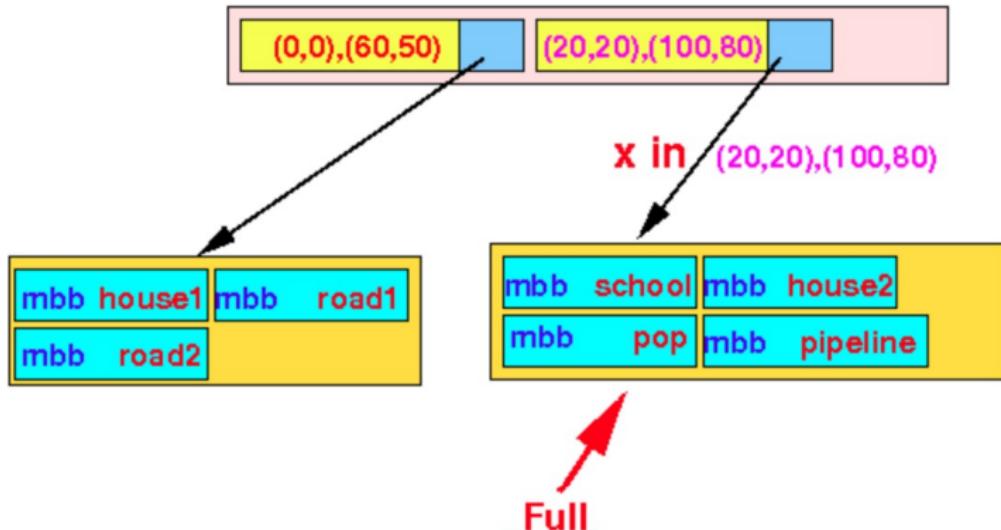
- Insert the object  $x$  into the R-tree



## Insert example 3: Steps of the Insert Algorithm

- We find a BB that contains the object  $x$

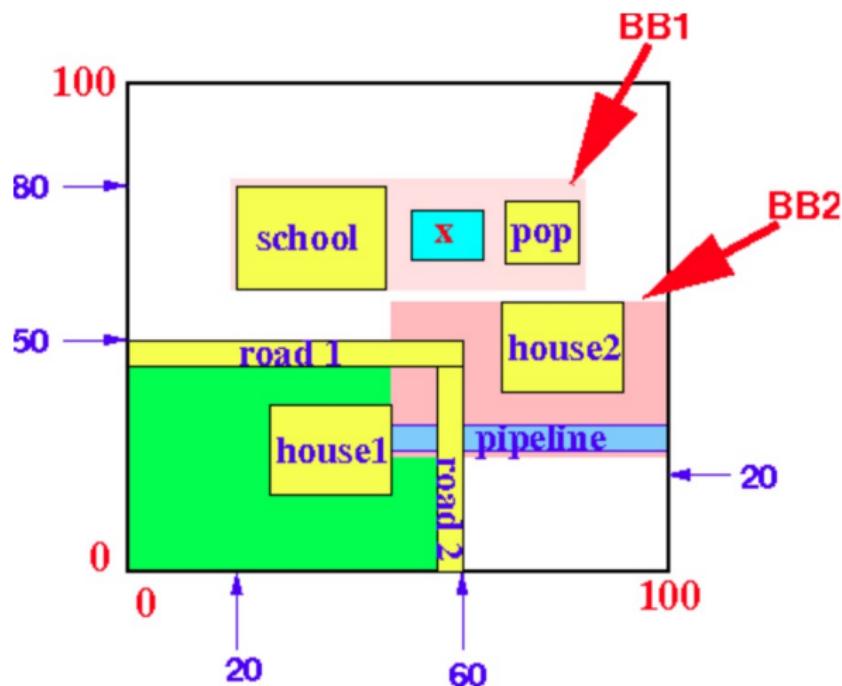
R-tree:



- However, the leaf node is full. We partition the objects into 2 groups

## Insert example 3: Steps of the Insert Algorithm

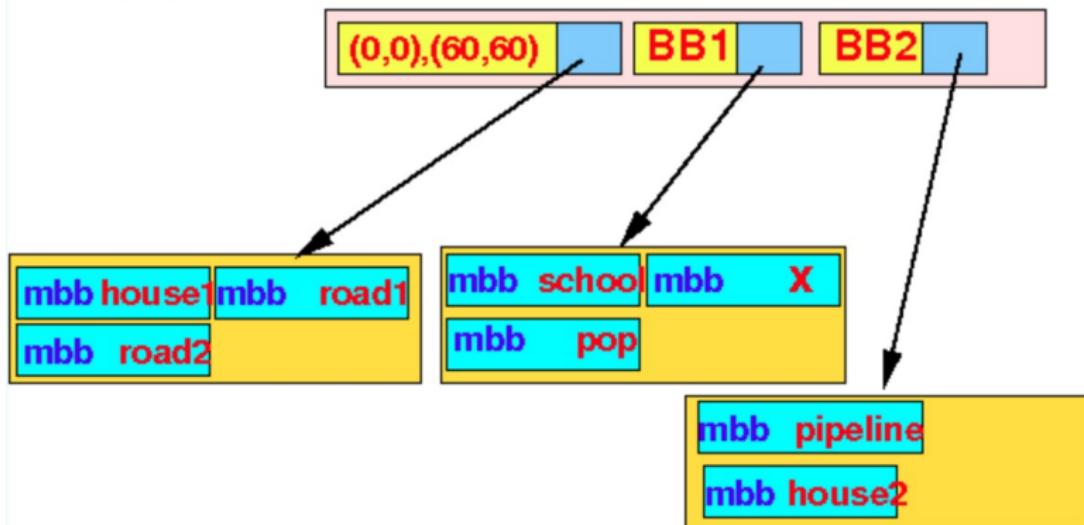
- Suppose we partition the objects as follows



## Insert example 3: Steps of the Insert Algorithm

- The new R-tree is then

R-tree:

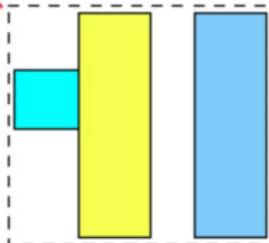


- Notice that: The bounding box  $((20,20), 100, 80)$  in the parent node is replaced by 2 bounding boxes BB1 and BB2

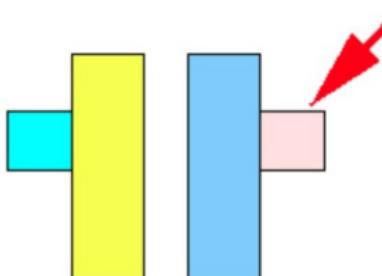
## How to partition objects in a internal/leaf node

- How to partition (BB or MBB) objects in an overflow node
  - Suppose a node contains the following (BB or MBB) objects

Bounding Box

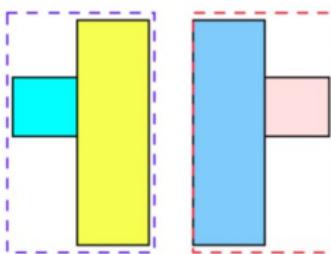


- We insert the following object

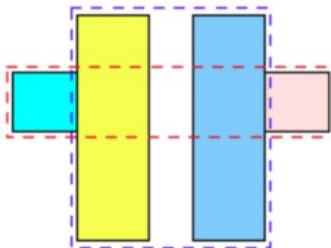


## A note on how to partition objects in a internal/leaf node

- The node is now overflow and we want to split the objects into 2 groups.
- How should the objects be split
  - An obvious choice is:



- A not so obvious choice is:



- This is a superior split because the total area of the bounding boxes is smaller

# A note on how to partition objects in a internal/leaf node

- The goal of the partitioning
  - Fact: The smaller Bounding Box, the more precisely the Bounding Box describes a search area
  - Therefore: When we split the (search) Bounding Boxes in a node, we want to minimize the total area of the bounding boxes of the split

# A naive object partitioning algorithm

- Problem description
  - Given a set of  $n$  objects
  - Find the partitioning of the  $n$  objects into 2 groups **A** and **B** such that:
    - $\text{AreaOfMBB(objects in A)} + \text{AreaOfMBB(objects in B)} = \text{minimal}$
- Naive Algorithm
  - Brute force: test every possible split
  - Complexity of the naive re-distribute algorithm =  $O(2^{n-1})$
- There exists a  $O(n^2)$  algorithm to find the reasonably good partitioning and is described in R-trees: a dynamic index structure for spatial searching)

# Topics

- Multi-dimensional information and query
- Motivation for Multi-dimensional indexes
- Multi-dimensional index structures
  - Hash like structures
  - Tree like structures
- Bitmap indices

## Bitmap indexes

- Assumption: Records in a file/relation occupy a permanent location in the file/relation
  - A records is uniquely identified by a position ID
- Definition: Current value set ( $F$ ): the current set of values stored in a field  $f$  in the records
- Example

Records:			
	field $f$		
Record 1:	.....	5	...
Record 2:	.....	7	...
Record 3:	.....	8	...

$F = \{5, 7, 8\}$

## Bitmap indexes

- Bitmap index of a field  $f$ : is a collection of bit vectors of length  $n$ , where  $n$  is the number of records
- There is one bit vector for each value  $v$  that appears in field  $f$
- The bit vector for the value  $v$  is equal to
  - $x_1 \ x_2 \ \dots \ x_i \ \dots \ x_n$
  - $x_i = 1$  if the  $i^{th}$  record's field  $f = v$ , otherwise = 0

## Bitmap indexes: Example

- A file has 6 records

Fields:	A	B
record 1:	30	foo
record 2:	30	bar
record 3:	40	baz
record 4:	50	foo
record 5:	40	bar
record 6:	30	baz

- The bitmap index for the field A is

value	123456	
30	110001	<---- bit vector
40	001010	
50	000100	

Explanation:

The value 30 appears in the records: 1, 2, 6  
So: bit #1, #2 and #6 are set

## Bitmap indexes: Example

- A file has 6 records

Fields:	A	B
record 1:	30	foo
record 2:	30	bar
record 3:	40	baz
record 4:	50	foo
record 5:	40	bar
record 6:	30	baz

- The bitmap index for the field B is

value	123456	
foo	100100	<---- bit vector
bar	010010	
baz	001001	

Explanation:

The value **foo** appears in the records: 1, 4  
So: bit #1 and #4 are set

## Bitmap indexes: Example: people who buy jewelry

- Data on people who buy jewelry

(age, salary (in \$1,000))

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)

- The bitmap index on age is

Value	123456789012
25	100000001000
30	000000010000
45	010000000100
50	001110000010
60	000000000001
70	000001000000
85	000000100000

## Bitmap indexes: Example: people who buy jewelry

- Data on people who buy jewelry

(age, salary (in \$1,000))

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)

- The bitmap index on salary is

Value	123456789012
60	110000000000
75	001000000000
100	000100000000
110	000001000000
120	000001000000
140	000000100000
260	000000010001
275	000000000010
350	000000000100
400	000000001000

# Using Bitmap indexes

- Example query:

Find people (who by jewelry) such that age = 50 and salary = 100

- Answer:

	123456789012
	=====
Bitmap index for age = 50	= 001110000010
Bitmap index for salary = 100	= 000100000000
	-----
	AND: 000100000000
Record #4	

## Multi-dimensional nature of Bitmap indexes

- There are some multi-dimensional queries that can be answered efficiently using bitmap indexes

# 1) Partial Match queries using Bitmap indexes

- Query: Find people (buyers of jewelry) whose age = 50
- Solution:

Bitmap index for age:

Value	123456789012
-------	--------------

25	100000001000
30	000000010000
45	010000000100
50	001110000010
60	000000000001
70	000001000000
85	000000100000

<----- These people

Records: 3, 4, 5 and 11

## 2) Range Match queries using Bitmap indexes

- Query: Find people (buyers of jewelry) where  $45 \leq \text{age} \leq 55$ ,  $100 \leq \text{salary} \leq 200$
- Solution:

Bitmap index for age:

Value	123456789012	
25	100000001000	
30	000000010000	
45	010000000100	$45 \leq \text{age} \leq 50$
50	001110000010	
60	000000000001	
70	000001000000	
85	000001000000	

Or value = 011110000110

## 2) Range Match queries using Bitmap indexes

- Query: Find people (buyers of jewelry) where  $45 \leq \text{age} \leq 55$ ,  $100 \leq \text{salary} \leq 200$
- Solution:

Bitmap index for salary:

Value	123456789012
60	110000000000
75	001000000000
100	000100000000
110	000001000000
120	000010000000
140	000000100000
260	000000010001
275	000000000010
350	000000000100
400	000000001000

$100 \leq \text{salary} \leq 200$

Or value = 000111100000

## 2) Range Match queries using Bitmap indexes

- Query: Find people (buyers of jewelry) where  $45 \leq \text{age} \leq 55$ ,  $100 \leq \text{salary} \leq 200$
- Solution:

$(45 \leq \text{age} \leq 50)$  AND  $(100 \leq \text{salary} \leq 200)$ :

123456789012

=====

011110000110

000111100000 (AND)

-----

000110000000

Records: 4 and 5

# Compression

- Observation

- Each record has one value in indexed attribute
- For  $n$  records and domain of size  $|D|$ 
  - If  $D$  is large  $\Rightarrow$  1's in a bit-vector will be very rare (Only  $\frac{1}{|D|}$  bits are 1)
  - $\Rightarrow$  waste of space

- Solution

- Compress data. Uncompressed indexes are large
- Need to make sure that **and** and **or** are still fast

# Types of Compression

- Lossy
  - Compression that involves the removal of data.
- Lossless
  - Compression that involves no removal of data

## Run length encoding (RLE)

- very simple form of lossless data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.
- Instead of actual 0-1 sequence encode length of 0 or 1 runs
- One bit to indicate whether 0/1 run + several bits to encode run length
- Useful for compressing data that contains repeated values e.g. output from a filter, many consecutive values are 0.
- Very simple compared with other compression techniques
- Reversible (Lossless) compression decompression is just as easy
- But how many bits to use to encode a run length?
  - Gamma codes or similar to have variable number of bits

## Run length encoding (RLE): Example

- *AAAAAAA BBB CCCCC*  $\Rightarrow$  7A 3B 5C
- *0001 0000 1110 1111* (**2 bytes**)  
 $\Rightarrow$  3, 1 , 4, 3 , 1 , 4 (**6 bytes**)
- if we use one byte to encode a run we have 7 bits for length = max run length is 128(127)
- If there are no repeating values in the data, RLE will double the amount of the numbers compared with the original data

## Run length encoding (RLE)

- Compression effectiveness depends on input
- Must have consecutive runs of values in order to maximize compression
  - Best case: all values same
    - Can represent any length using two values
  - Worst case: no repeating values
    - Compressed data twice the length of original
- Should only be used in situations where we know for sure have repeating values

# Elias Codes<sup>1</sup>

- Gamma Codes

- Represent a positive integer  $x$  with  $\lfloor \log_2 x \rfloor$  zeros, followed by the **binary representation** of the number
- Not efficient for numbers larger than 15
- *Prefix-free*, because the number of initial zeros tells us exactly how many bits the code come afterwards.
- Code for  $x$  has a length of exactly  $2 \times \lfloor \log_2 x \rfloor + 1$  bits

- example:

$x$	<i>Elias-Gamma</i>
1	1
2	010
3	011
4	00100

---

<sup>1</sup>Slide Credit: Hannah Bast, University of Freiburg

# Elias Codes<sup>2</sup>

- Delta Codes

- Represent a positive integer  $x$  with  $1 + \lfloor \log_2 x \rfloor$  stored as a gamma code.
- This is followed by the **binary representation** of the number **without** the most significant bit (leading 1)
- *Prefix-free*, because the prefix-free Elias-Gamma code tells us exactly how many bits the code come afterwards.

- example:

$x$	<i>Elias-Delta</i>
1	1
2	0100
3	0101
4	01100

---

<sup>2</sup>Slide Credit: Hannah Bast, University of Freiburg

# Hybrid Encoding

- Run length encoding
  - Can waste space
  - And/or run length not aligned to byte/word boundaries
- Encode some bytes of sequence as is and only store long runs as run length
  - EWAH
  - BBC (that's what Oracle uses)

## Bitmap indexes

- Fast for read intensive workloads
  - Used a lot in data warehousing
- Often build on the fly during query processing
  - As we will see later in class