CS525: Advanced Database Organization

Notes 6: Multi-dimensional indexes

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

February 12, 14, 19, 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

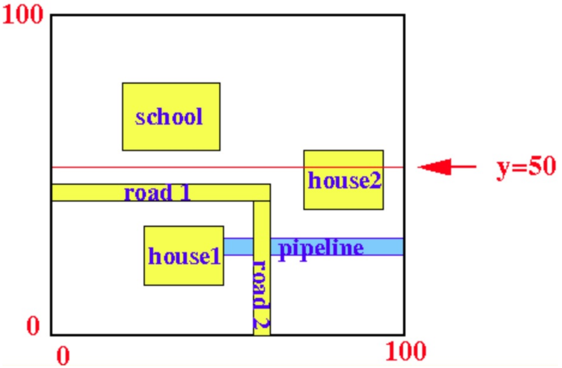
- There are information that are naturally `multi-dimensional`
  - e.g., `Geographic information:`
    - Stores objects in a (typically) two-dimensional space.
    - The objects may be points or shapes.
    - Often, these databases are maps, where the stored objects could represent houses, roads, bridges,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

Multi-dimensional queries

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

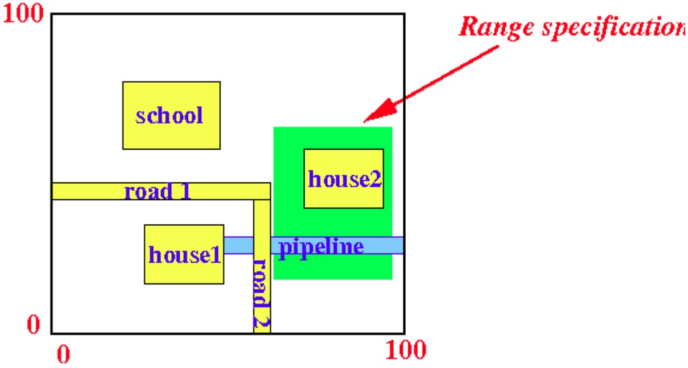
Partial Match queries

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



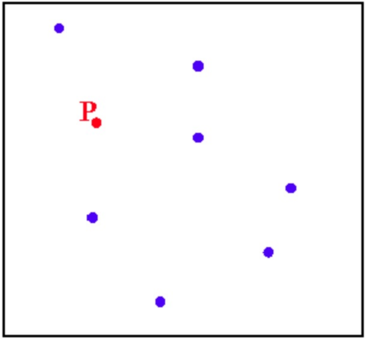
Range queries

- 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:



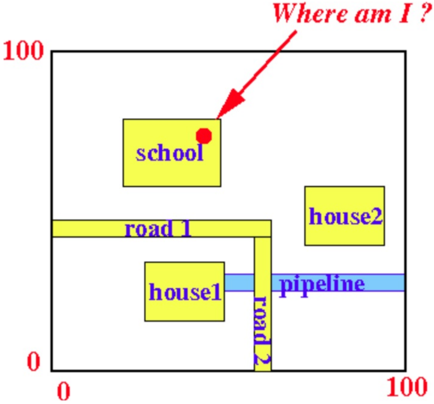
Nearest neighbor queries

- 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

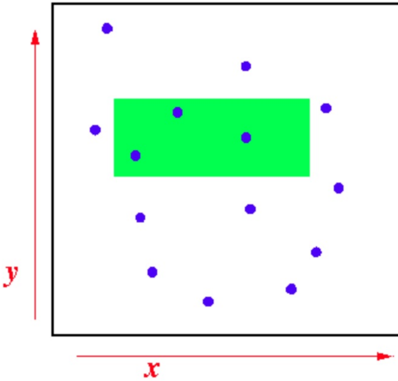


Motivation for developing multi-dimensional indexes

- Are muliti-dimensional indexes necessary?
- Can `one-dimensional index` technique support `geometrical (2-dimensional)` queries efficiently?
- `Case Study`: Try to process a `range query` using a `B-tree index`

Processing the geometrical query using a B-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`

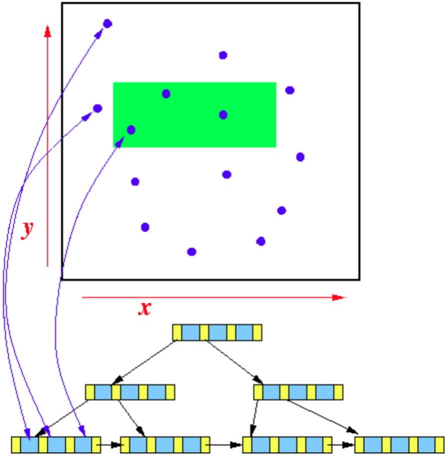


Processing the geometrical query using a B-tree index

- Suppose we have B^+-tree indexes on:
  - The *x*-coordinate attribute of Object and
  - The *y*-coordinate attribute of Object

Processing the geometrical query using a B-tree index

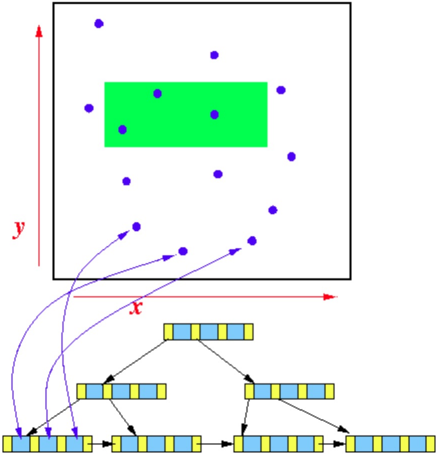
- The B$^+$-tree on the *x*-coordinate information looks like this:



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

Processing the geometrical query using a B-tree index

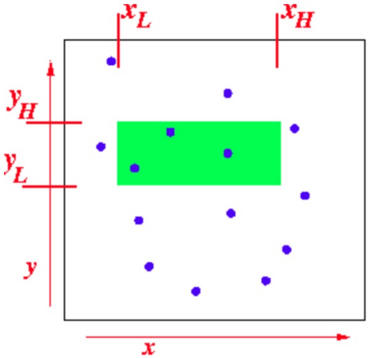
- The `B`$^+$`-tree` on the *y-coordinate* information looks like this:



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

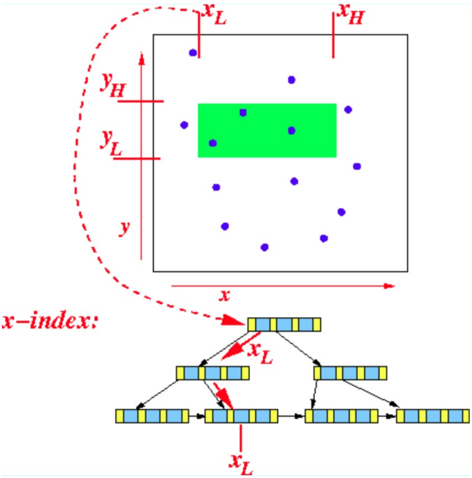
Processing the geometrical query using a `B-tree index`

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



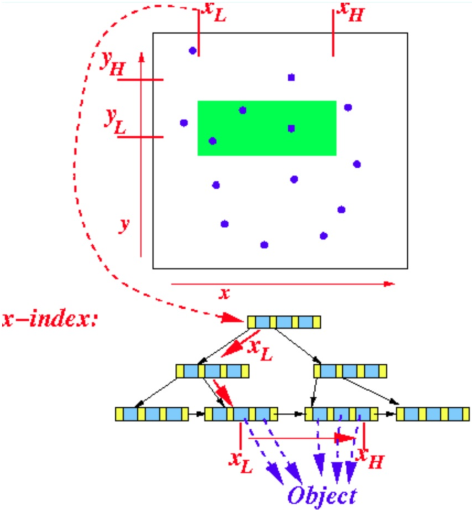
How to use the B^+-`tree` `indexes` to process range query

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

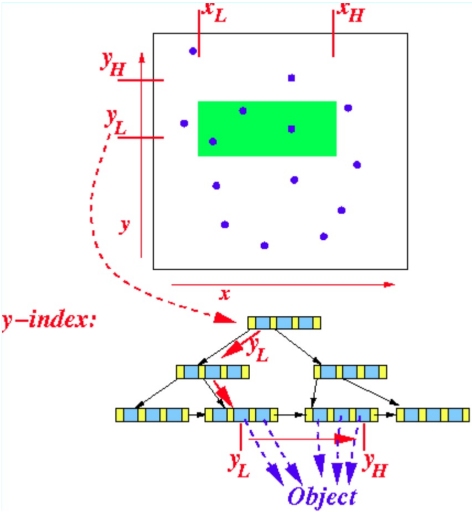


How to use the B+-tree indexes to process range query

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

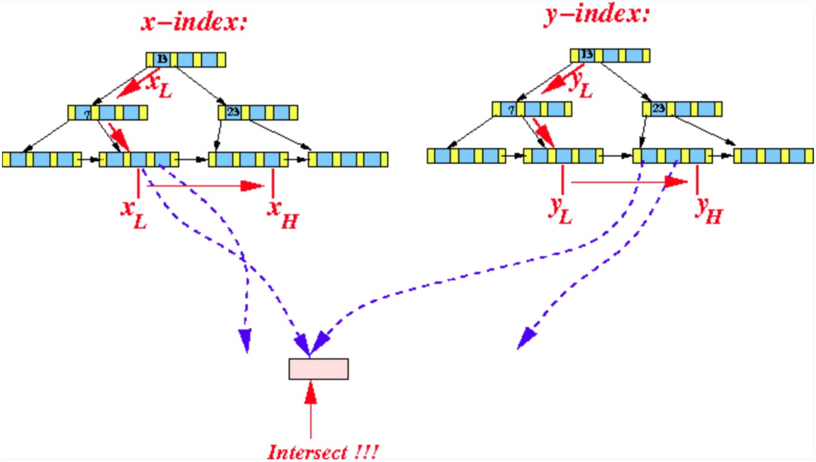


2. Do the same for the *y-coordinate*



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

3. Compute the intersection of the 2 pointer sets



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

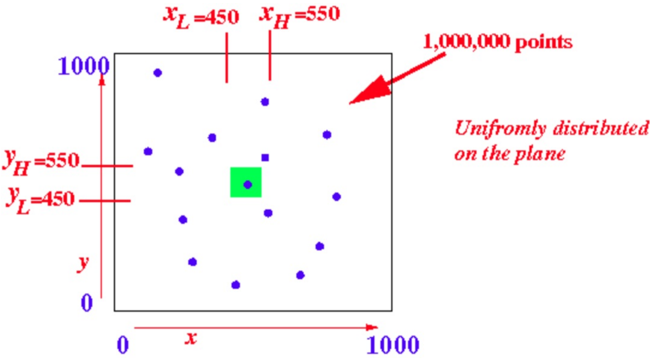
4. Retrieve the records using the record pointers in the intersection

- These records are guarantee 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

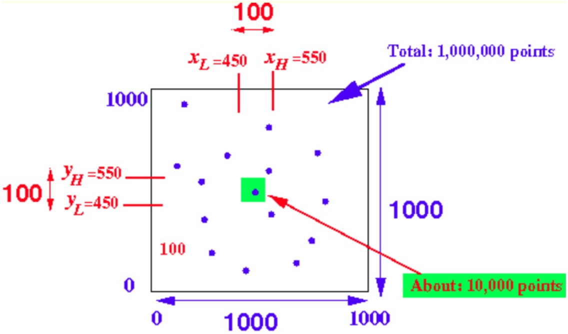
- Consider the following situation:



- Some statistics:
  - `Green area` = $100 \times 100 = 10,000$
  - `Total area` = $1000 \times 1000 = 1,000,000$
  - `Green area` = $0.01 \times \mathit{Total\ \ area}$

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 $[450, 550]$

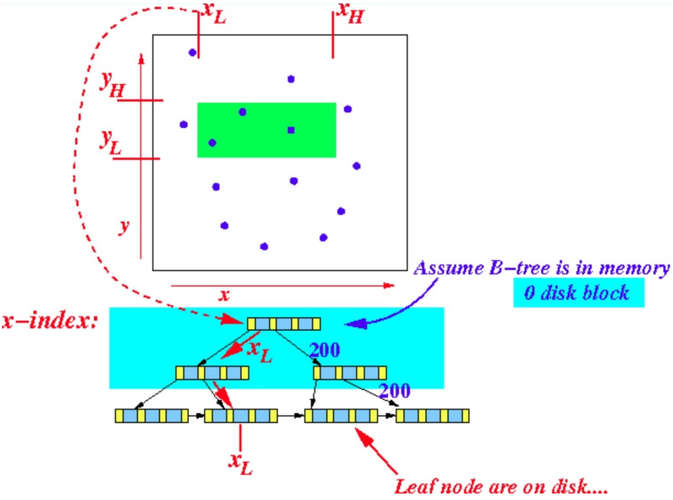
$$\cong 0.1 \times 1,000,000 = 100,000$$

Storage information

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

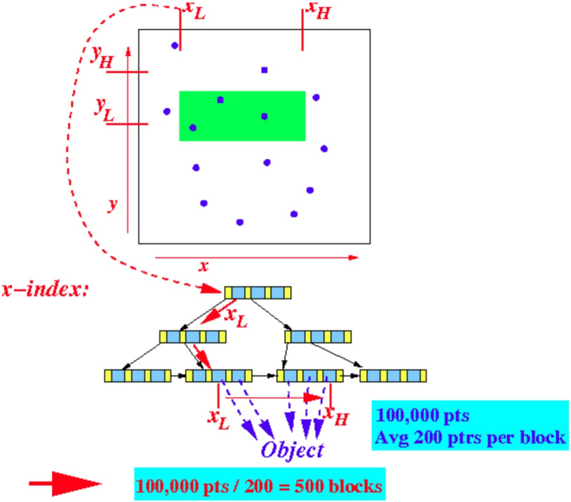
Compute the Processing Cost

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



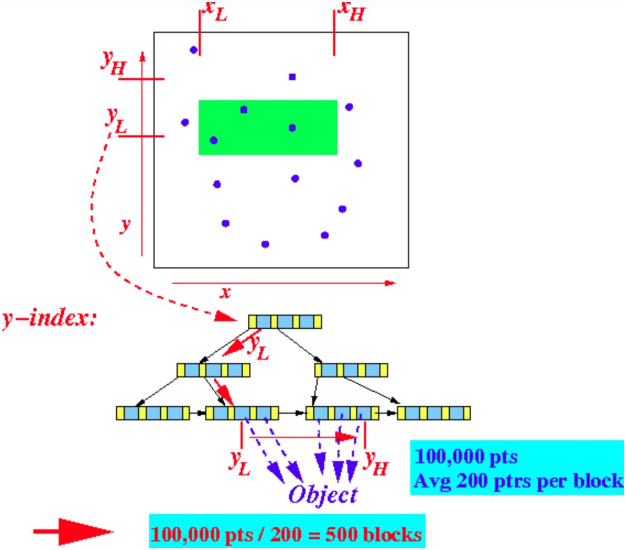
Compute the Processing Cost

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

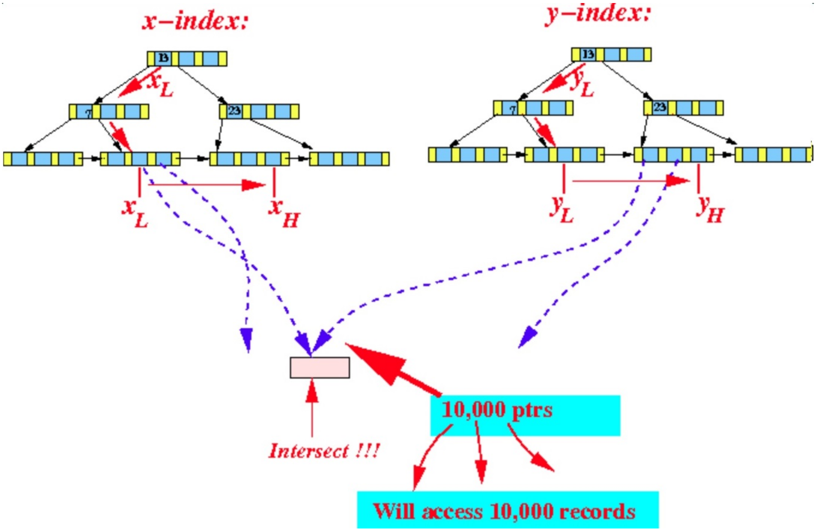


Compute the Processing Cost

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



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
- $\Rightarrow$ using the `B-tree index` does not help us improve performance

Conclusion

- We cannot store geographically ``related'' data randomly
  - If related geographical data is store 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 the geometrical data
  - Need a more appropriate index structure for multi-dimensional data

Multi-dimensional index structures

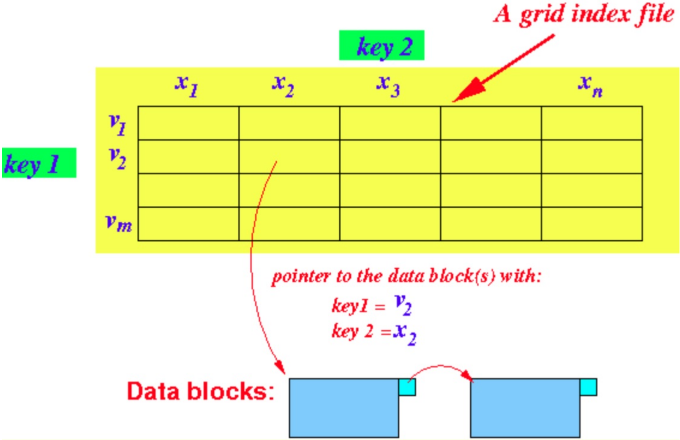
- Hash like structures
  - Grid files
  - Partitioned Hashing functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

Grid Index

- 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

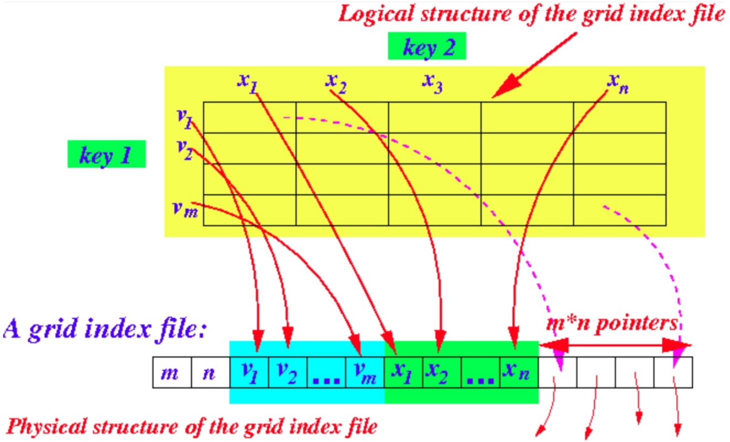
- `Grid index file:` an index that is organized into a 2-dimensional structure



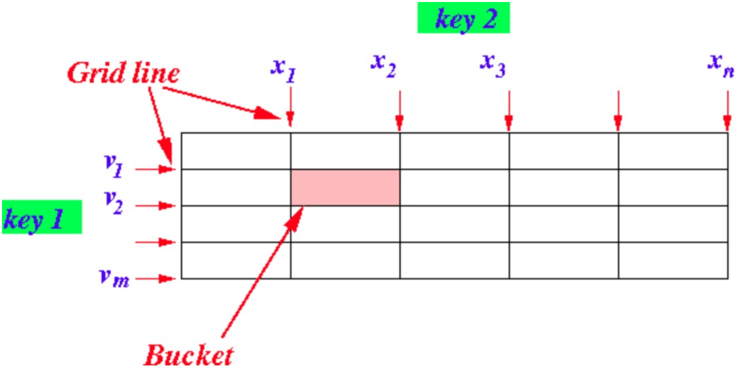
- Note: `Geographically ``related'' data` (i.e.: points that are close to each other) are stored in the same `data block`

Storage Structure of Grid Index File

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



Buckets and Grid lines

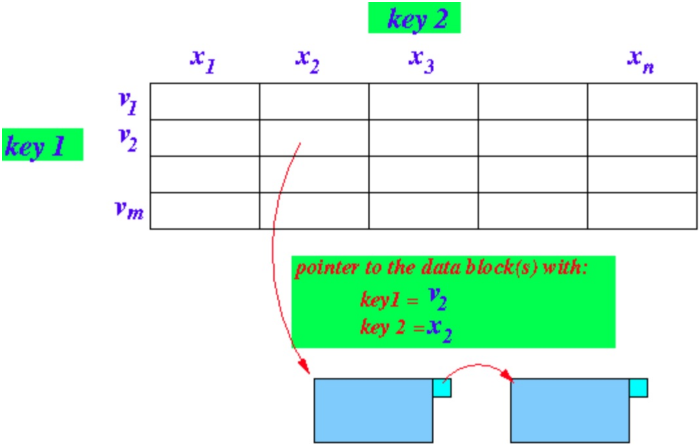


Interpreting the grid lines

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

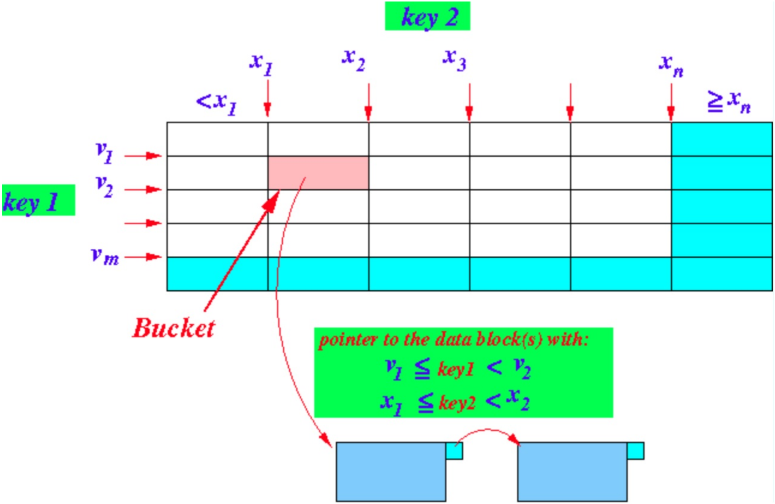
1) As individual points
2) As intervals

Interpreting the grid lines: Point interpretation



- The grid lines represents 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

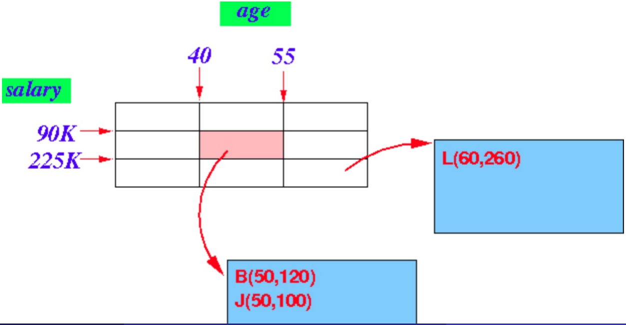
- Data on `people` who buy jewelry:

```
(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

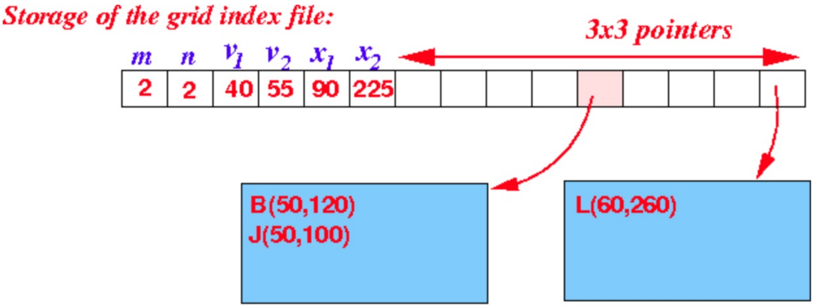
Age:	0-40	40-55	≥ 55+
Salary:	0-90K	90K-225K	≥ 225K+

- Grid index file



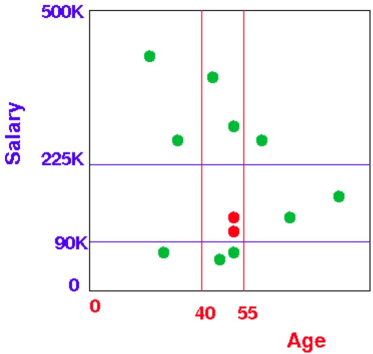
Example of a Grid index file

- How the grid index file is stored:



Example of a Grid index file

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

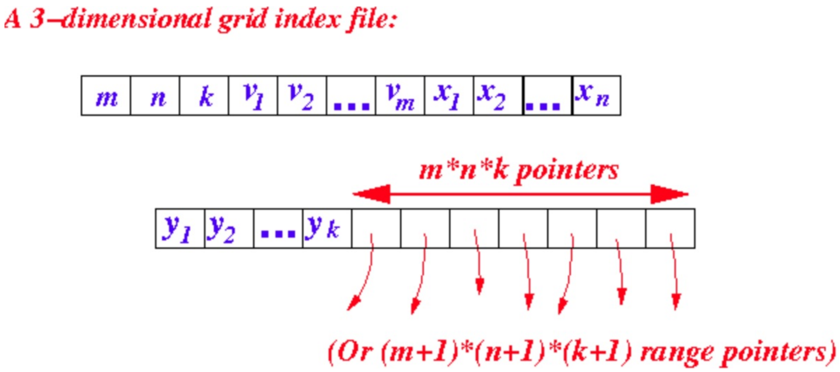


- For the following data set

(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

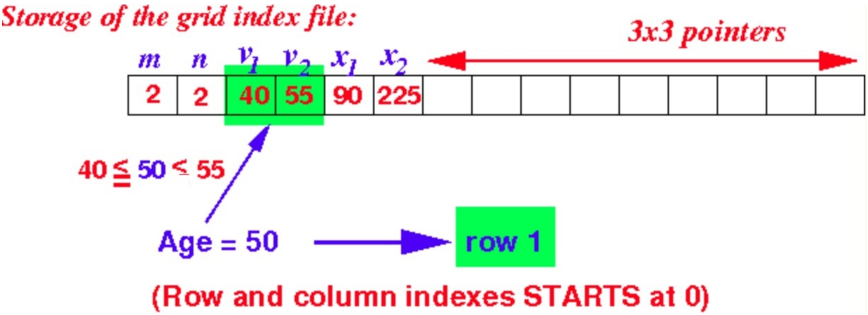


Lookup a search key

- **Given:** `search key (Age = 50, Salary = 100)`
- `How to find this record`

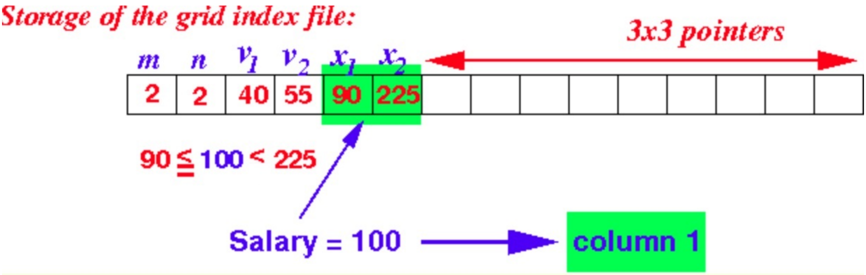
Lookup a search key

- Find the `row index` using `age = 50`



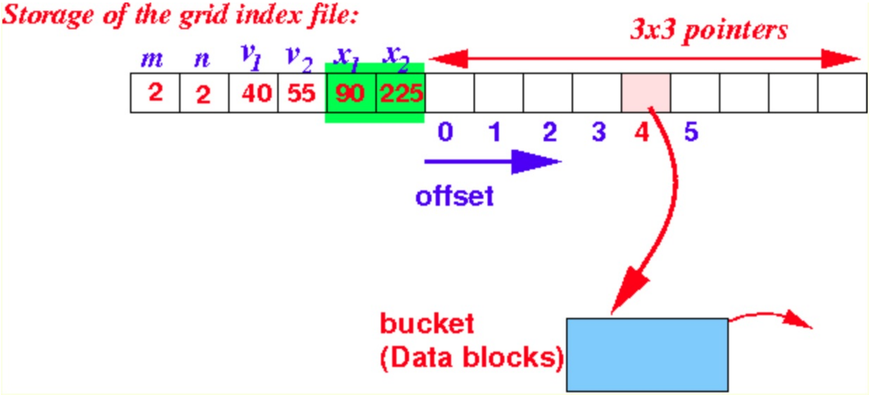
Lookup a search key

- Find the `column index` using `salary = 100`



Lookup a search key

- Find the `offset` (this is the standard way to find an array element)
- `offset = row index` $\times$ `(column width) + column index` $=$ $1 \times 3 + 1 = 4$
- Access the `blocks` and search for the record



Insert a new record in a Grid Index file

Algorithm 1 insert(record)

```
Lookup (record.SearchKey)
Let b = the last bucket block
if  b has room for record then
    Insert record in block b
else
    Allocate an overflow block for bucket
    Link overflow block to b
    Insert record in overflow bucket block
end if
```

Performance Analysis: lookup/insert a search key

- Assumption: The grid index file can be store 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

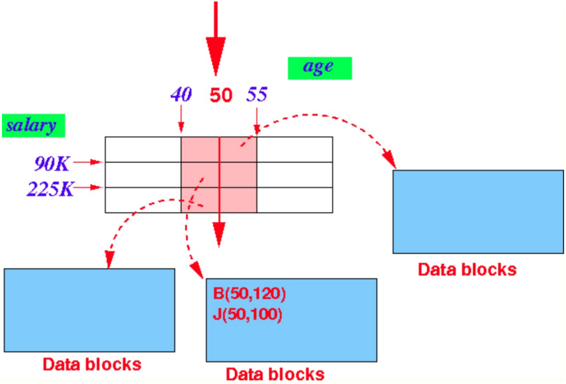
- Assumption: The grid index file can be store in memory
- Insert performance
  - In addition to the lookup cost
  - 1 more `block write operation` to update `the bucket block`
  - If `overflow`, need to update the overflow link in the `bucket` and write `an overflow block`)

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

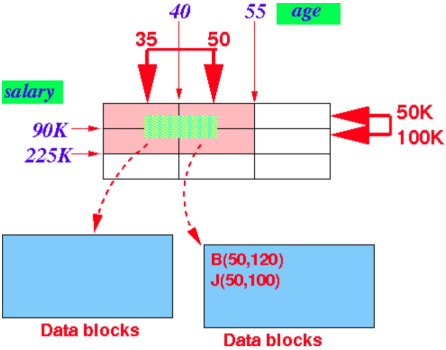
- The query specifies conditions on some dimensions but not on all dimensions
- Find all jewelry purchases by people with age = 50



- 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 $35 \leq age \leq 50$, $50K \leq salary \leq 100K$



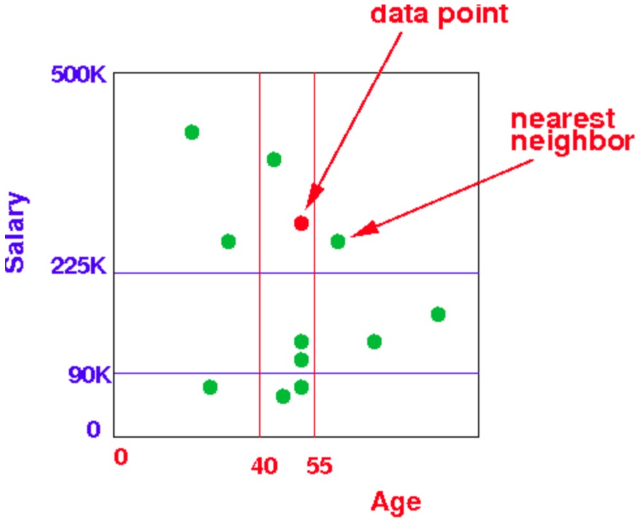
- In this example, we must access 4 `disk blocks`

Announcement

- Coding assignment 2 due date: Sunday, March 11, 2018 by midnight (Chicago time:)
- Quiz 1:
  - Post: Friday February 23.
  - Due on Blackboard: Tuesday February 27 by midnight (Chicago time)
- Midterm: Close notes/book/friends: March 5 in class time

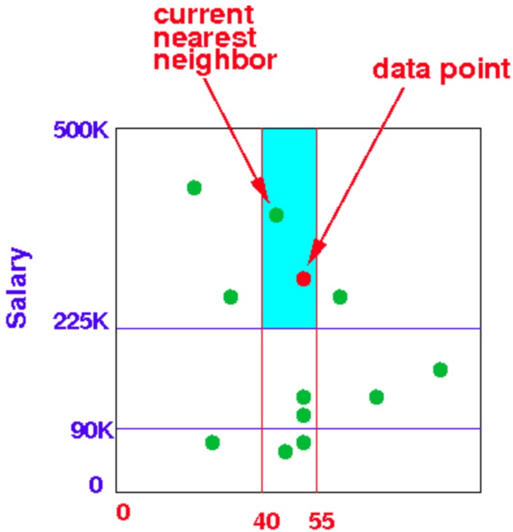
3) Nearest neighbor queries

- Find the nearest neighbor of a data point



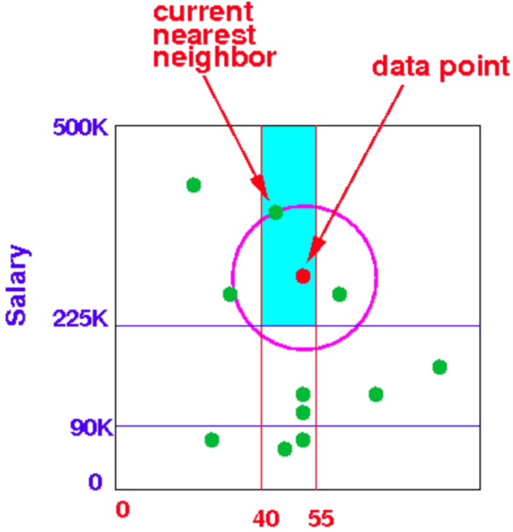
3) Nearest neighbor queries

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



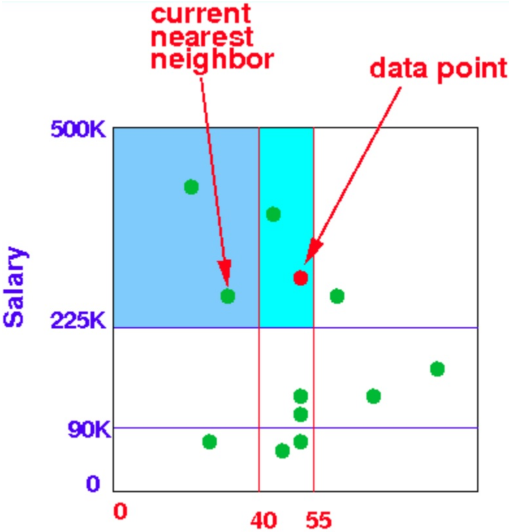
3) Nearest neighbor queries

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



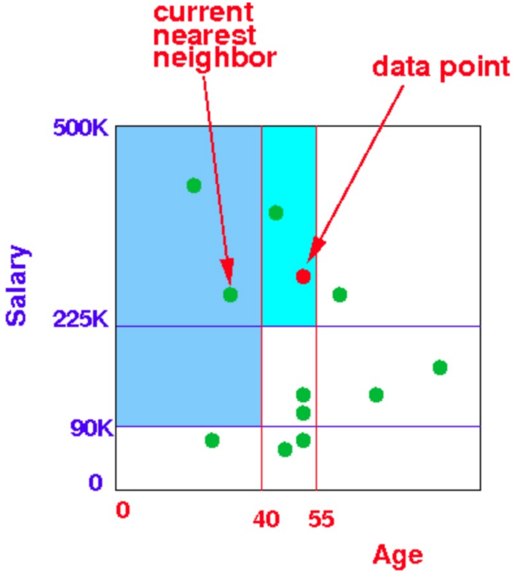
3) Nearest neighbor queries

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



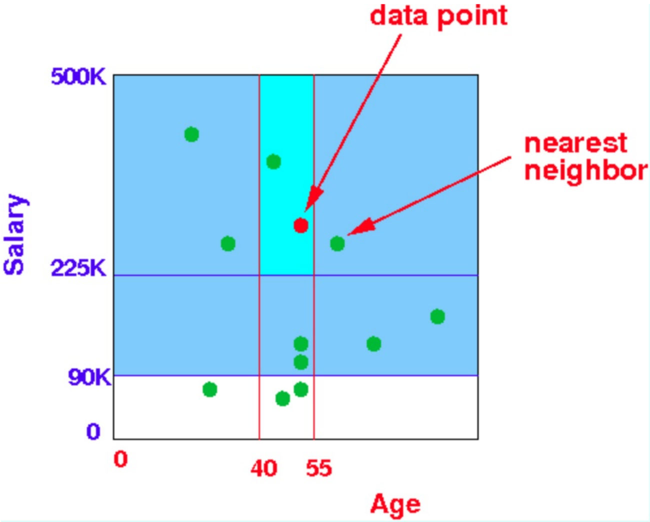
3) Nearest neighbor queries

- And so forth



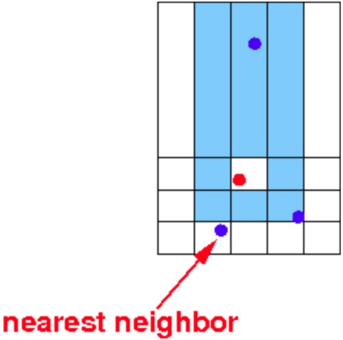
3) Nearest neighbor queries

- And so forth



3) Nearest neighbor queries

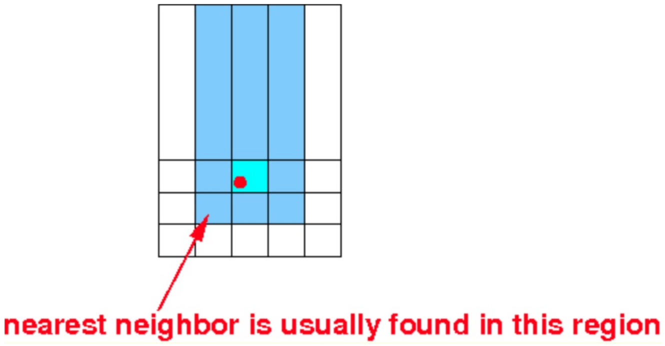
- Note: You may need to expand the search range beyond the adjacent regions



- 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 expend the range of the search

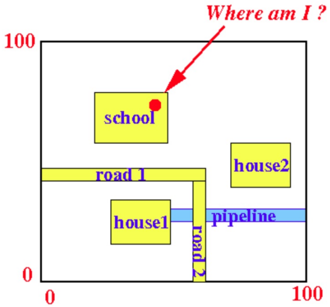
3) Nearest neighbor queries: Performance

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



4) Where-am-I queries

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



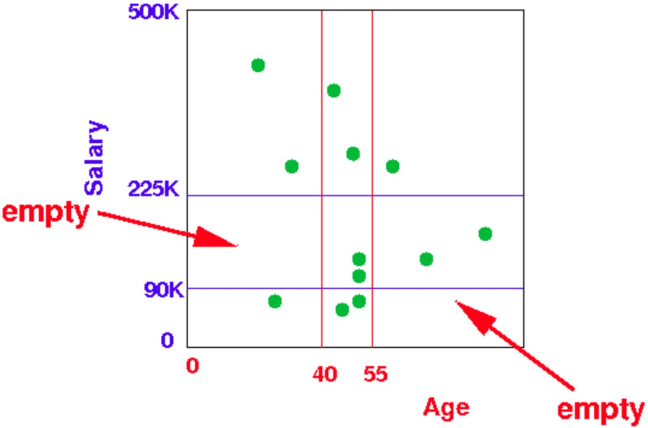
- `Grid index` cannot represent objects (can only present points)
- ⇒ `Grid Index` cannot handle `Where-am-I` type of queries
- The only kind of index that can handle `Where-am-I queries` is the `R-tree (Region-tree)` (Discussed later)

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`



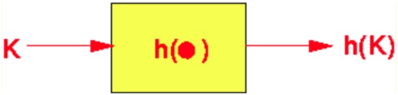
- Especially when you have 3 or more dimensions. You will have many buckets that are empty.

Multi-dimensional index structures

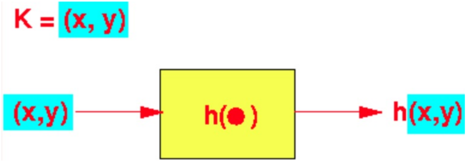
- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

Partitioned Hashing

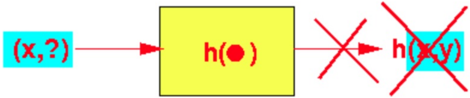
- Traditional hashing



- Problem with traditional hashing
  - If the key is composite



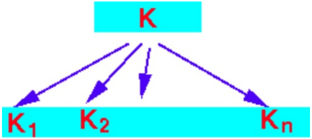
  - and some component of the key is not known



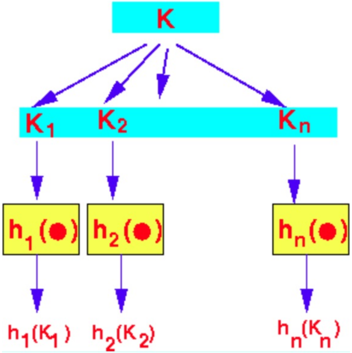
  - we cannot compute a meaningful hash value at all

Partitioned Hashing

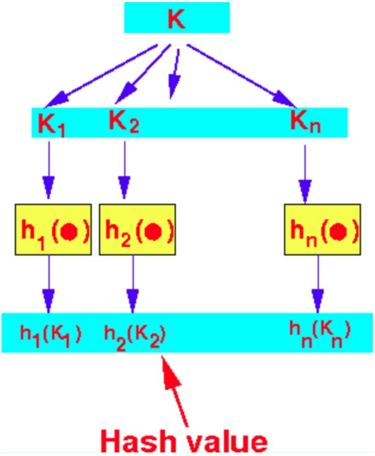
- Partitioned Hashing
  - The key is a composite:



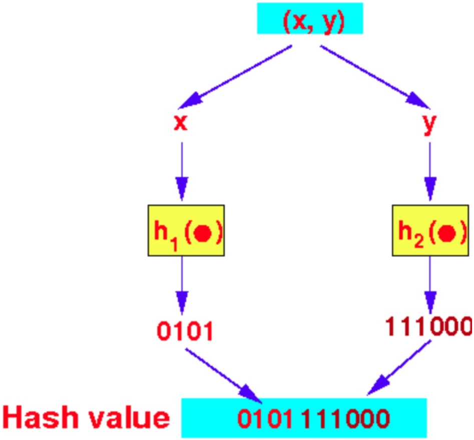
  - Use n `hash functions`, one function on one component



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

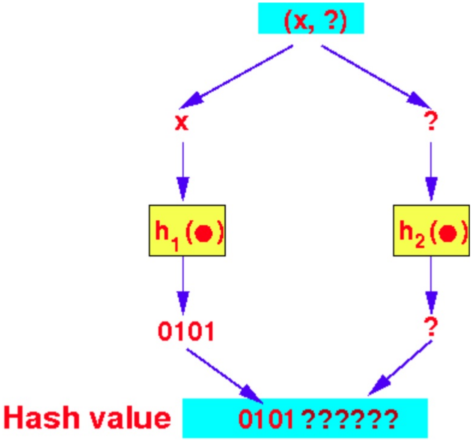


Partitioned Hashing: Example



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

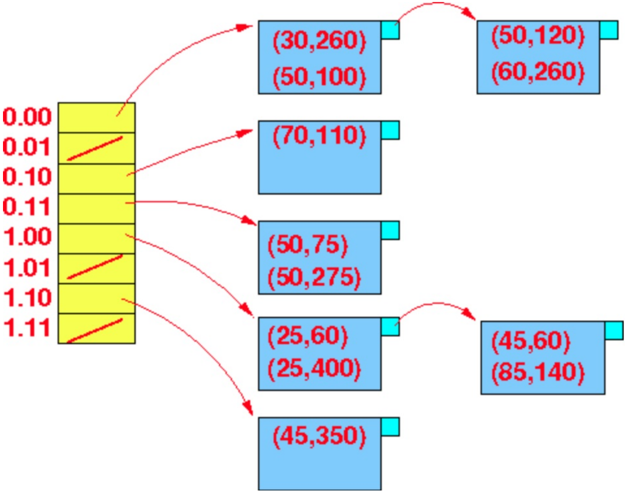
Age:	$h_1($ age $)$	=	age % 2
Salary:	$h_2($ salary $)$	=	salary % 4

- 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)
```

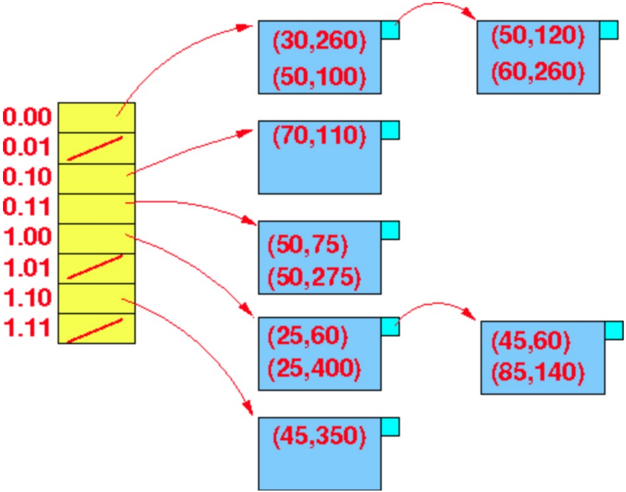
Partitioned Hashing: A complete example

- The Partitioned Hash index



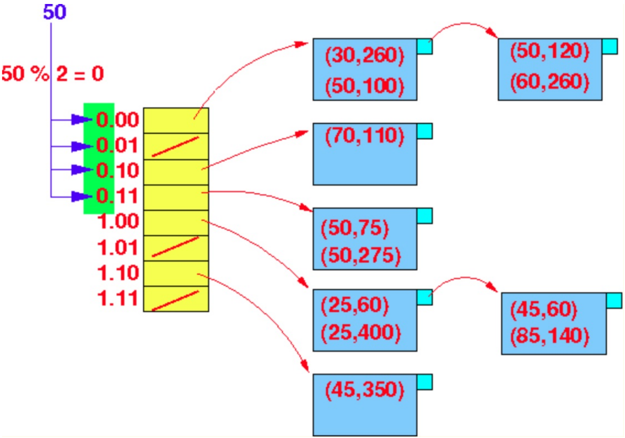
Using a Partitioned Hashing

- The Partitioned Hash index



1) Partial Match queries

- Find people with age = 50



- Age = 50 will hash to the hash value Hash(age) = $0 \times \times$.
- 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: $35 \leq age \leq 50, \ 50K \leq salary \leq 100K$

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

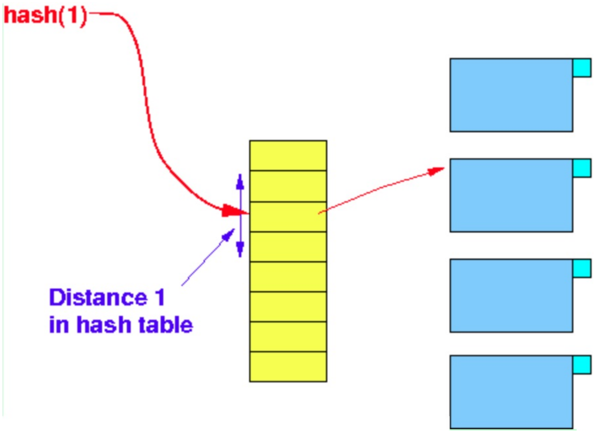
b) Collect all the buckets (eliminate duplicate block pointers)

c) Access all (unique) buckets (disk blocks)

- ⇒ Hashing is not appropriate for range type queries

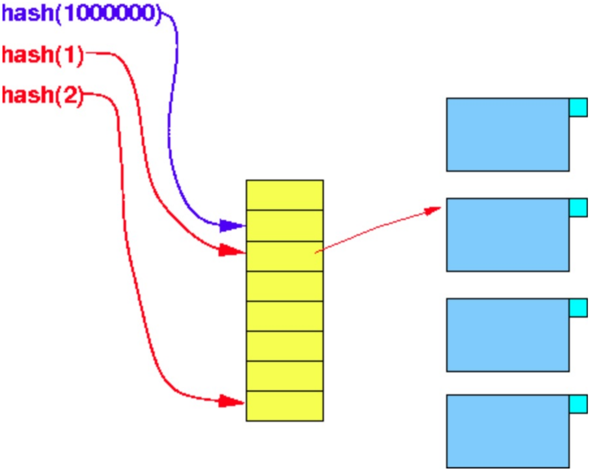
3) Nearest neighbor queries

- 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 ≤ 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

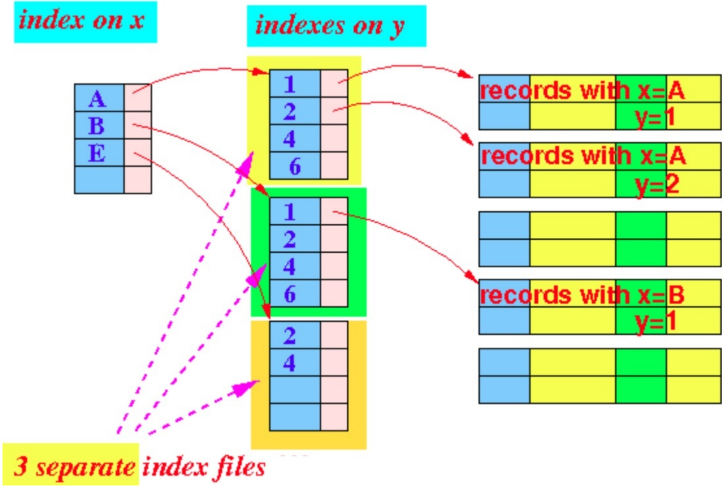
- Good hash functions will randomize the records
- $\Rightarrow$ Partitioned hashing will achieve good occupancy rate per bucket

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

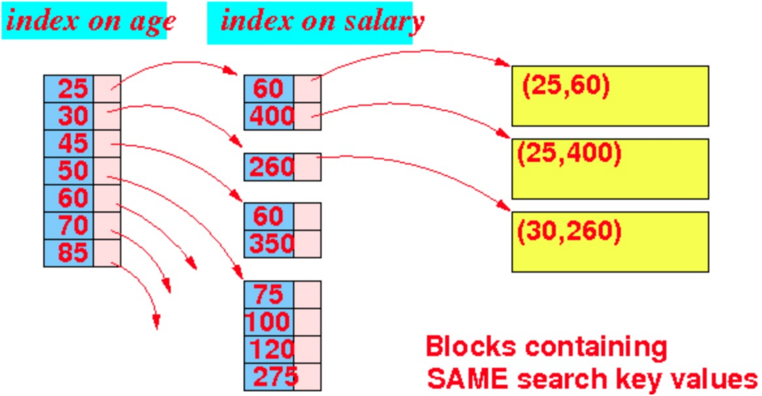


Multiple-key index: Example

- Data on people who buy jewelry

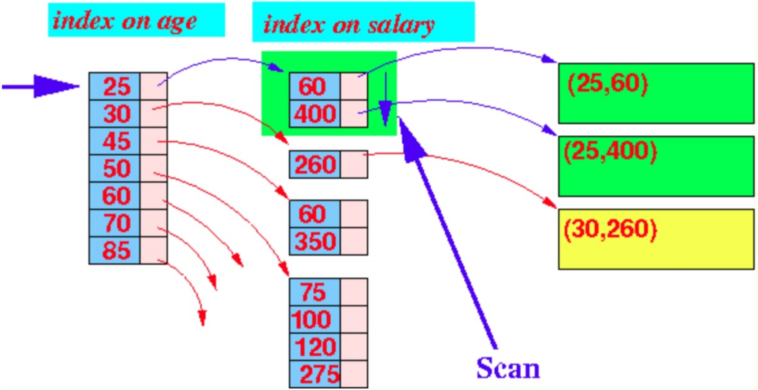
```
(age, salary (in $1,000))

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)



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

- Find all people with age =25



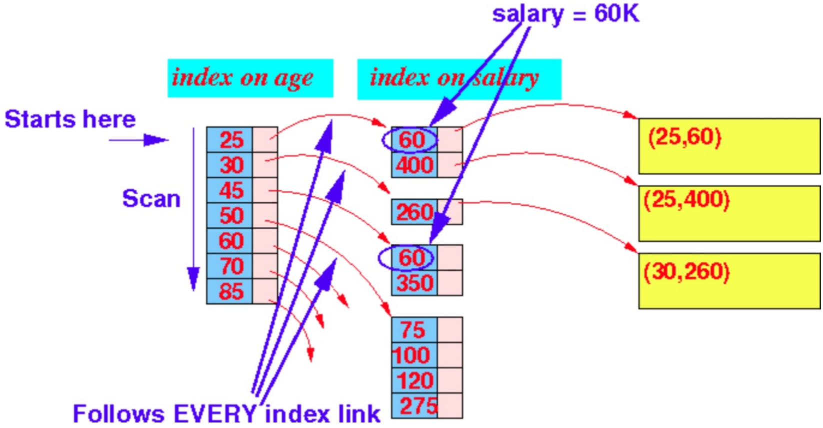
- Use the index on age to find the index block(s) for age = 25
- Then, scan all entries in the `salary index` file (list of blocks) indexed by age= 25 to find the records

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

1) Partial Match queries

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

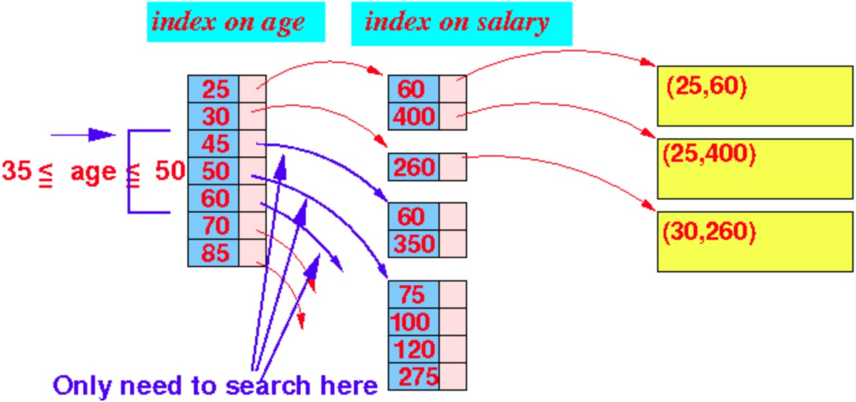


- Result: many disk accesses

2) Range queries

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

- 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

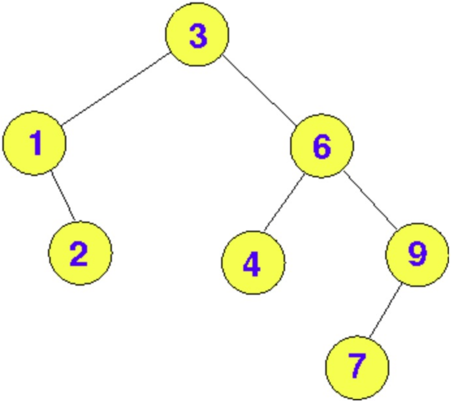
kd (k-dimensional) tree:

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

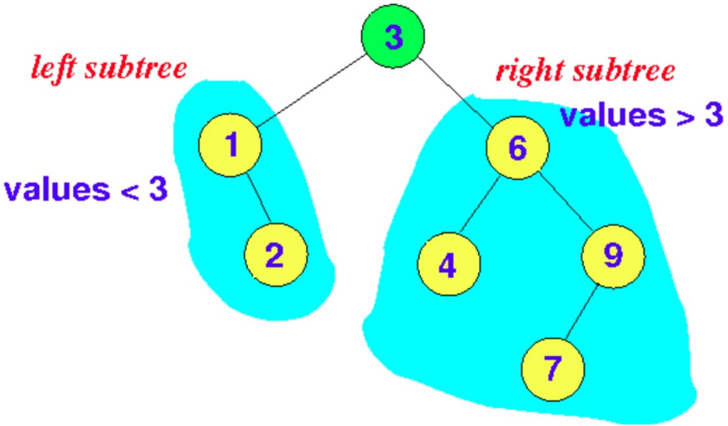
Review: Binary Search Tree

- `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

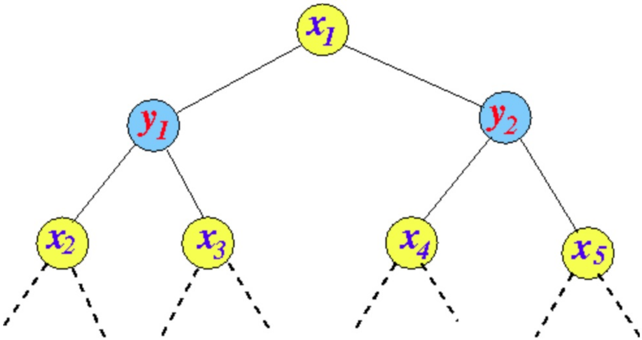


The kd-tree

- 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)

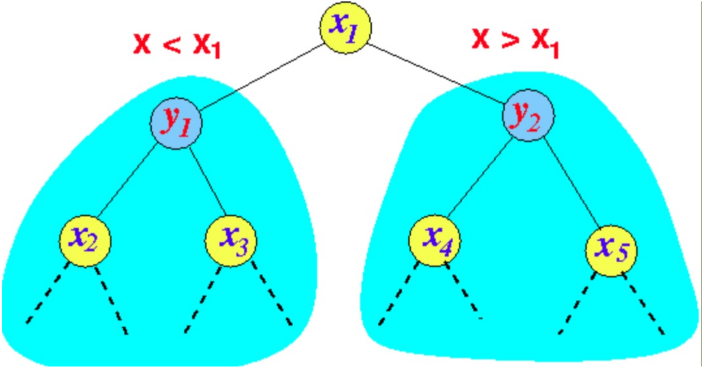
Example: a 2-dimensional kd-tree

- 2 dimentions: x and y



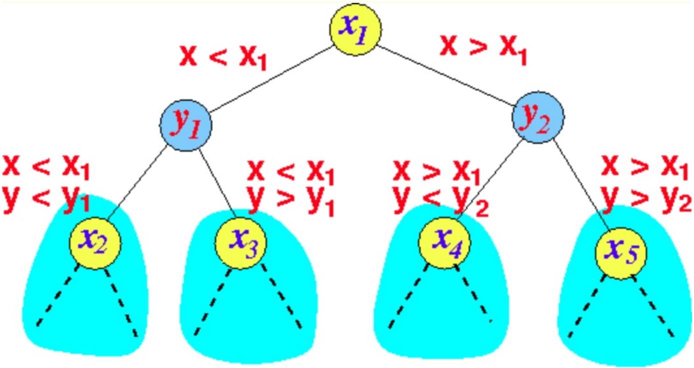
Properties

- `Subtrees` of x_1 must satisfy this property



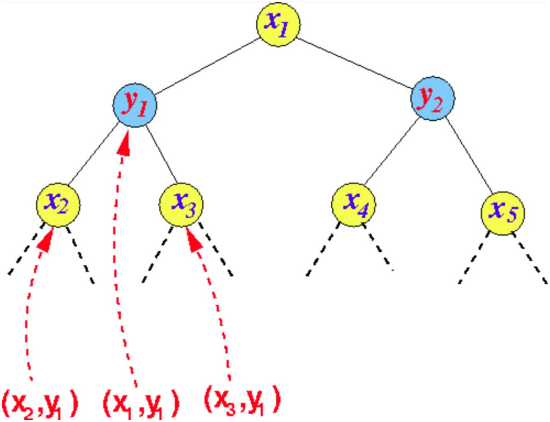
Properties

- Subtrees of y_1 and y_2 must satisfy this property



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

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

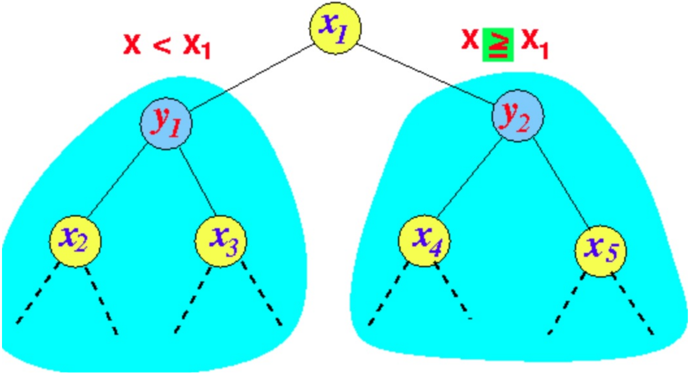


- The node y_1 contains the data (record) for (x_1,y_1)
- The node x_2 contains the data (record) for (x_2,y_1)
- And so on

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.: $\mathtt{x}_1$ or $\mathtt{y}_4$) of the attribute
  - Pointers to the (2) children nodes

- 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

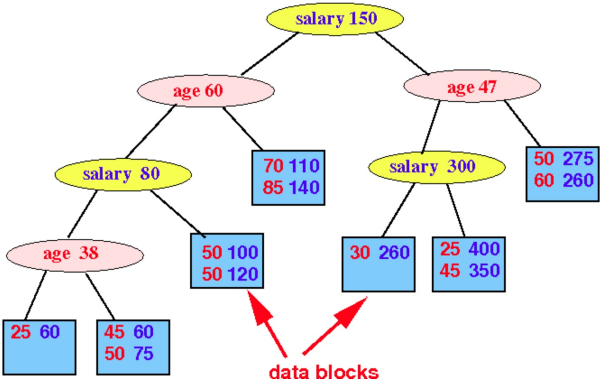
Example kd-tree

- Data on people who buy jewelry

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

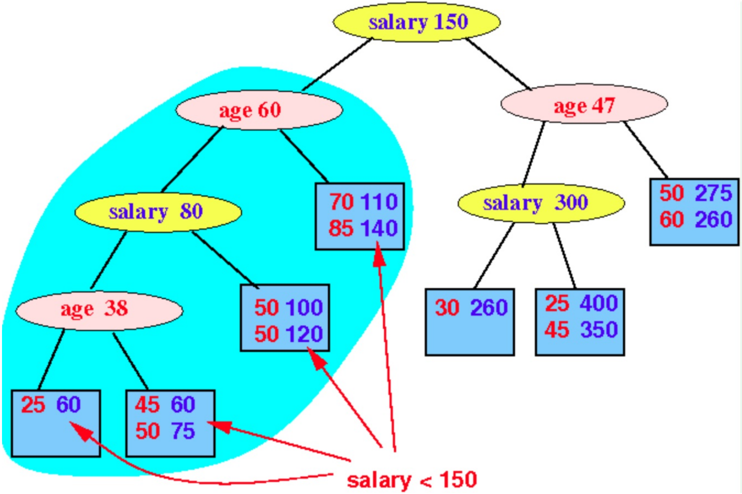
A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

- A kd-tree for the data:



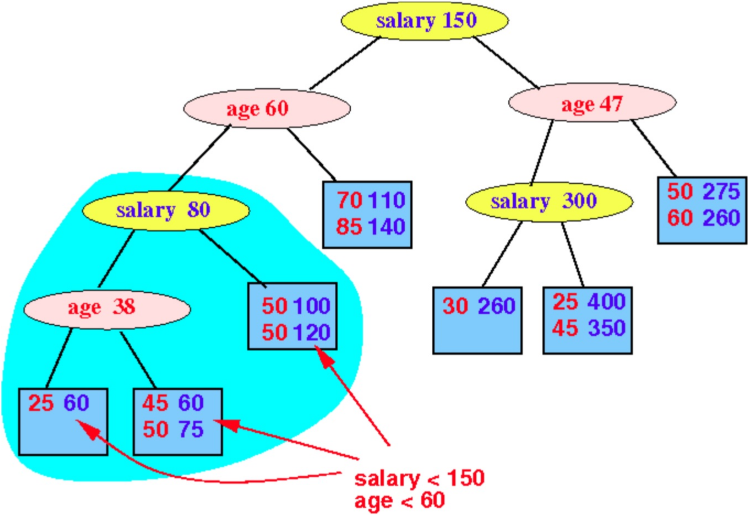
Example kd-tree

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



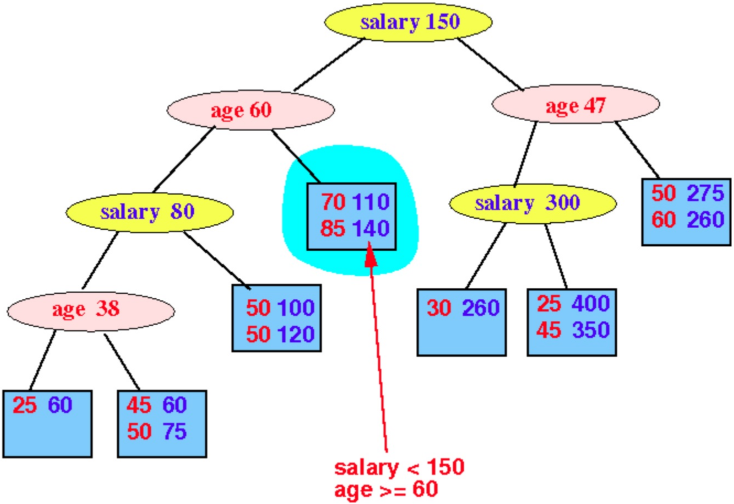
Example kd-tree

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



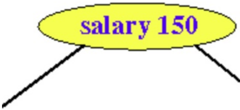
Example kd-tree

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

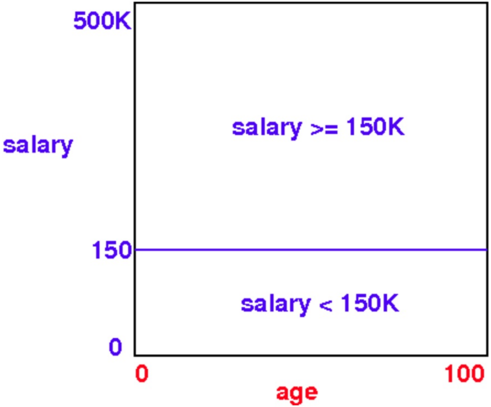


How a kd-tree partitions the data space

- The root node

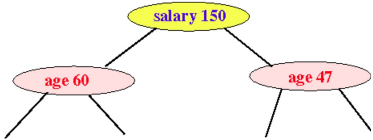


- partitions the data space in half

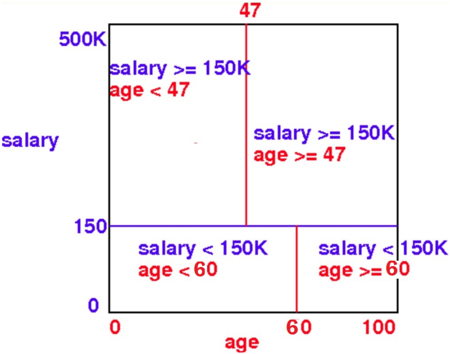


How a kd-tree partitions the data space

- The age nodes at level 2

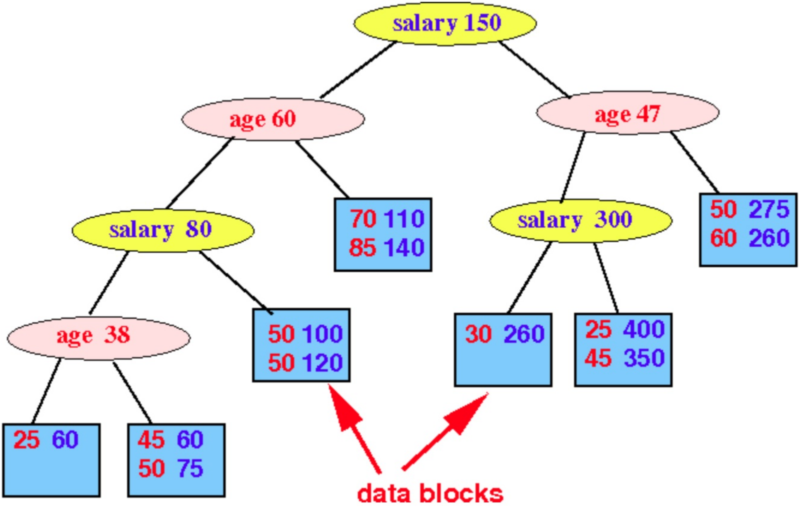


- partitions each sub-space in half



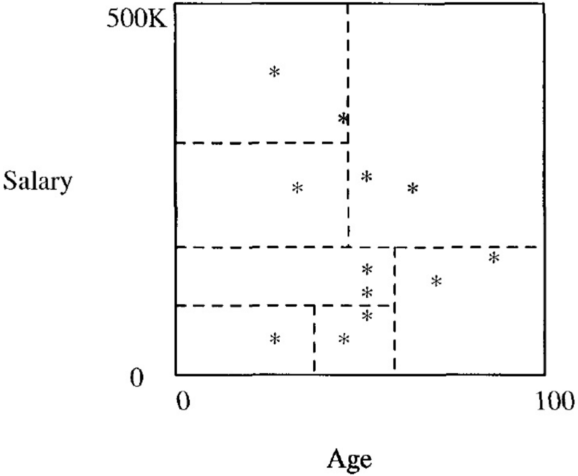
How a kd-tree partitions the data space

- This kd-tree



How a kd-tree partitions the data space

- will divide the data space up as follows



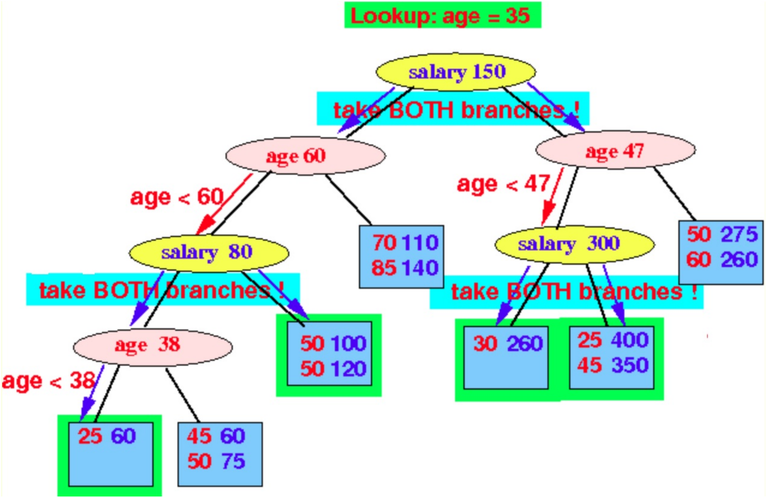
Using kd-tree for common multi-dim queries

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

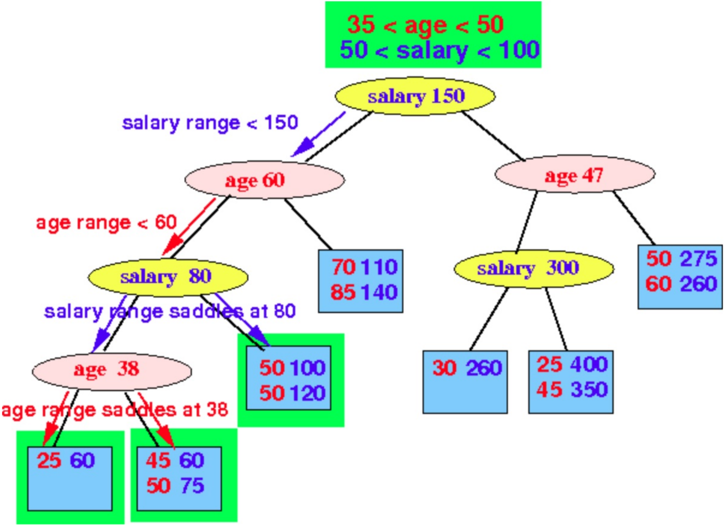
- Find `all person` with age = 35



2) Range queries

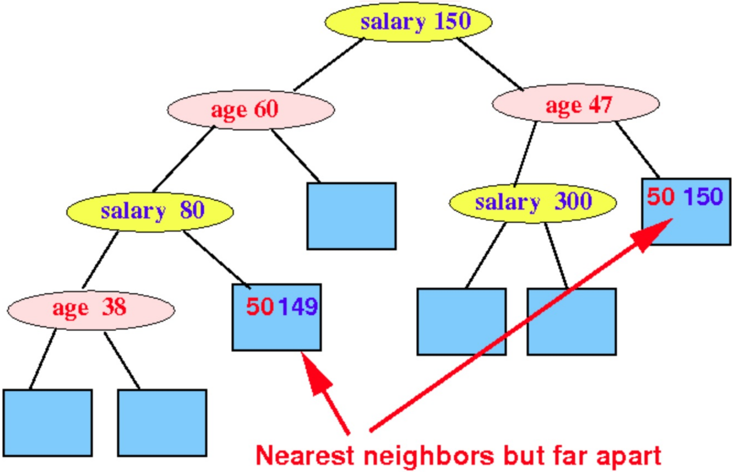
- Search Algorithm
  - For the `search range` is `completely contained` by the `left subtree`, then
    - Take only the `left branch` of the `subtree` for the `search value`
  - For the `search range` is `completely contained` by the `right subtree`, then
    - Take only the `right branch` of the `subtree`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

2) Range queries: Example



3) Nearest neighbor queries

- Not easy to 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`

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 nodepointers` (n way split)
    - One `parent node` pointer (except for the root node)
  - The child node pointers will point to every possible combination of $<$ and $\geq$ relationships with the `search key` values

Quad-tree on common multi-dimensional queries

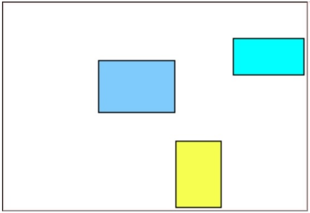
- A `quad-tree` is similar to a `kd-tree`
- The techniques discussed in the `kd-tree` applies to the `Quad-tree`

Multi-dimensional index structures

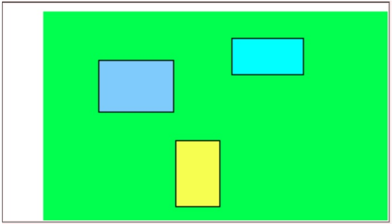
- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

The R-tree (Region-tree)

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

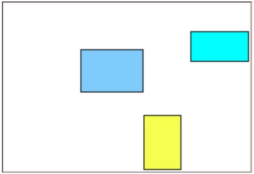


- The Bounding Box for this group of objects

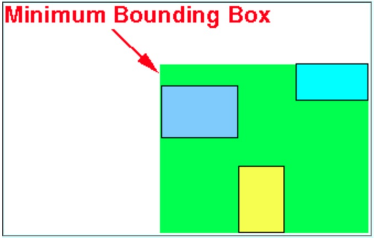


The R-tree (Region-tree)

- 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)

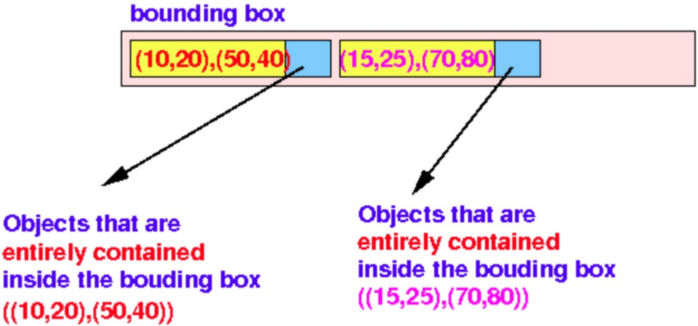
- **Note:** A `rectangle` can be represented as follows
  - coordinate of the lower left corner
  - coordinate of the upper right corner
- Example: `Rectangle:` $\big($`(10,20), (50,40)`$\big)$

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: $\Big(\big(\texttt{(10,20),(50,40)}\big)\texttt{,ptr1}\Big)$
- The `leaf nodes` contains a number of entries of the following format:
  - **(min bounding box, object pointer)**
  - Example: $\Big(\big(\texttt{(10,20),(50,40)}\big)\texttt{,house-ptr}\Big)$

Property of a R-tree

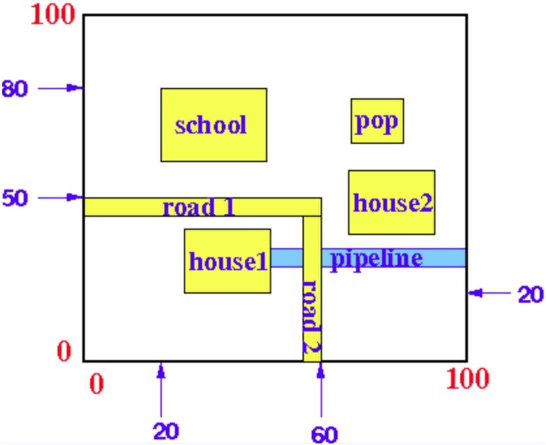
- An internal node of the R-tree has the following structure



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

R-tree: Example

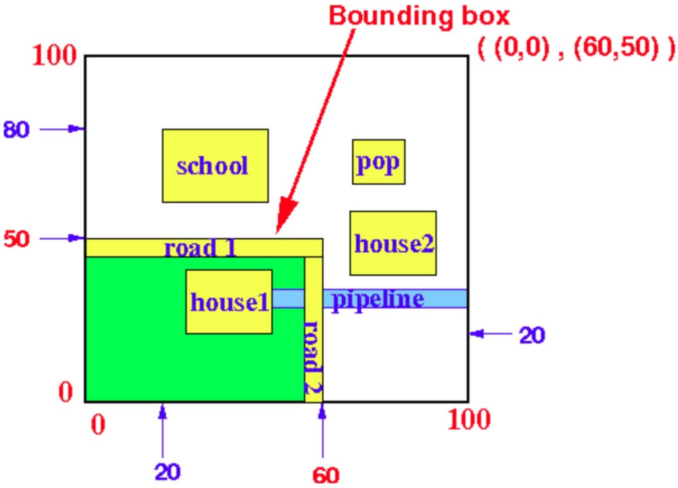
- Objects that we want to represent



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

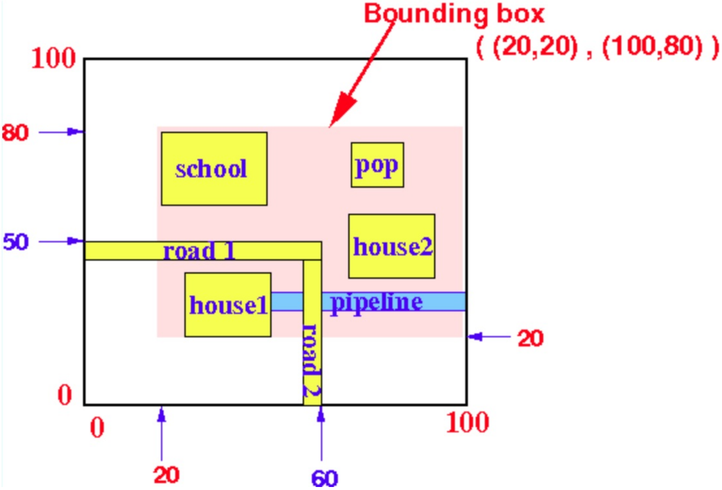
R-tree: Example

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



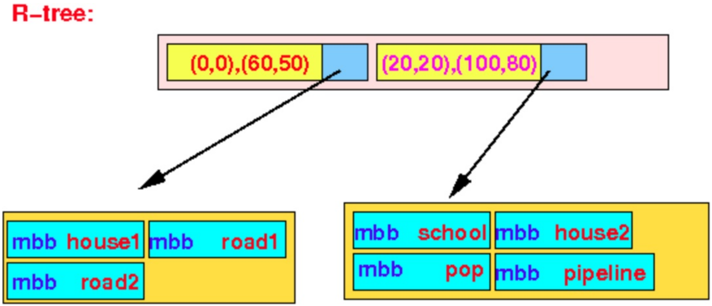
R-tree: Example

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



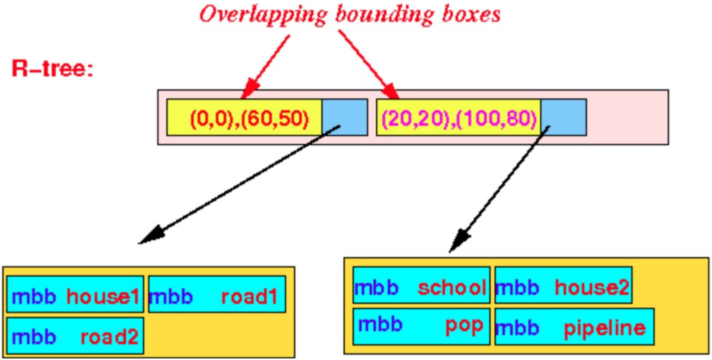
R-tree: Example

- The `R-tree` that uses the previous `bounding boxes`



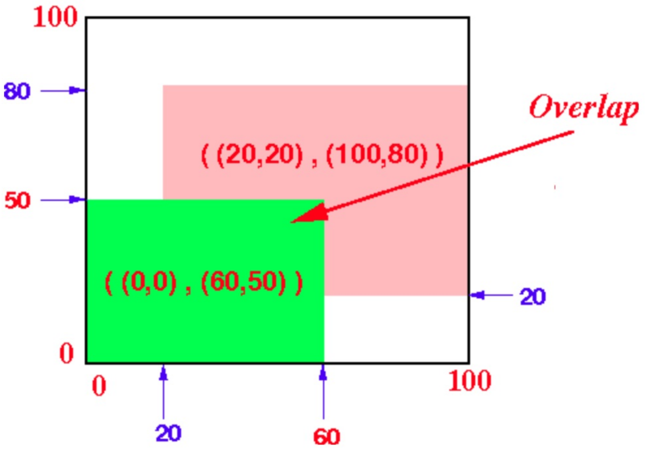
- The `minimum bounding box (mbb)` field for different objects are different

- 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)

```
// n = current node of the search in the R-tree
if ( n == internal node ) then
    for each entry (BB, childptr) in internal node n ) do
        // Look in subtree if (x,y) is inside bounding box
        if (x,y) ∈ BB then
            Lookup((x,y), childptr, result)
        end if
    end for
else
    //n is a leaf node
    for ( each object Ob in node n) do
        if (x,y) ∈ MBB(Ob) then
            Add Ob to result // Object Ob contains point (x,y)
        end if
    end for
end if
```

R-tree- Insert

- Similar to `B-tree`, but more complex
  - Overlap: multiple choices where to add entry
  - Split harder because more choice how to split node (compare `B-tree` = 1 choice)

1) Find potential subtrees for current node
   - Choose one for insert (e.g., the one the would grow the least)
   - continue until leaf is found
2) Insert into leaf
3) Leaf is full? ⇒ split
   - Find best split (minimum overlap between new nodes) is hard ($O(2^M)$)
   - Use linear or quadratic heuristics (original paper: R-trees: a dynamic index structure for spatial searching)
4) Adapt parents if necessary

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
  - $x_1\ x_2\ \ldots\ x_i\ \ldots\ x_n$
  - $x_i = 1$ 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         000010000000
  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        <----- These people
  60       000000000001
  70       000001000000
  85       000000100000

Records: 3, 4, 5 and 11
```

2) Range Match queries using Bitmap indexes

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

```
Bitmap index for age:

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

Or value = 011110000110
```

2) Range Match queries using `Bitmap indexes`

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

```
Bitmap index for salary:

Value        123456789012
-------------------------------
  60         110000000000
  75         001000000000
 100         000100000000
 110         000001000000        100 ≤ salary ≤ 200
 120         000010000000
 140         000000100000
 260         000000010001
 275         000000000010
 350         000000000100
 400         000000001000

Or value   = 000111100000
```

2) Range Match queries using `Bitmap indexes`

- Query: `Find people (buyers of jewelry) where` $45 \le age \le 55$, $100 \le salary \le 200$
- Solution:

```
(45 ≤ age ≤ 50) AND (100 ≤ salary ≤ 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 $|D|$
    - Only $\frac{1}{|D|}$ bits are 1
    - $\Rightarrow$ waste of space
- Solution
  - Compress data
  - Need to make sure that **and** and **or** is still fast

`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