# Systems Infrastructure for Data Science

Web Science Group
Uni Freiburg
WS 2012/13

# Lecture III: Multi-dimensional Indexing

# Querying Multi-dimensional Data

```
SELECT *
  FROM CUSTOMERS
 WHERE ZIPCODE BETWEEN 8000 AND 8999
   AND REVENUE BETWEEN 3500 AND 6000
```

- This example query involves a **range predicate** in **two dimensions**.

- The general case: **spatial queries** over **spatial data**.

# Spatial Data

- Spatial data is used to model multi-dimensional points, lines, rectangles, polygons, cubes, and other geometric objects that exist in space.

- Two main types:
  - **Point** Data
  - **Region** Data

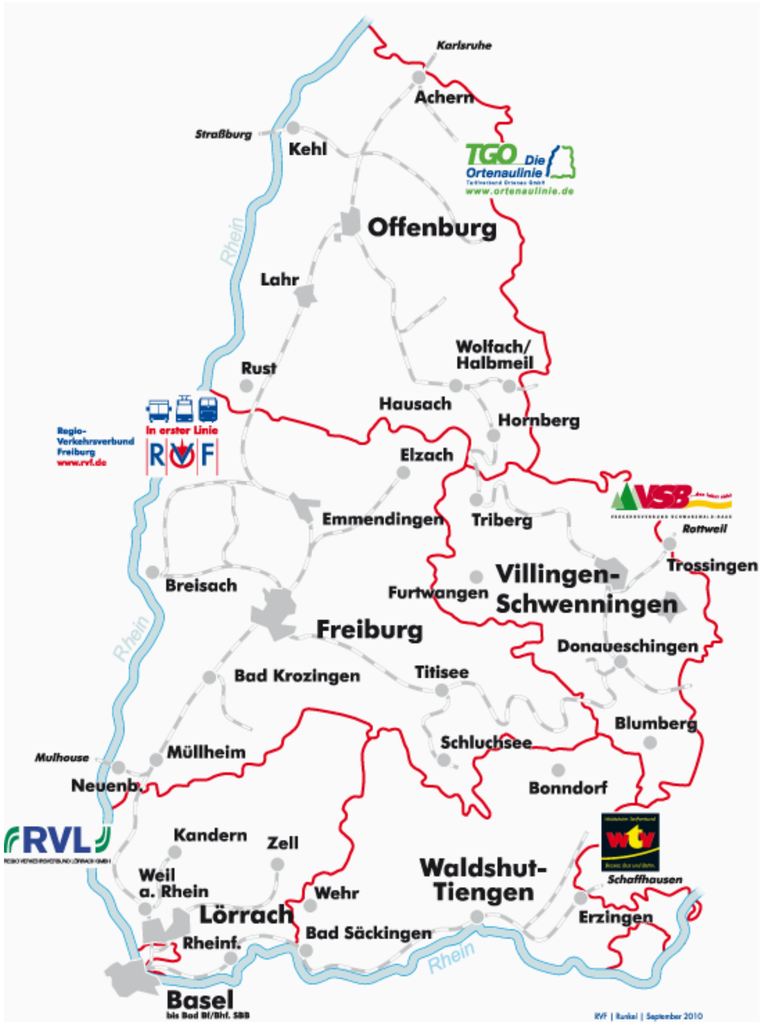
# Point Data

- Points in a multi-dimensional space
- No area or volume
- Examples:
  - **Raster data** such as satellite imagery, where each pixel stores a directly measured value corresponding to a location in space (e.g., temperature, color)
  - **Feature vectors** extracted from images, text, signals such as time series, where the point data is obtained by transforming a data object

# Region Data

- Objects have **spatial extent** (i.e., occupy a certain region of space) characterized by their location and boundary.
- DB typically stores geometric approximations for objects called “**vector data**”, which is constructed using points, line segments, polygons, etc.
- Examples:
  - **Geographic applications** (roads and rivers represented as line segments; countries and lakes represented as polygons)
  - **Computer-Aided Design (CAD) applications** (airplane wing represented as polygons)

# A Familiar Example for Spatial Data with Points, Lines, and Regions



# Spatial Queries

- Spatial queries refer to queries on spatial data.

- Three main types:
  - **Spatial range** queries
  - **Nearest neighbor** queries
  - **Spatial join** queries

# Spatial Range Queries

- A spatial range query has an associated region (i.e., location and boundary).
- The query should return all regions that overlap the specified range or all regions contained within the specified range.
- Examples: relational queries, GIS queries, CAD/CAM queries.
  - Find all employees with salaries between $50K and $60K, and ages between 40 and 50.
  - Find all cities within 100 kilometers of Freiburg.
  - Find all rivers in Baden-Württemberg.

# Nearest Neighbor Queries

- A nearest neighbor query ($k$-NN) returns the $k$ objects that have the smallest distance to a given reference object.
- Results must be ordered by proximity.
- Examples: GIS queries, similarity search in multi-media databases
  – Find the 10 cities nearest to Freiburg.
  – Find the 10 images that are the most similar to this picture of the criminal suspect (*using feature vector point data for images*).

# Spatial Join Queries

- In a spatial join query, the **join condition involves regions and proximity**.
- These queries often times involve **self-join** operations and are expensive to evaluate.
- Example: Consider a relation with **points** representing a city or a mountain.
  - Find pairs of cities within 200 kilometers of each other.
  - Find all cities near a mountain.
- It gets more complex if we represent objects with **region** data instead of point data.

# Spatial Applications Recap

- Traditional relations with *k* fields ~ collections of *k*-dimensional points
- Geographic Information Systems (GIS)
  - Geo-spatial information (2- and 3-dim datasets)
  - **All types** of spatial queries and data are common.
- Computer-Aided Design/Manufacturing (CAD/CAM)
  - Store spatial objects such as surface of airplane wing
  - Both **point and range** data.
  - **Range queries and spatial join queries** are the most common.
- Multi-media Databases
  - Images, audio, video, text, etc. stored and retrieved by content
  - First converted to **feature vector** form (high dimensionality)
  - **Nearest-neighbor queries** (for querying similarity) are the most common.

# Many Solutions for Multi-dimensional Indexing

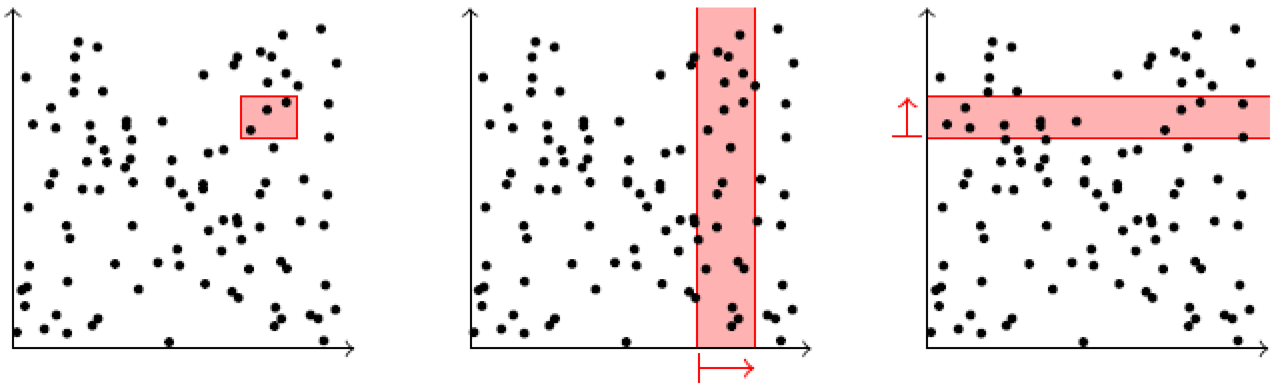
Quad Tree [Finkel 1974]
R-tree [Guttman 1984]
R+-tree [Sellis 1987]
R*-tree [Geckmann 1990]
Vp-tree [Chiueh 1994]
UB-tree [Bayer 1996]
SS-tree [White 1996]
M-tree [Ciaccia 1996]
Pyramid [Berchtold 1998]
DABS-tree [Bohm 1999]
Slim-tree [Faloutsos 2000]
P-Sphere-tree [Goldstein 2000]

K-D-B-Tree [Robinson 1981]
Grid File [Nievergelt 1984]
LSD-tree [Henrich 1989]
hB-tree [Lomet 1990]
TV-tree [Lin 1994]
hB--tree [Evangelidis 1995]
X-tree [Berchtold 1996]
SR-tree [Katayama 1997]
Hybrid-tree [Chakrabarti 1999]
IQ-tree [Bohm 2000]
landmark file [Bohm 2000]
A-tree [Sakurai 2000]

➢ Note that none of these is a “fits all” solution.

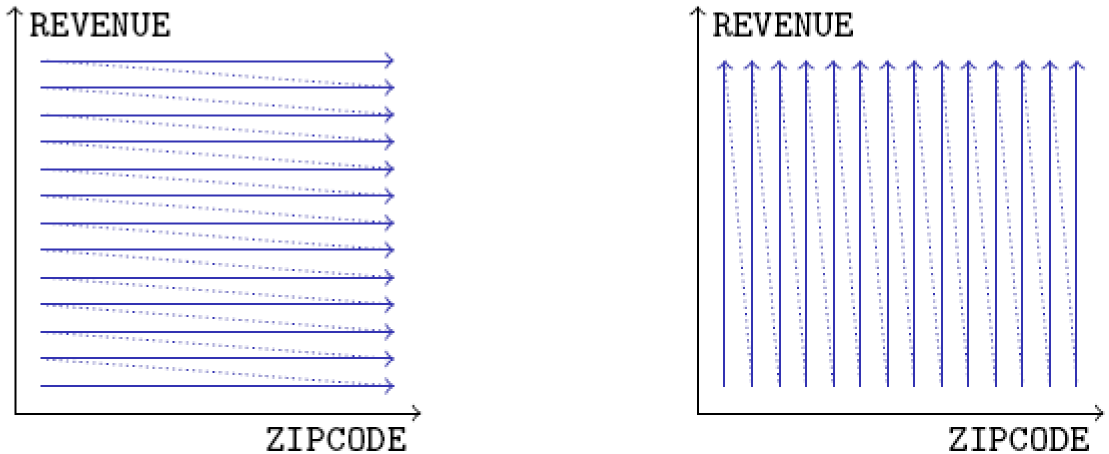
# Can't we just use a B⁺-tree?

- Maybe two B⁺-trees, over ZIPCODE and REVENUE each?



- Can only scan along either index at once, and both of them produce many **false hits**.
- If all you have are these two indexes, you can do **index intersection**:
  – Perform both scans in separation to obtain the *rids* of candidate tuples.
  – Then compute the (**expensive!**) intersection between the two *rid* lists (IBM DB2: IXAND – index AND'ing).

# Maybe with a Composite Key?




⟨REVENUE, ZIPCODE⟩ index ⟨ZIPCODE, REVENUE⟩ index

- Exactly the same thing!
  - Indexes over composite keys are **not symmetric**: **The major attribute dominates** the organization of the B+-tree.
- Again, you can use the index if you really need to. Since the second argument is also stored in the index, you can discard non-qualifying tuples before fetching them from the data pages.

# Single-dimensional Indexes

- $B^+$-trees are fundamentally single-dimensional indexes.
- When we create a **composite search key in $B^+$-tree**, e.g., an index on <*age, sal*>, we effectively **linearize** the 2-dimensional space, since we sort the data entries first by *age* and then by *sal*.
- Consider the following data entries:

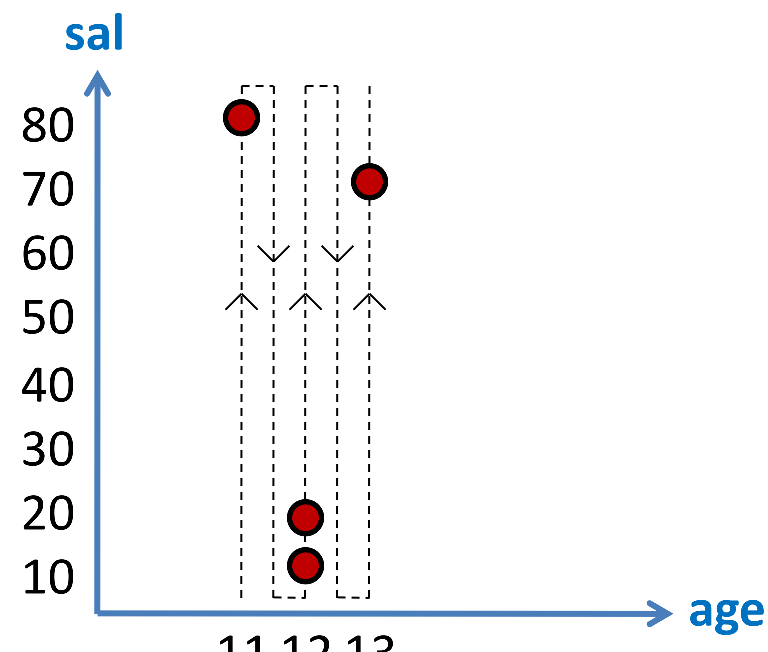
  <11, 80>

  <12, 10>

  <12, 20>

  <13, 70>

linear sort order in $B^+$-tree



# Multi-dimensional Indexes

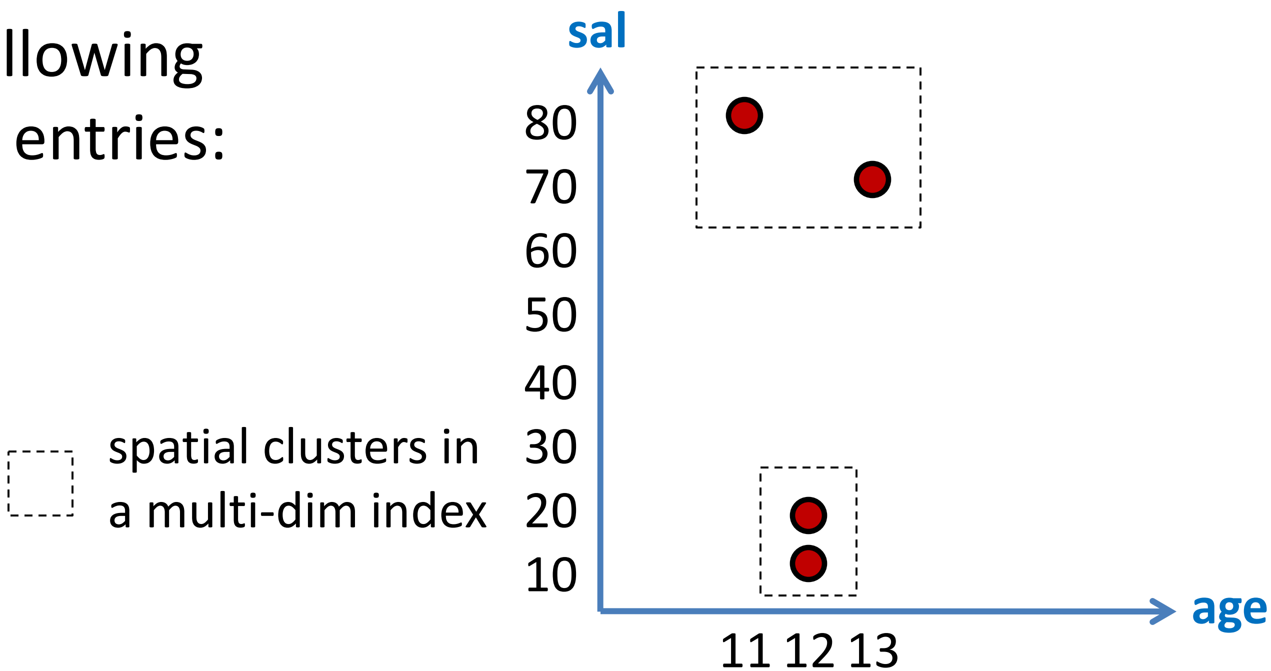
- A multi-dimensional index **clusters** entries so as to exploit "nearness" in multi-dimensional space.
- Keeping track of entries and maintaining a balanced index structure presents a challenge.
- Consider the following *<age, sal>* data entries:

  <11, 80>

  <12, 10>

  <12, 20>

  <13, 70>



# Example Queries ($B^+$-tree vs. Multi-dim)

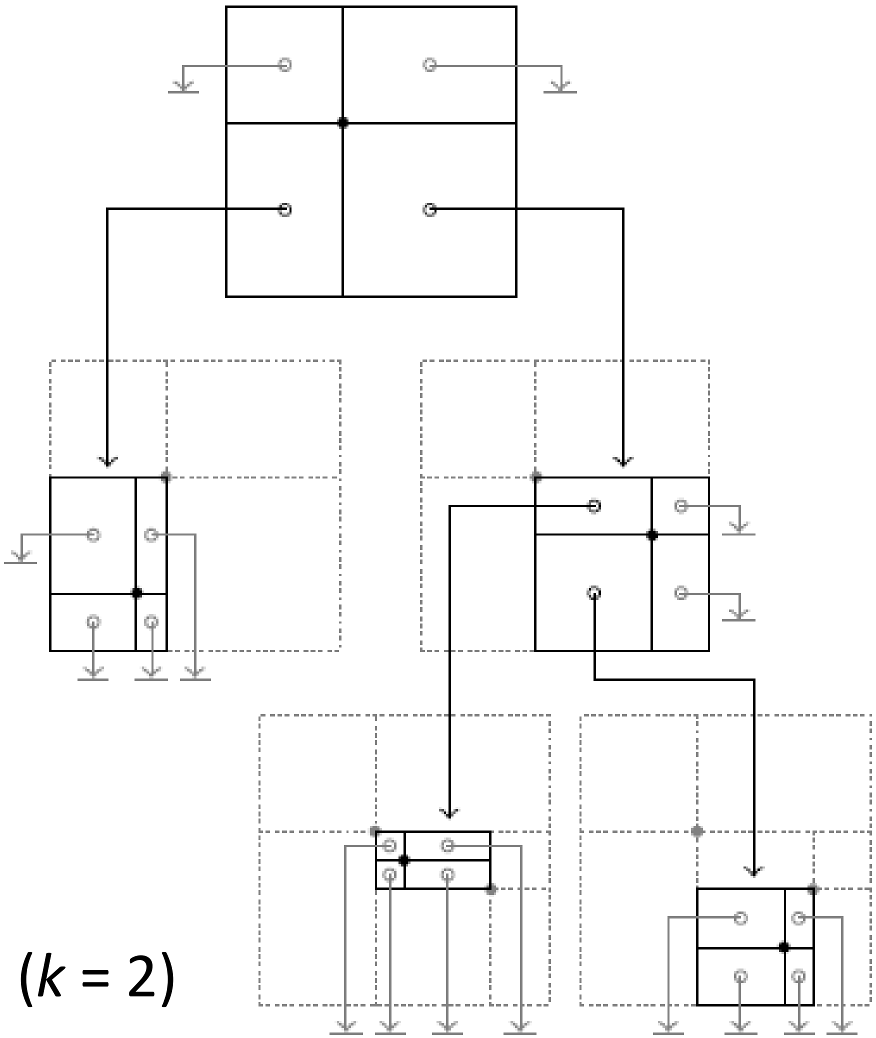
- *age < 12*
  - $B^+$-tree performs better than the multi-dim index.
- *sal < 20*
  - $B^+$-tree can not be used, since *age* is the first field in the search key.
- *age < 12 AND sal < 20*
  - $B^+$-tree effectively utilizes only the index on *age*, and performs badly if most tuples satisfy *age < 12*.

➢ If almost all data entries are to be retrieved in *age* order, then the multi-dim spatial index is likely to be slower than the $B^+$-tree index.

# Multi-dimensional Indexes

- $B^+$-trees can answer one-dimensional queries only.
- We'd like to have a multi-dimensional index structure that
  – is **symmetric** in its dimensions,
  – **clusters** data in a space-aware fashion,
  – is **dynamic** with respect to updates, and
  – provides good support for **useful queries**.
- We'll start with data structures that have been designed for **in-memory** use, then tweak them into **disk-aware** database indexes.

# Point Quad Trees

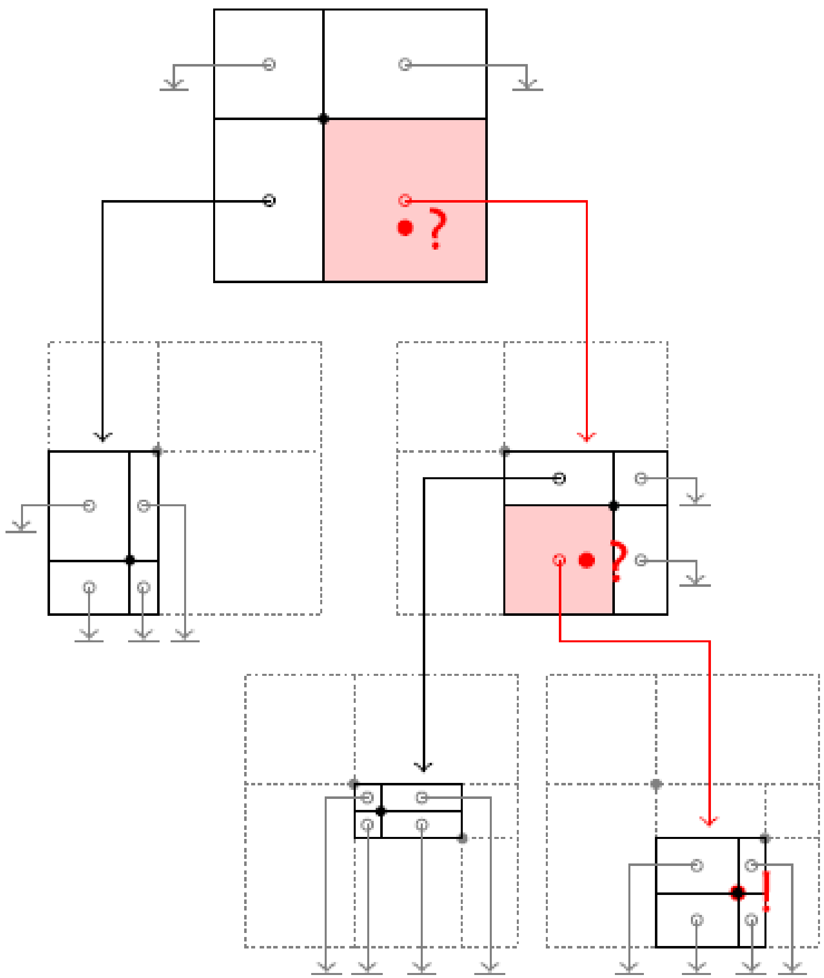


(*k* = 2)

- A binary tree in *k* dimensions
  => $2^k$-ary tree
- Each data point **partitions** the data space into $2^k$ **disjoint regions**.
- In each node, a region points to another node (representing a **refined partitioning** for that region) or to a special **null value**.

➢ Finkel and Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", Acta Informatica, vol. 4, 1974.

# Searching a Point Quad Tree



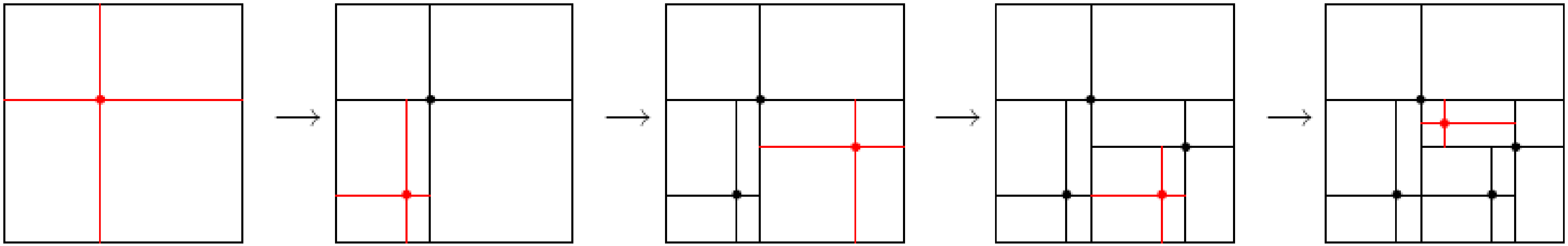
```
Function: p_search (q, node)
if q matches data point in node then
    return data point ;
else
    P ← partition containing q ;
    if P points to null then
        return not found ;
    else
        node′ ← node pointed to by P ;
        return p_search (q, node′) ;
```

---

```
Function: pointsearch (q)
return p_search (q, root) ;
```

# Inserting into a Point Quad Tree

- Inserting a point $q_{new}$ into a quad tree happens analogously to an insertion into a binary tree:
  - Traverse the tree just like during a search for $q_{new}$ until you encounter a partition *P* with a null pointer.
  - Create a new node *n'* that spans the same area as *P* and is partitioned by $q_{new}$, with all partitions pointing to null.
  - Let *P* point to *n'*.
- Note that this procedure does **not** keep the tree **balanced**.



# Evaluating Range Queries with a Point Quad Tree Index

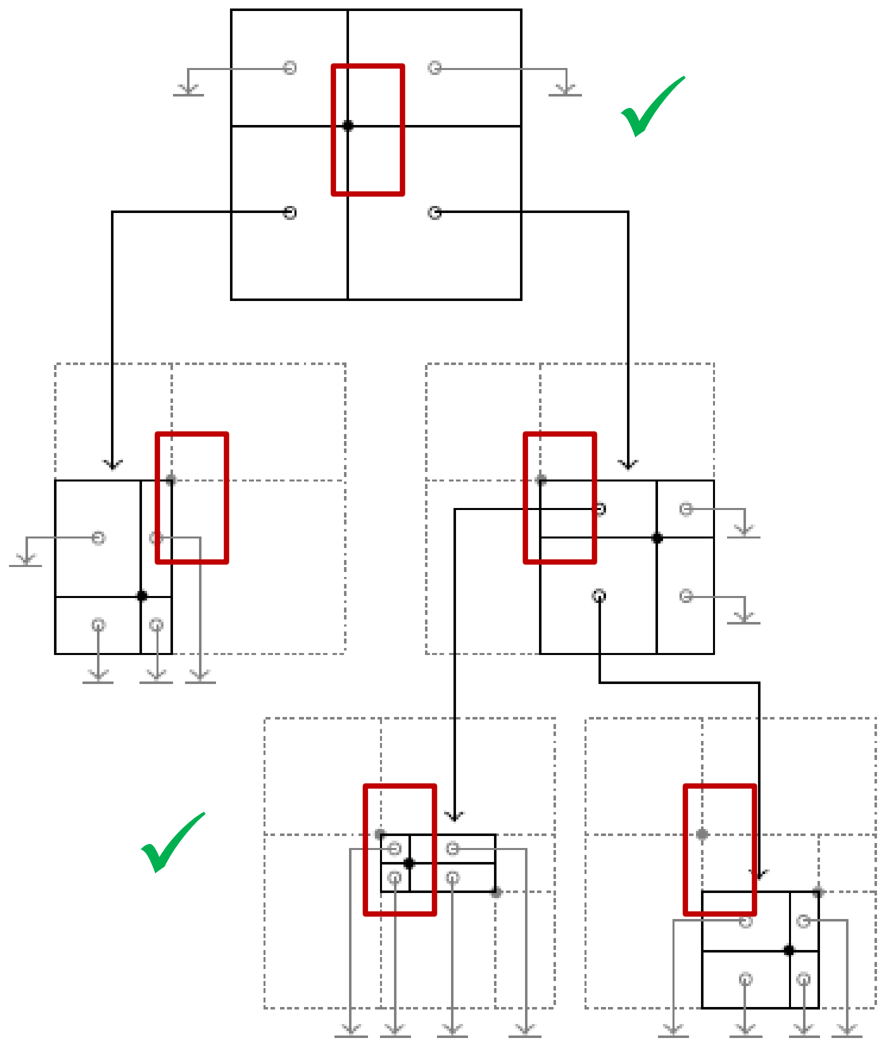
- To evaluate a range query (i.e., rectangular regions), we may need to follow several children of a given quad tree node.

```
Function: r_search (Q, node)
if data point in node is in Q then
    append data point to result ;
foreach partition P in node that intersects with Q do
    node′ ← node pointed to by P ;
    r_search (Q, node′) ;
```

---

```
Function: regionsearch (Q)
return r_search (Q, root) ;
```

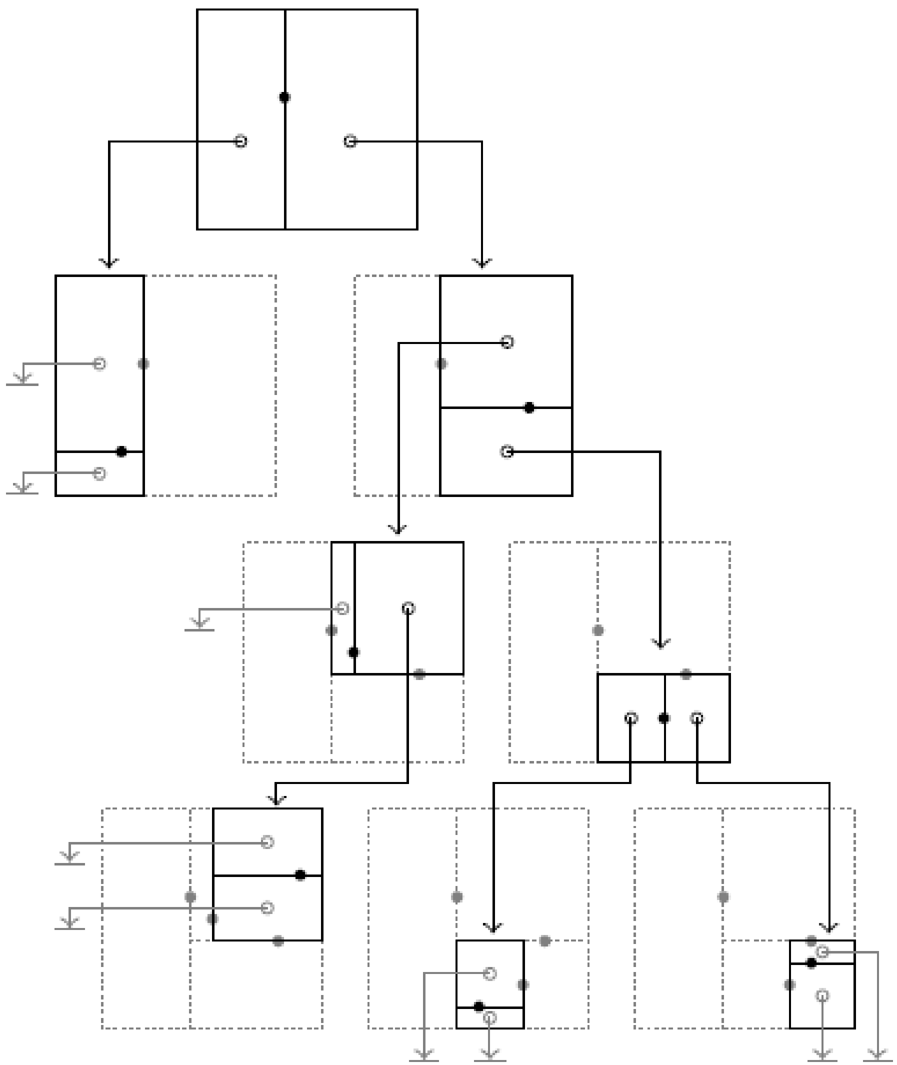
# Range Query Example



# Point Quad Trees

- Point Quad Trees
  - ✓ are **symmetric** with respect to all dimensions
  - ✓ can answer **point queries** and **region queries**
- However,
  - ✘ the shape of a quad tree depends on the **insertion order** of its content, in the worst case **degenerates** into a **linked list**
  - ✘ **null** pointers are **space inefficient** (particularly for large *k*)
  - ✘ they can only store **point data**
- Also, quad trees are designed for main memory.

# k-d Trees



(*k* = 2)

- Index *k*-dimensional data, but keep the tree **binary**.
- For each tree level $l$, use a **different discriminator dimension** $d_l$ along which to partition.
  - Typically: **round robin**

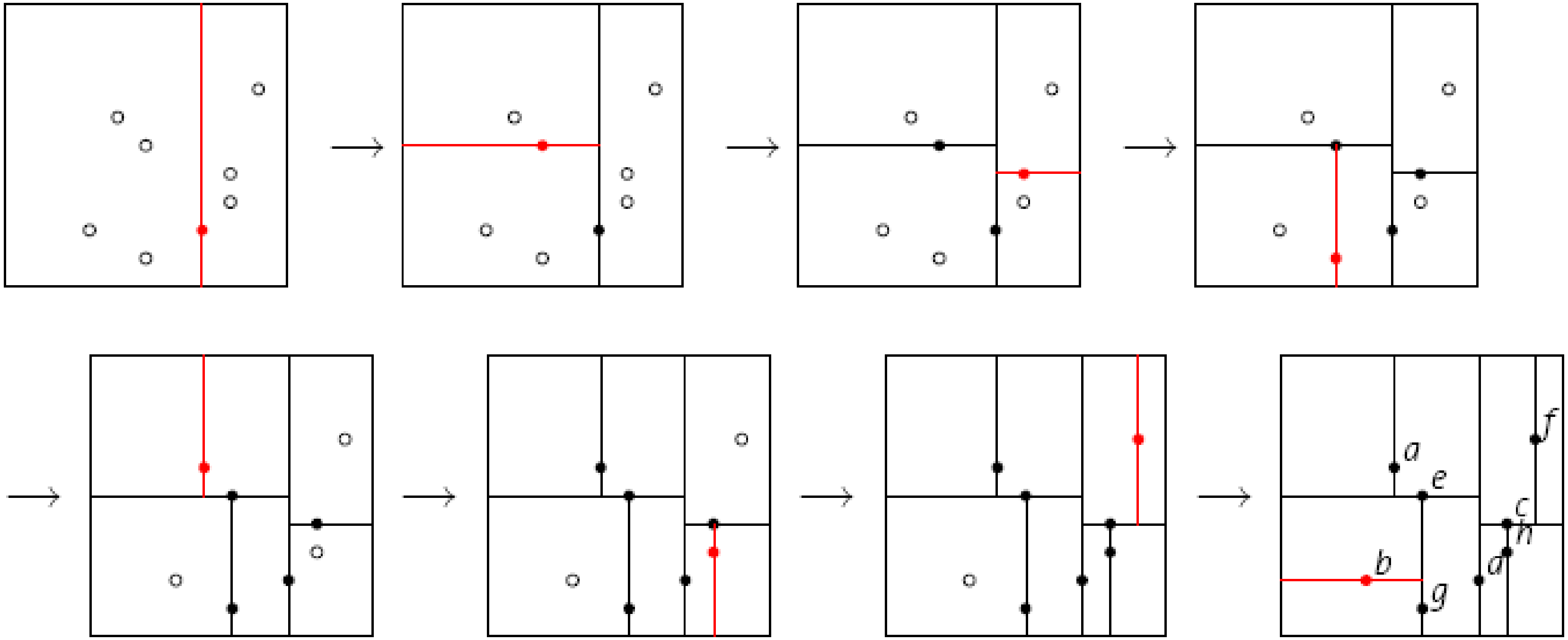
➢ Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", Communications of the ACM, 18:9, 1975.

# k-d Trees

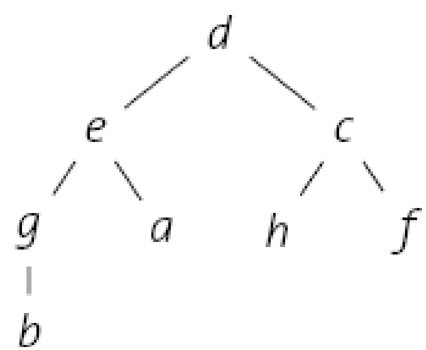
- k-d trees inherit the positive properties of the point quad trees, but improve on **space efficiency**.
- For a given point set, we can also construct a **balanced** k-d tree ($v_i$ denotes coordinate $i$ of point $v$):

**Function:** `kdtree` (*pointset*, *level*)<br>
**if** *pointset* is empty **then**<br>
    **return null** ;<br>
**else**<br>
    $p \leftarrow$ median from *pointset* (along $d_{level}$) ;<br>
    $points_{\text{left}} \leftarrow \{v \in pointset \text{ where } v_{d_{level}} < p_{d_{level}}\}$;<br>
    $points_{\text{right}} \leftarrow \{v \in pointset \text{ where } v_{d_{level}} \geq p_{d_{level}}\}$;<br>
    $n \leftarrow$ new $k$-d tree node, with data point $p$ ;<br>
    $n.left \leftarrow$ `kdtree` ($points_{\text{left}}$, $level + 1$) ;<br>
    $n.right \leftarrow$ `kdtree` ($points_{\text{right}}$, $level + 1$) ;<br>
    **return** $n$ ;

# Balanced k-d Tree Construction




Resulting tree shape:

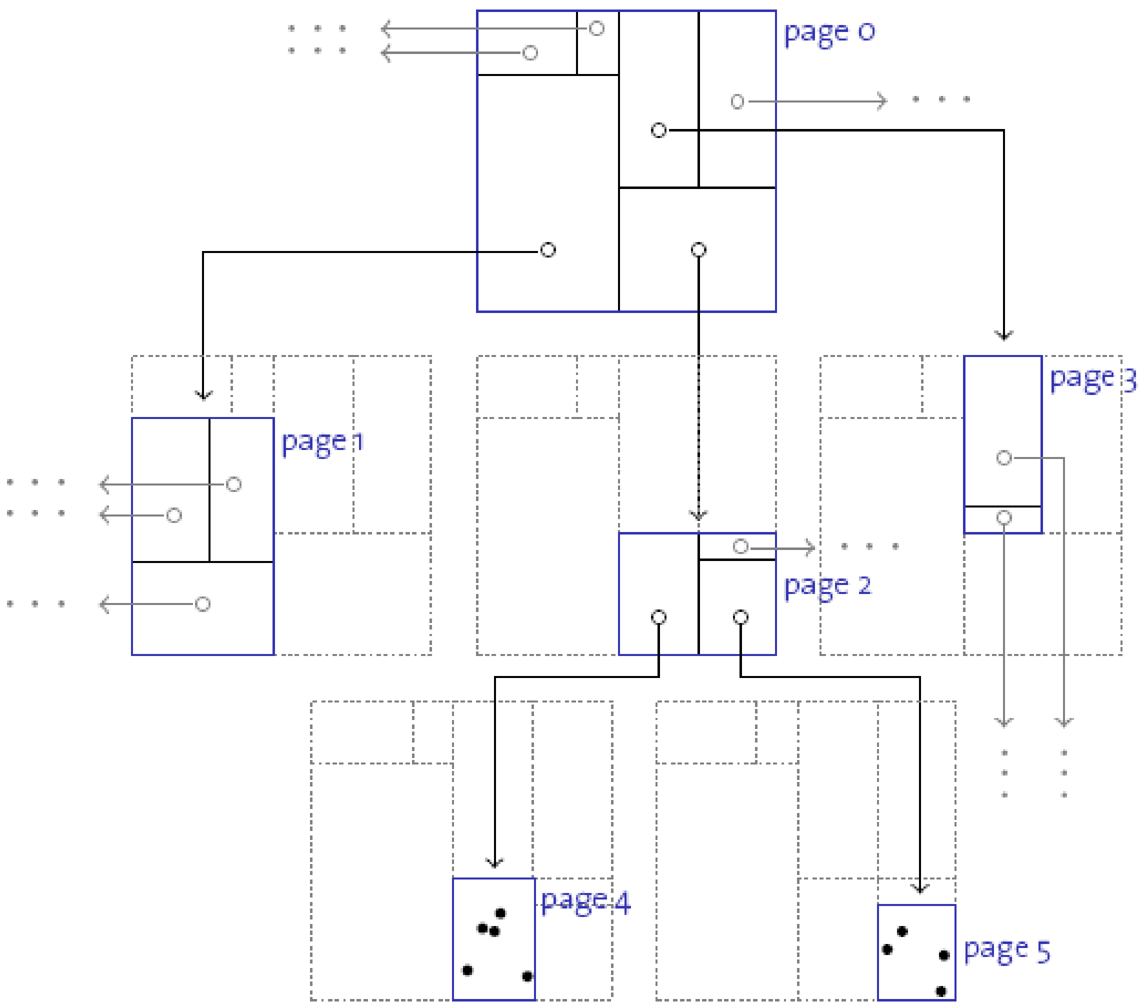


# K-D-B Trees

- k-d trees improve on some of the deficiencies of point quad trees:
  - ✓ We can **balance** a k-d tree by **re-building** it. (For a limited number of points and in-memory processing, this may be sufficient.)
  - ✓ We are no longer wasting big amounts of **space**.
- It’s time to bring k-d trees to the **disk**. The **K-D-B Tree**
  - uses **page** as an organizational unit (e.g., each node in the K-D-B tree fills a page)
  - uses a **k-d tree-like layout** to organize each page

➢ John T. Robinson, “The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes”, SIGMOD’81.

# K-D-B Trees




**region pages:**

- contain entries $\langle region, pageID \rangle$
- no **null** pointers
- form a **balanced** tree
- all regions **disjoint** and **rectangular**

**point pages:**

- contain entries $\langle point, rid \rangle$
- $\rightsquigarrow$ $B^+$-tree leaf nodes

# K-D-B Tree Operations

- **Searching** a K-D-B Tree is straight forward:
  - Within each page determine all regions $R_i$ that contain the query point $q$ (intersect with the query region $Q$).
  - For each of the $R_i$, consult the page it points to and recurse.
  - On point pages, fetch and return the corresponding record for each matching data point $p_i$.

- When **inserting** data, we keep the K-D-B Tree balanced, much like we did in the $B^+$-tree:
  - Simply insert a *<region, pageID>* (*<point, rid>*) entry into a region page (point page) if there is **sufficient space**.
  - Otherwise, **split** the page.

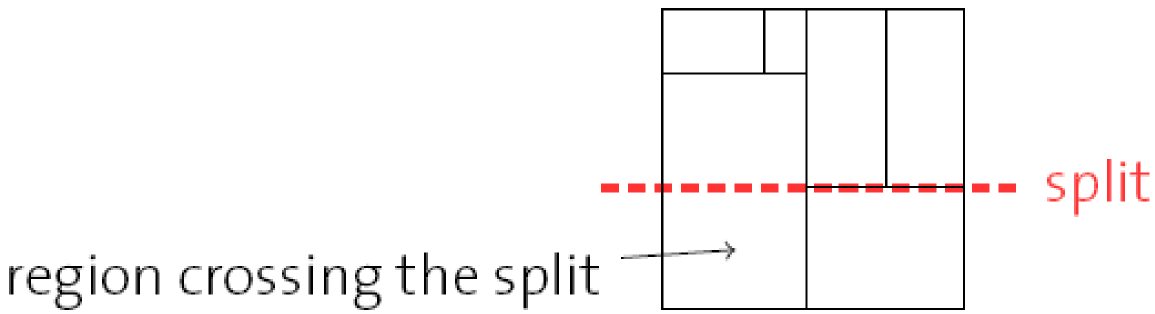
# Splitting a Point Page

- Splitting a point page *p*:
  1. Choose a dimension *i* and an *i*-coordinate $x_i$ along which to split, such that the split will result in two pages ($p_{left}$ and $p_{right}$) that are not overfull.
  2. Move data points with $p_i < x_i$ and $p_i \geq x_i$ to new pages $p_{left}$ and $p_{right}$, respectively.
  3. Replace <*region*, $p_i$> in the parent of *p* with <*left region*, $p_{left}$> <*right region*, $p_{right}$>.

- Step 3 may cause an overflow in *p*'s parent and, hence, lead to a split of a region page.

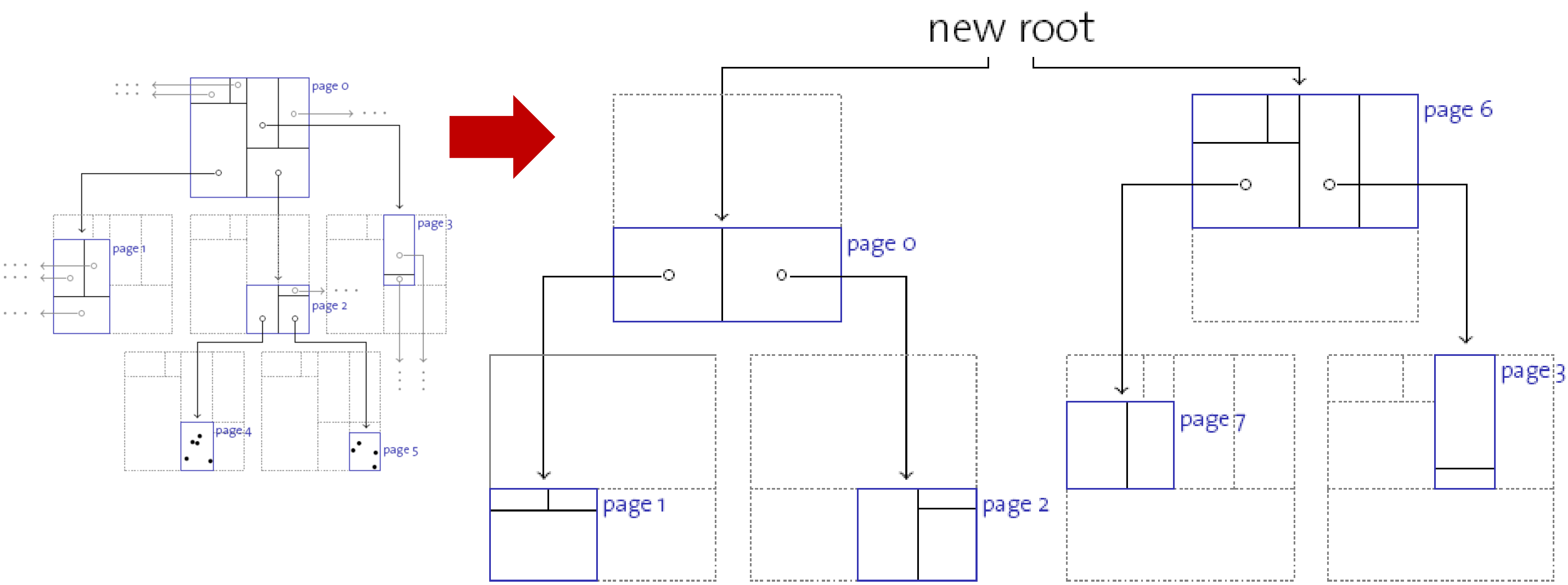
# Splitting a Region Page

- Splitting a **point page** and moving its data **points** to the resulting pages is straight forward.
- In case of a **region page** split, by contrast, some **regions** may intersect with **both** sides of the split.




- Such regions need to be **split**, too.
- This can cause a **recursive splitting downward** in the tree.

# Example Region Page Split




- Region page 1 => pages 1 and 7 (point pages not shown)
- Root page 0 => pages 0 and 6 (creation of new root)

# K-D-B Trees

- K-D-B Trees
  - ✓ are **symmetric** with respect to all dimensions
  - ✓ **cluster** data in a space-aware and page-oriented fashion
  - ✓ are **dynamic** with respect to updates
  - ✓ can answer **point queries** and **region queries**
- However,
  - ✘ we still don’t have support for **region data** and
  - ✘ K-D-B Trees (like k-d trees) won’t handle **deletes** dynamically.
- This is because we always partitioned the data space such that
  - every region is **rectangular**
  - regions never **intersect**

# R-Trees

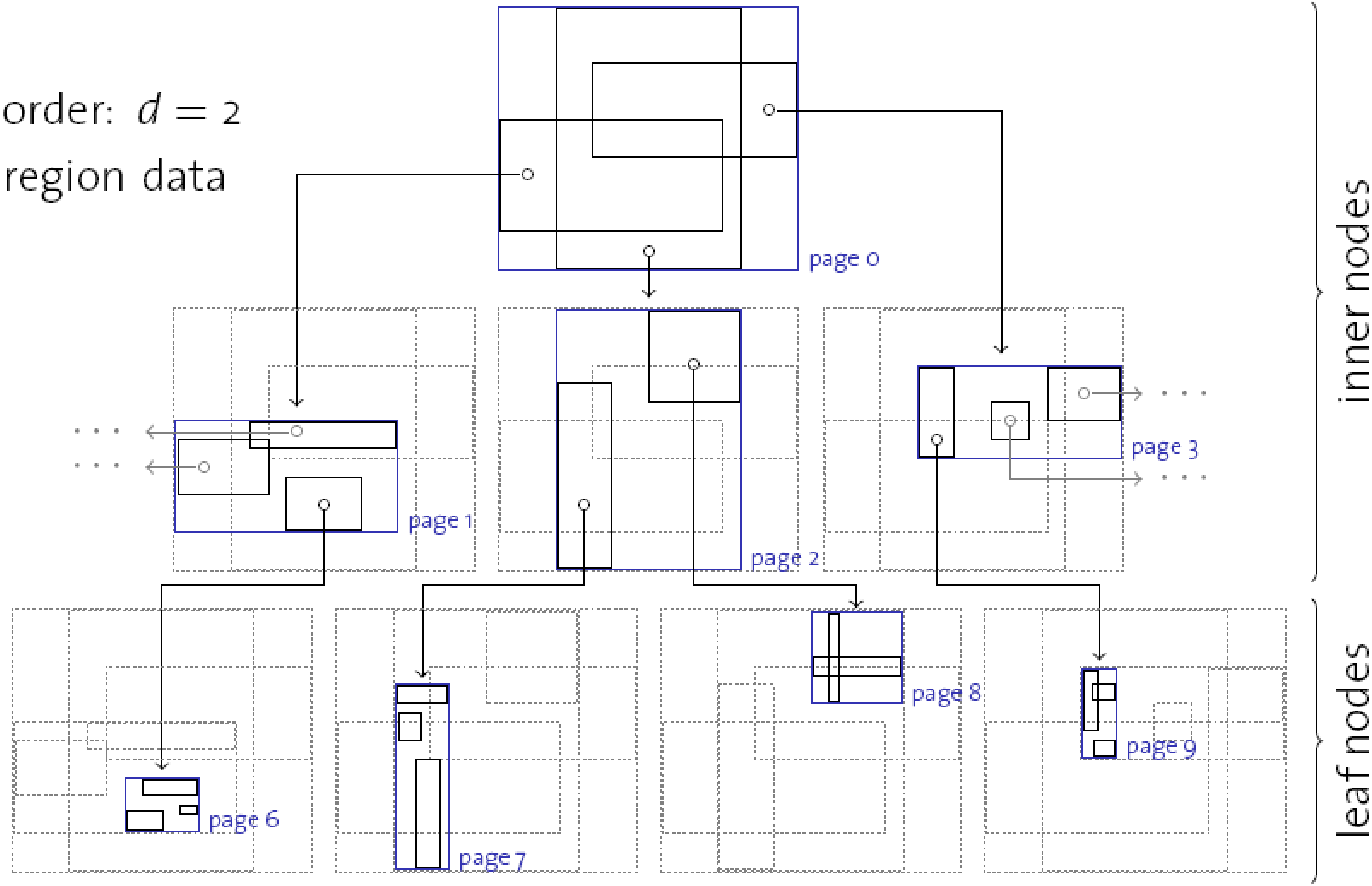
- R-trees do not have the disjointness requirement.
  - R-tree inner or leaf nodes contain *<region, pageID>* and *<region, rid>* entries, respectively. *region* is the **minimum bounding rectangle** that spans all data items reachable by the respective pointer.
  - Every node contains between *d* and *2d* entries except the root node (as in $B^+$-tree).
  - Insertion and deletion algorithms keep an R-tree **balanced at all times**.
- R-trees allow the storage of **point and region data**.

➢ Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", SIGMOD'84.

# R-Tree Example

order: $d = 2$

region data



# Searching an R-Tree

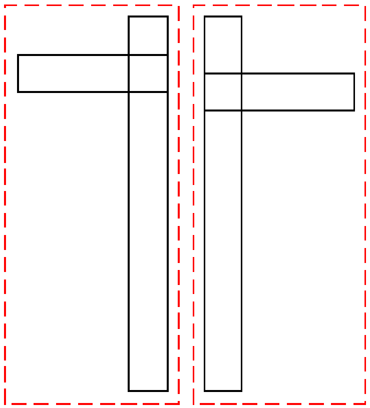
- Start at the root.
  - If current node is non-leaf, for each entry *<E, ptr>,* if region *E* overlaps *Q*, search subtree identified by *ptr*.
  - If current node is leaf, for each entry *<E, rid>,* if *E* overlaps *Q*, *rid* identifies an object that might overlap *Q*.
- While searching an R-tree, we may have to descend into more than one child node for point and region queries (in contrast, a $B^+$-tree equality search goes to just one leaf).

# Inserting into an R-Tree

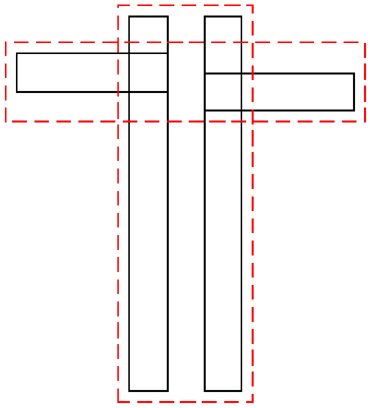
- Inserting into an R-tree very much resembles $B^+$-tree insertion:
  1. Choose a leaf node *n* to insert the new entry.
     - Try to minimize the necessary region enlargement(s).
  2. If *n* is full, split it (resulting in *n* and *n'*) and distribute old and new entries evenly across *n* and *n'*.
     - Splits may propagate bottom-up and eventually reach the root (as in $B^+$-tree).
  3. After the insertion, some regions in the ancestor nodes of *n* may need to be adjusted to cover the new entry.

# Splitting an R-Tree Node

- To split an R-tree node, we have more than one alternative.



bad split



good split

- Heuristic: Minimize the totally covered area.
  - Goal: To reduce the likelihood of both regions being searched on subsequent queries. Redistribute so as to minimize the total area.
  - Exhaustive search for the best split is infeasible. Guttman proposes two ways to approximate the search. Follow-up papers (e.g., the R*-tree paper) aim at improving the quality of node splits.

# Deleting from an R-Tree

- All R-tree invariants are maintained during deletions.
    1. If an R-tree node *n* underflows (i.e., less than *d* entries are left after a deletion), the whole node is deleted.
    2. Then, all entries that existed in *n* are re-inserted into the R-tree, as discussed before.
- Note that Step 1 may lead to a recursive deletion of *n*'s parent.
    - Deletion, therefore, is a rather expensive task in an R-tree.

# R-Tree Variants

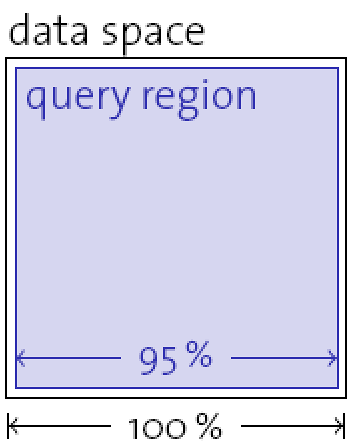
- The **R*-tree** uses the concept of **forced reinserts** to reduce overlap in tree nodes. When a node overflows, instead of splitting:
  - Remove some (say, 30% of the) entries and reinsert them into the tree.
  - Could result in all reinserted entries fitting on some existing pages, avoiding a split.
- R*-trees also use a different heuristic, minimizing **box perimeters** rather than box areas during insertion.
- Another variant, the **$R^+$-tree**, avoids overlap by inserting an object into **multiple leaves** if necessary.
  - Searches now take a single path to a leaf, at cost of redundancy.

# Indexing High-dimensional Data

- Typically, high-dimensional datasets are collections of points, not regions.
  - Example: Feature vectors in multi-media applications
  - Very sparse
- Nearest neighbor queries are common.
  - R-tree becomes worse than sequential scan for most datasets with more than a dozen dimensions.
- As dimensionality increases, contrast (i.e., the ratio of distances between nearest and farthest points) usually decreases; “nearest neighbor” is not meaningful.
  - In any given data set, it is advisable to empirically test contrast.

# High Dimensional Spaces

- For large *k*, all the techniques we discussed become ineffective:
  - Example: for $k = 100$, we'd get $2^{100} \sim 10^{30}$ partitions per node in a point quad tree. Even with billions of data points, almost all of these are empty.
  - Consider a really big search region, cube-sized covering 95% of the range along each dimension:

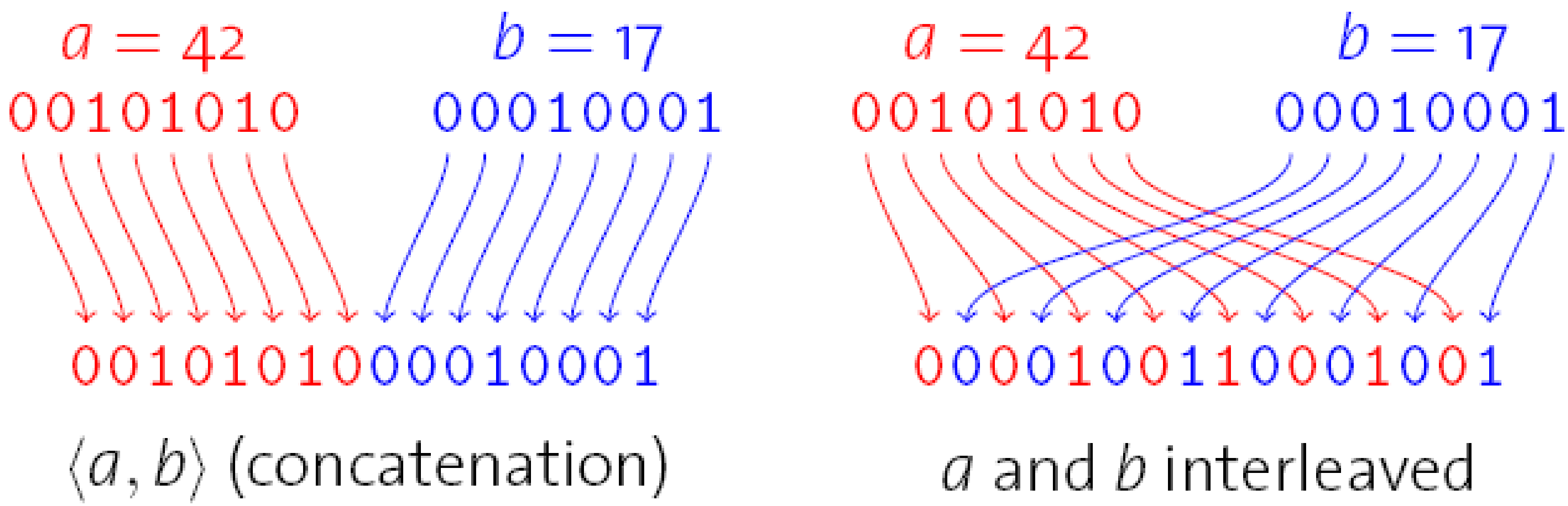



For $k = 100$, the probability of a point being in this region is still only $0.95^{100} \approx 0.59\,\%$.

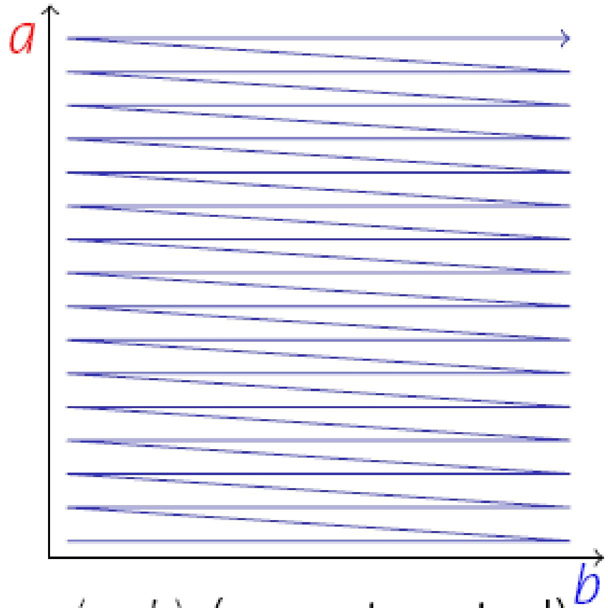
  - We experience the **curse of dimensionality** here.

# Bit Interleaving

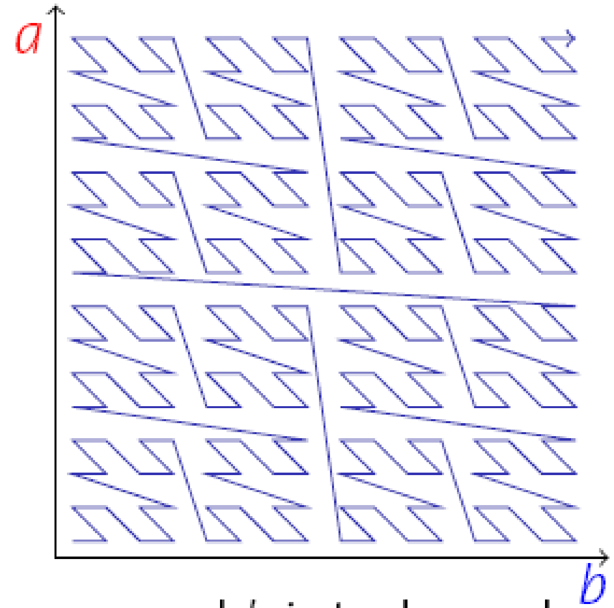
- We saw earlier that a B+-tree over concatenated fields <*a*, *b*> doesn't help our case, because of the asymmetry between the role of *a* and *b* in the index.
- What happens if we interleave the bits of *a* and *b* (hence, make the B+-tree "more symmetric")?



# Z-Ordering




$\langle a, b \rangle$ (concatenated)

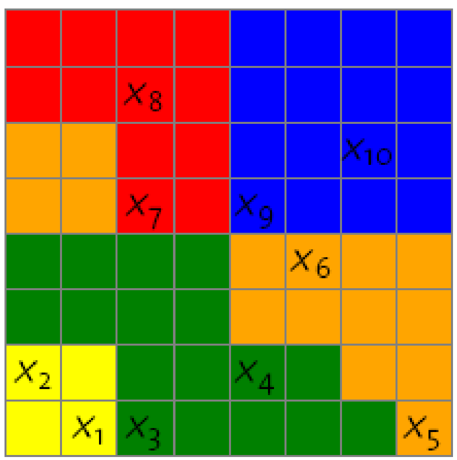


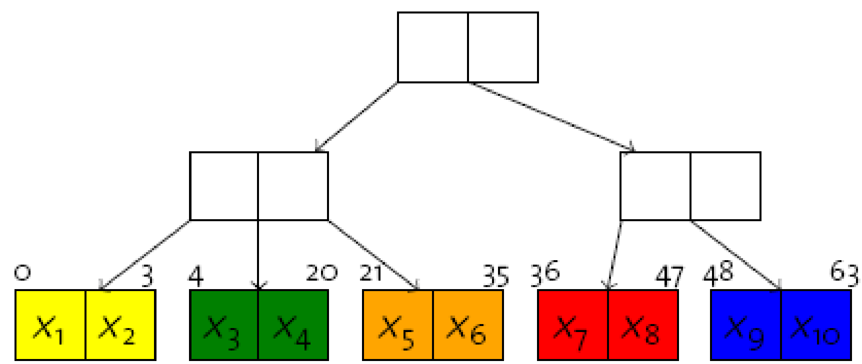

$a$ and $b$ interleaved

- Both approaches **linearize** all coordinates in the value space according to some order.
- Bit interleaving leads to what is called the **Z-order**.
- The Z-order (largely) preserves **spatial clustering**.

# B$^+$-trees over Z-Codes

- Use a **B$^+$-tree** to index Z-codes of multi-dimensional data.
- Each leaf in the B$^+$-tree describes an **interval** in the **Z-space**.
- Each interval in the Z-space describes a **region** in the multi-dimensional data space.




- To retrieve all data points in a query region *Q*, try to touch only those leaf pages that **intersect** with *Q*.

# Summary

- **Point Quad Tree**
  - k-dimensional analogy to binary trees; main memory only.
- **k-d Tree, K-D-B Tree**
  - k-d tree: Partition space one dimension at a time (round-robin).
  - K-D-B Tree: $B^+$-tree-like organization with pages as nodes; nodes use a k-d-like structure internally.
- **R-Tree**
  - Regions within a node may overlap; fully dynamic; for point and region data.
- **Curse Of Dimensionality**
  - Most indexing structures become ineffective for large k; fall back to sequential scanning and approximation/compression.