

Architecture and Implementation of Database Systems (Winter 2016/17)

Jens Teubner, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

Winter 2016/17

Part IV

Multi-Dimensional Indexing

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

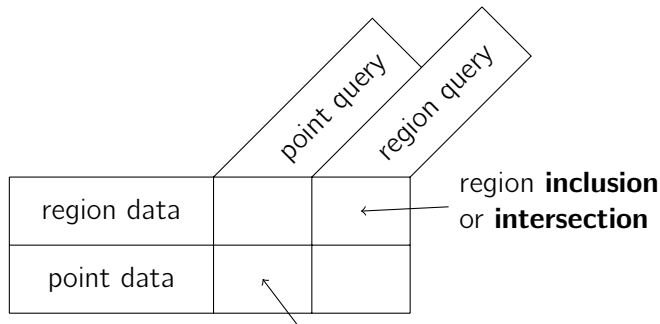
This query involves a **range predicate** in **two** dimensions.

Typical use cases with **multi-dimensional data** include

- **geographical data** (GIS: Geographical Information Systems),
- **multimedia retrieval** (*e.g.*, image or video search),
- **OLAP** (Online Analytical Processing).

... More Challenges ...

Queries **and** data can be **points** or **regions**.

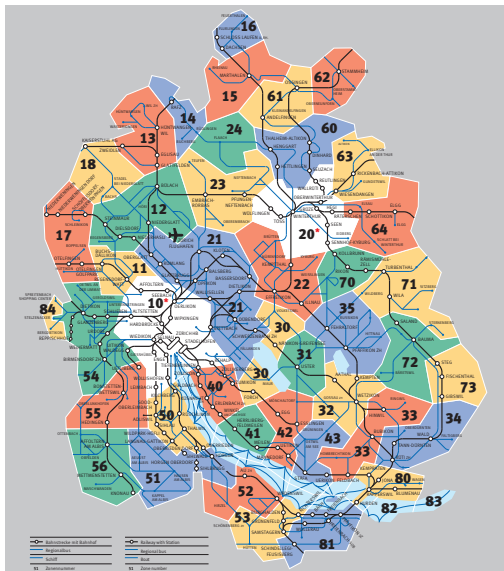


most interesting: ***k*-nearest-neighbor search** (*k*-NN)

... and you can come up with many more meaningful types of queries over multi-dimensional data.

Note: All equality searches can be reduced to one-dimensional queries.

Points, Lines, and Regions



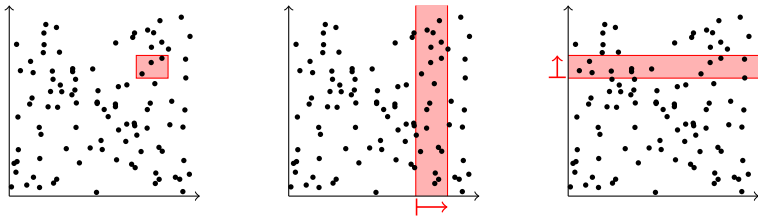
... More Solutions

Quad Tree [Finkel 1974]	K-D-B-Tree [Robinson 1981]
R-tree [Guttman 1984]	Grid File [Nievergelt 1984]
R ⁺ -tree [Sellis 1987]	LSD-tree [Henrich 1989]
R*-tree [Geckmann 1990]	hB-tree [Lomet 1990]
Vp-tree [Chiueh 1994]	TV-tree [Lin 1994]
UB-tree [Bayer 1996]	hB- Π -tree [Evangelidis 1995]
SS-tree [White 1996]	X-tree [Berchtold 1996]
M-tree [Ciaccia 1996]	SR-tree [Katayama 1997]
Pyramid [Berchtold 1998]	Hybrid-tree [Chakrabarti 1999]
DABS-tree [Böhm 1999]	IQ-tree [Böhm 2000]
Slim-tree [Faloutsos 2000]	landmark file [Böhm 2000]
P-Sphere-tree [Goldstein 2000]	A-tree [Sakurai 2000]

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

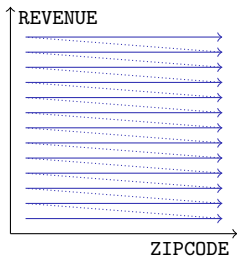
Can't We Just Use a B⁺-tree?

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

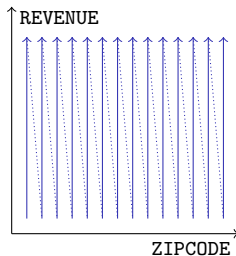


- Can only scan along **either** index at once, and both of them produce many **false hits**.
- If all you have are these two indices, you can do **index intersection**: perform both scans in separation to obtain the rids of candidate tuples. Then compute the (expensive!) intersection between the two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

Hmm, ... Maybe With a Composite Key?



$\langle \text{REVENUE}, \text{ZIPCODE} \rangle$ index



$\langle \text{ZIPCODE}, \text{REVENUE} \rangle$ index

■ Exactly the same thing!

Indices 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.

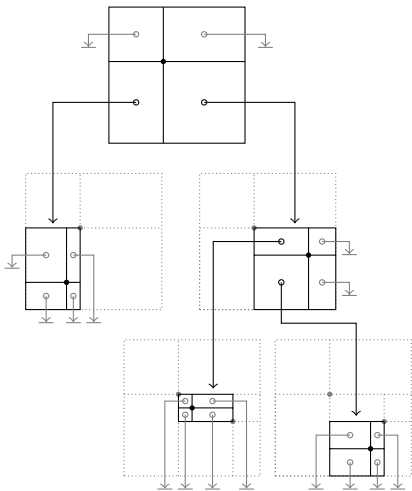
Multi-Dimensional Indices

- B⁺-trees can answer **one-dimensional** queries **only**.⁷
- 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 indices.

⁷Toward the end of this chapter, we'll see UB-trees, a nifty trick that uses B⁺-trees to support some multi-dimensional queries.

“Binary” Search Tree

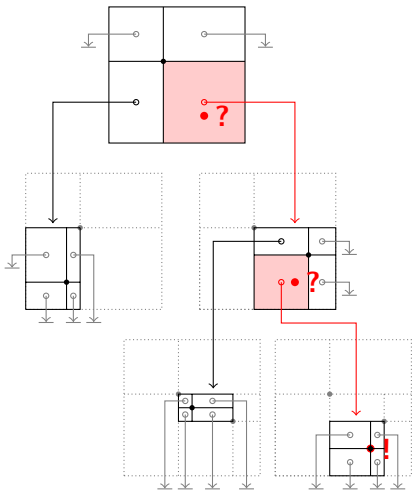
In k dimensions, a “**binary tree**” becomes a 2^k -**ary tree**.



- 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) or to a special **null** value.
- This data structure is a **point quad tree**.

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

Searching a Point Quad Tree



```
1 Function: p_search ( $q$ ,  $node$ )
2 if  $q$  matches data point in  $node$  then
3   | return data point ;
4 else
5   |  $P \leftarrow$  partition containing  $q$  ;
6   | if  $P$  points to null then
7     | return not found ;
8   | else
9     |  $node' \leftarrow$  node pointed to by  $P$  ;
10    | return p_search ( $q$ ,  $node'$ ) ;
```

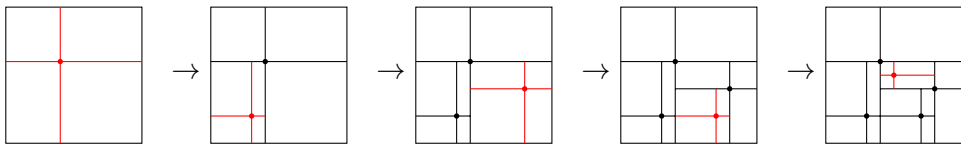
```
1 Function: pointsearch ( $q$ )
2 return p_search ( $q$ ,  $root$ ) ;
```

Inserting into a Point Quad Tree

Inserting a point q_{new} into a quad tree happens analogously to an insertion into a binary tree:

- 1 **Traverse** the tree just like during a search for q_{new} until you encounter a partition P with a **null** pointer.
- 2 Create a **new node** n' that spans the same area as P and is partitioned by q_{new} , with all partitions pointing to **null**.
- 3 Let P point to n' .

Note that this procedure does **not** keep the tree **balanced**.



Range Queries

To evaluate a **range query**⁸, we may need to follow **several** children of a quad tree node *node*:

```
1 Function: r_search (Q, node)
2 if data point in node is in Q then
3   | append data point to result ;
4 foreach partition P in node that intersects with Q do
5   | node' ← node pointed to by P ;
6   | r_search (Q, node') ;
```

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

⁸We consider **rectangular** regions only; other shapes may be answered by querying for the **bounding rectangle** and post-processing the output.

Point Quad Trees—Discussion

Point quad trees

- ✓ are **symmetric** with respect to all dimensions and
- ✓ can answer **point queries** and **region queries**.

But

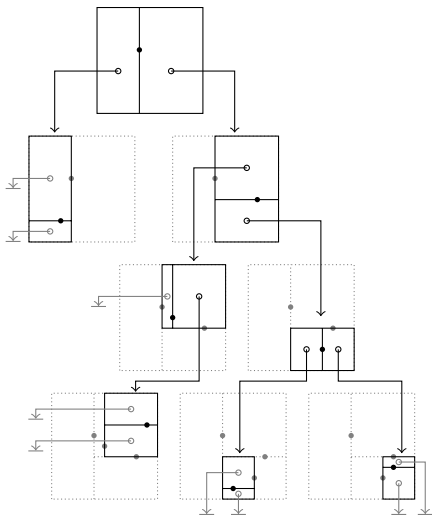
- ✗ the shape of a quad tree depends on the **insertion order** of its content, in the worst case **degenerates** into a **linked list**,
- ✗ **null** pointers are **space inefficient** (particularly for large k).

In addition,

- ☹ they can only store **point data**.

Also remember that quad trees are designed for **main memory**.

k -d Trees



- 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**

- This is a **k -d tree**.

↗ Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Comm. ACM*, vol. 18, no. 9, Sept. 1975.

k -d Trees

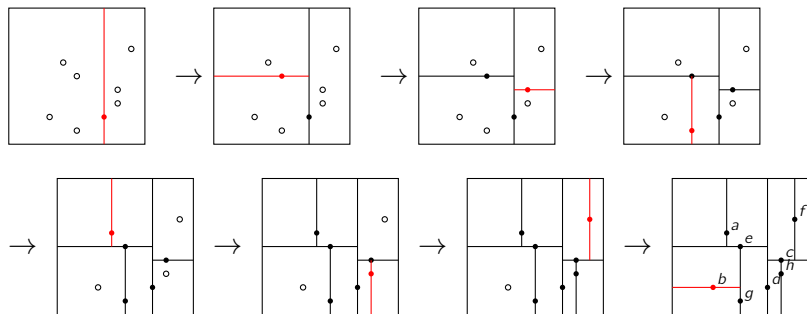
k -d trees inherit the positive properties of the point quad trees, but improve on **space efficiency**.

For a given point set, we can also construct a **balanced** k -d tree:⁹

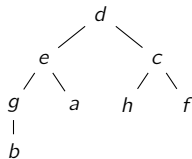
```
1 Function: kdtree (pointset, level)
2 if pointset is empty then
3   return null ;
4 else
5    $p \leftarrow$  median from pointset (along  $d_{level}$ ) ;
6    $points_{left} \leftarrow \{v \in pointset \text{ where } v_{d_{level}} < p_{d_{level}}\}$ ;
7    $points_{right} \leftarrow \{v \in pointset \text{ where } v_{d_{level}} \geq p_{d_{level}}\}$ ;
8    $n \leftarrow$  new  $k$ -d tree node, with data point  $p$  ;
9    $n.left \leftarrow kdtree (points_{left}, level + 1)$  ;
10   $n.right \leftarrow kdtree (points_{right}, level + 1)$  ;
11  return  $n$  ;
```

⁹ v_i : coordinate i of point v .

Balanced k -d Tree Construction



Resulting tree shape:



k -d trees improve on some of the deficiencies of point quad trees:

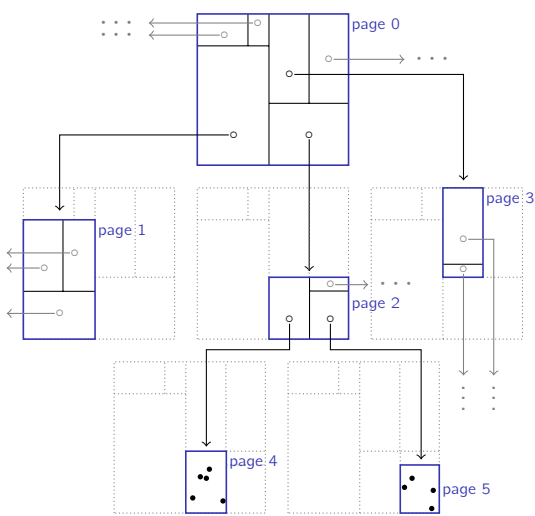
- ✓ We can **balance** a k -d tree by **re-building** it.
(For a limited number of points and in-memory processing, this may be sufficient.)
- ✓ We're no longer wasting big amounts of **space**.
- ✗ k -d trees are not really symmetric with respect to space dimensions.

It's time to bring k -d trees to the disk. The **K-D-B-Tree**

- uses **pages** as an organizational unit
(e.g., each node in the K-D-B-tree fills a page) and
- 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 1981*.

K-D-B-Tree Idea



region pages:

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

point pages:

- contain entries $\langle \text{point}, \text{rid} \rangle$
- \leadsto B⁺-tree leaf nodes

- **Searching** a K-D-B-Tree works straightforwardly:
 - 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 $\langle region, pageID \rangle$ ($\langle point, rid \rangle$) entry into a region page (point page) if there's **sufficient space**.
 - **Otherwise, split** the page.

Splitting a Point Page

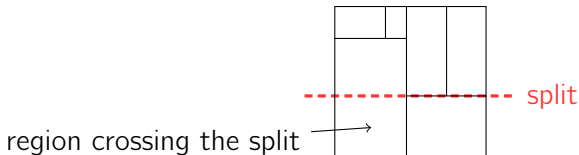
Splitting a point page p :

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

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

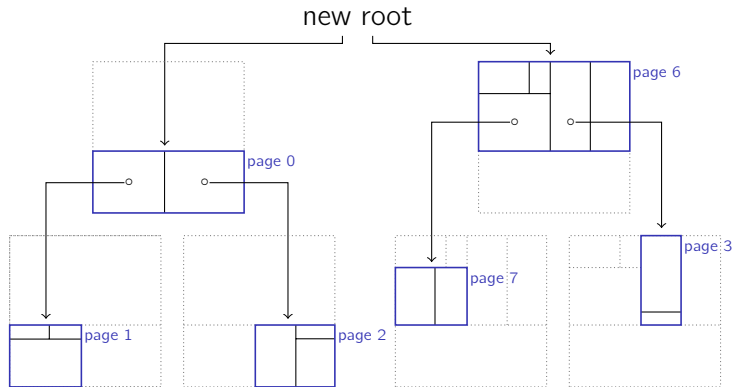
Splitting a Region Page

- Splitting a **point page** and moving its data **points** to the resulting pages is straightforward.
- In case of a **region page split**, by contrast, some **regions** may intersect with **both** sides of the split (e.g., page 0 on slide 131).



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

Example: Page 0 Split in Tree on Slide 131



- Root page 0 → pages 0 and 6 (≈ creation of new root).
- Region page 1 → pages 1 and 7 (point pages not shown).

K-D-B-Trees

- ✓ are **symmetric** with respect to all dimensions,¹⁰
- ✓ **cluster** data in a space-aware and page-oriented fashion,
- ✓ are **dynamic** with respect to updates, and
- ✓ 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** and
- regions never **intersect**.

¹⁰However, split dimensions must be chosen, which re-introduces asymmetry.

R-trees do not have the disjointness requirement.

- R-tree inner or leaf nodes contain $\langle region, pageID \rangle$ or $\langle region, rid \rangle$ 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 ($\leadsto B^+$ -tree).¹¹
- **Insertion** and **deletion** algorithms keep an R-tree **balanced** at all times.

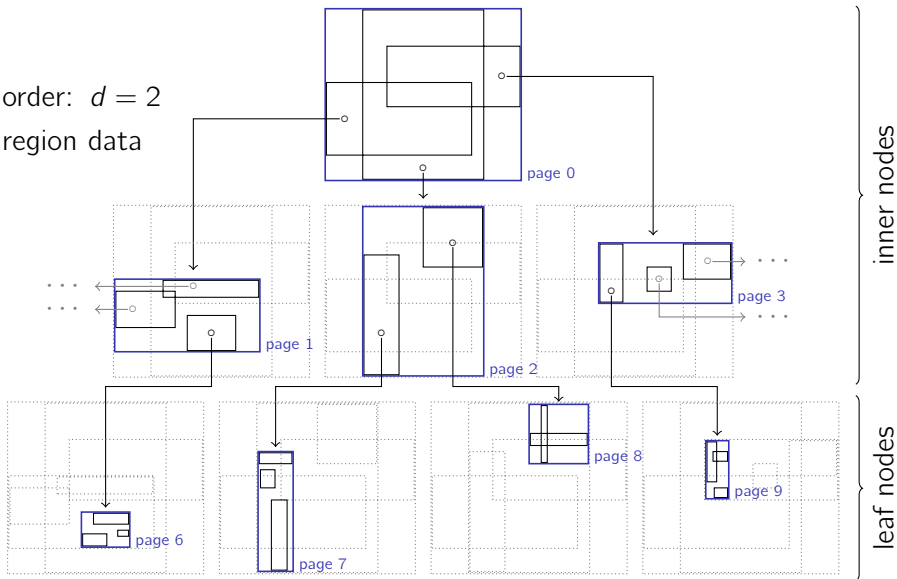
R-trees allow the storage of **point and region data**.

↗ Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD 1984*.

¹¹except the root node

R-Tree: Example

- order: $d = 2$
- region data



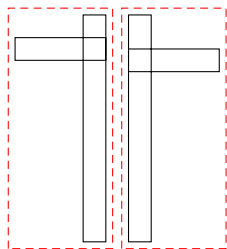
While **searching** an R-tree, we may have to descend into more than one child node for point **and** region queries (\nearrow range search in point quad trees, slide 125).

Inserting into an R-tree very much resembles B⁺-tree insertion:

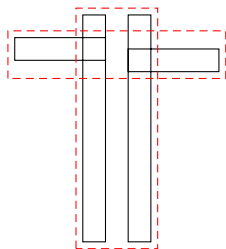
- 1 **Choose** a leaf node n to insert the new entry.
 - Try to minimize the necessary region enlargement(s).
- 2 If n is **full**, **split** it (resulting in n and n') and distribute old and new entries evenly across n and n' .
 - Splits may propagate bottom-up and eventually reach the root (\nearrow 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*

*according
to Guttman

Heuristic: Minimize the totally covered area.

- **Exhaustive** search for the best split infeasible.
- Guttman proposes two ways to **approximate** the search.
- Follow-up papers (e.g., the R*-tree) aim at improving the quality of node splits.

All R-tree invariants (slide 137) 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.

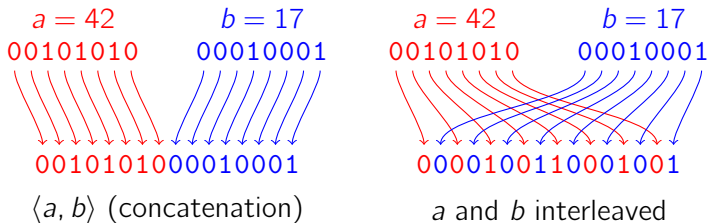
- Deletion, therefore, is a rather **expensive** task in an R-tree.

Spacial indexing in mainstream database implementations.

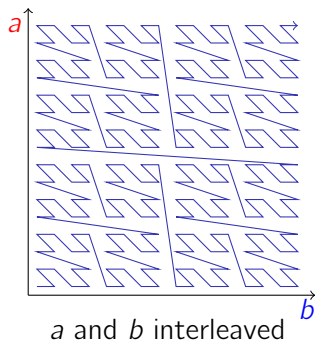
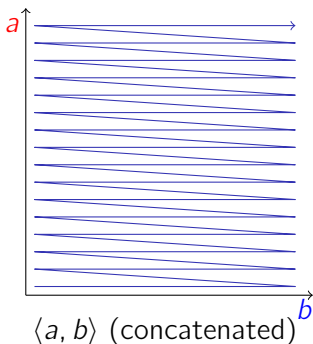
- Indexing in commodity systems is typically based on **R-trees**.
- Yet, only few systems implement them out of the box (*e.g.*, PostgreSQL).

Bit Interleaving

- We saw earlier that a B⁺-tree over **concatenated** fields $\langle a, b \rangle$ 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")?



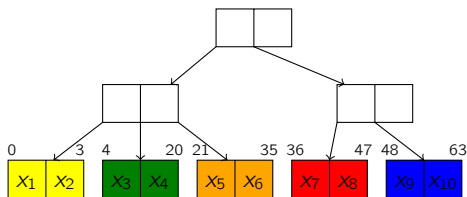
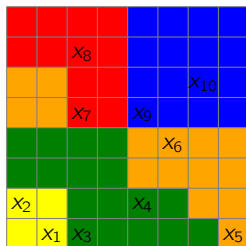
Z-Ordering



- Both approaches **linearize** all coordinates in the value space according to some **order**. ↗ see also slide 120
- Bit interleaving leads to what is called the **Z-order**.
- The Z-order (largely) preserves spacial **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.

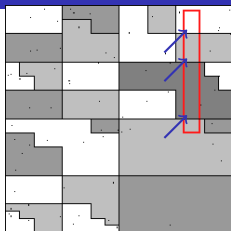


- To retrieve all data points in a query region Q , try to touch only those leaf pages that **intersect** with Q .

UB-Tree Range Queries

After each page processed, perform an **index re-scan** (↗) to fetch the next leaf page that intersects with Q .

```
1 Function: ub_range ( $Q$ )
2  $cur \leftarrow z(Q_{\text{bottom, left}})$ ;
3 while true do
4     // search  $B^+$ -tree page containing  $cur$  (↗ slide 70)
5      $page \leftarrow \text{search}(cur)$ ;
6     foreach data point  $p$  on  $page$  do
7         if  $p$  is in  $Q$  then
8             append  $p$  to result;
9     if region in  $page$  reaches beyond  $Q_{\text{top, right}}$  then
10        break;
11    // compute next Z-address using  $Q$  and data on current page
12     $cur \leftarrow \text{get\_next\_z\_address}(Q, page)$ ;
```



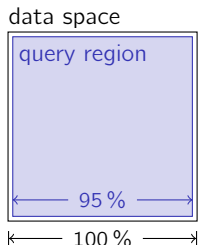
Example by Volker Markl and Rudolf Bayer, taken from <http://www.inf.uni-leipzig.de/~markl/papers/ubtree.pdf>

- The cost of a region query is **linear** in the **size of the result** Q and **logarithmic** with respect to the stored data volume N ($\frac{4 \cdot Q}{2d} \cdot \mathcal{O}(\log_d N)$).
- UB-trees are **fully dynamic**, a property inherited from the underlying B^+ -trees.
- The use of other **space-filling curves** to linearize the data space is discussed in the literature, too. *E.g.*, **Hilbert curves**.
- 🏠 UB-trees have been commercialized in the Transbase® database system.

Spaces with High Dimensionality

For large k , all the techniques we discussed become **ineffective**:

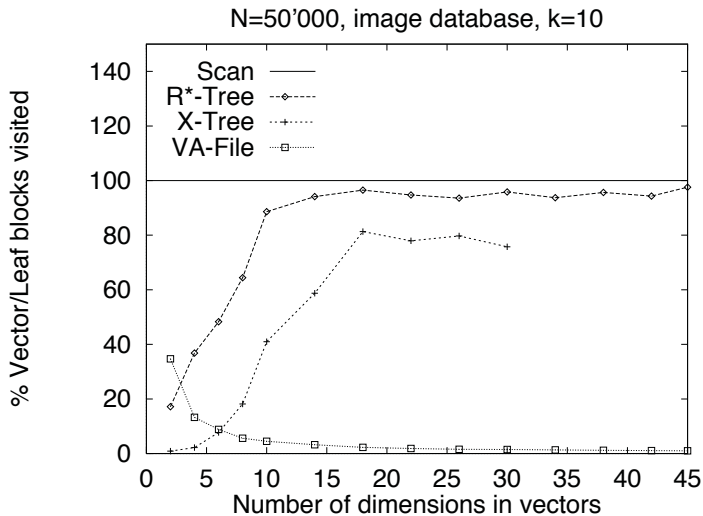
- *E.g.*, for $k = 100$, we'd get $2^{100} \approx 10^{30}$ partitions per node in a **point quad tree**. Even with billions of data points, **almost all** of these are empty.
- Consider a **really big** search region, cube-sized covering 95% of the range along **each** dimension:



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

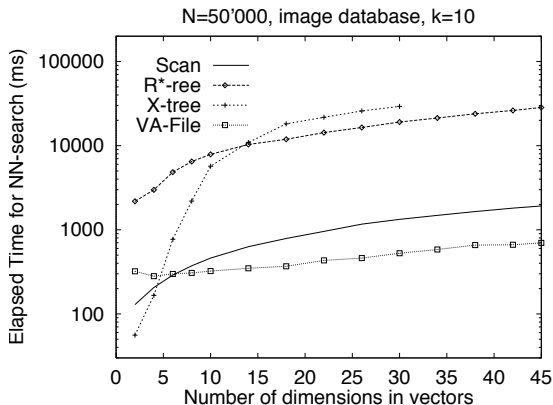
- We experience the **curse of dimensionality** here.

Page Selectivity for k -NN Search



Data: Stephen Bloch. What's Wrong with High-Dimensionality Search. *VLDB 2008*.

Query Performance in High Dimensions



- VA-File: **vector approximation file** ($|VA-File| \ll |data\ file|$)
- **Scan** VA-File and use it as a **filter** for actual disk pages.

Wrap-Up

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.

UB-Tree

use space-filling curve (Z-order) to linearize k -dimensional data, then use B⁺-tree.

Curse Of Dimensionality

most indexing structures become ineffective for large k ; fall back to seq. scanning and approximation/compression.