Basics (Background):

Multidimensional Index Structures

1

1 One-dimensional Index Structures
1.1 B-Tree (rep)

- B-Tree (is assumed to be known)
  - Are preferentially used in OLTP systems
  - Standard data structure in all relational DBS
  - Garantuees low number of disk accesses, efficient and stable insertion, search and deletion.
  - Compact storage, balanced, robust against dynamic updates

Def. as repetition: A B-tree is a balanced, ordered, multi-path tree:

Let k,h be two integers with k>0, h>= -1. A tree of the class T(k,h) is either

- An empty tree (h=-1) or
- An ordered tree in which
  1. Every path from the root to a leaf has the same length h,
  2. Each node outside of the root and the leaves has at least k+1 children and the root, if it is not itself a leaf, has at least 2 children, and
  3. Each node has at most 2k+1 children.

2

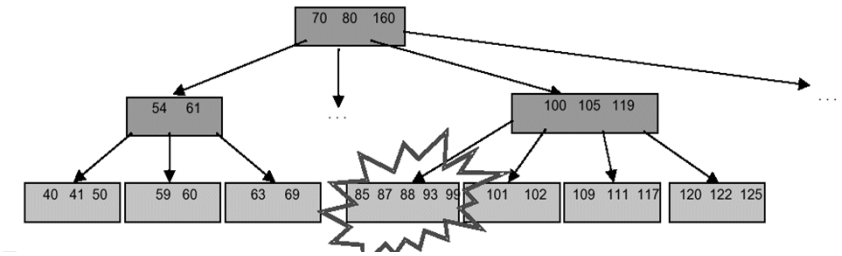
B-Tree

– Distinctive feature:
  - Each node and each leaf represent exactly one page
  - Extensive branching! Very small depth, very broad!

  - High degree of page filling of at least 50%
  - Logarithmic complexity when searching for, inserting and deleting tuples

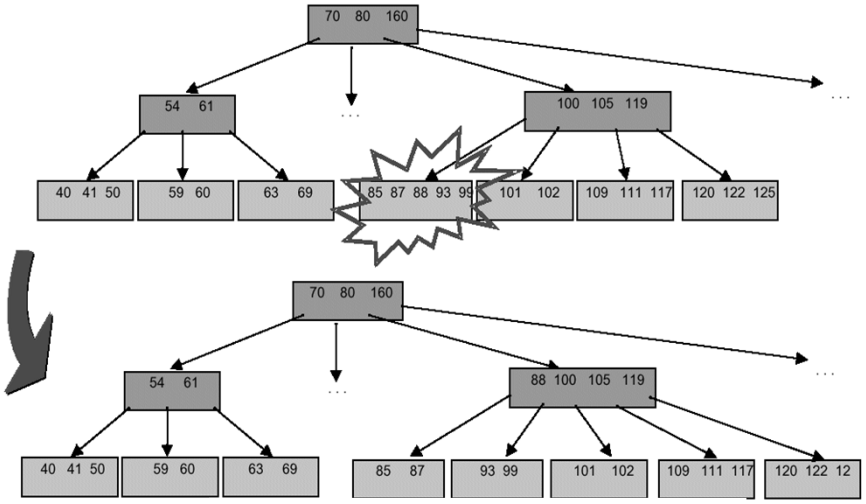
3

Insert value ‘88‘ in a B-Tree



4

Insert value ‘88‘ in a B-Tree



5

1.2 B*-Tree (rep)

B*-Tree:

– In order to obtain even greater branching:
  All values are stored in leaves, inner nodes contain only the shortest separators

In general:

– Relations are clustered by primary key in B-Trees
– Furthermore there are secondary indices (also as B-tree index, mostly in leaves without data, instead TIDs or primary key).

6

Problems with B-Tree and Variants in DW

- Example:
  - **Index on Product_Family(P), Country(C) and Year(Y)**
  - 2 Possibilities: three single indices or combined indices
  - Using single indices: physical clustering only via P!
  - Combined B-tree indices can only be followed in one direction (otherwise full table scan is required!)
  - i.e. queries which are not in the form PCY, e.g. only C and Y, are not supported
  - Combined B-tree indices on all foreign keys of the the fact table attain almost the same size as the fact table
- Problem: B-Tree index is asymmetrical!

7

2 Multidimensional Index Structures

- Multidimensional Indices:
  - Symmetrical treatment of all dimensions
  - Clustering inherent. Keep the spatial proximity of the data.
  - Are currently used especially in spatial databases
- Following below:
  - Grid-Files
  - R-Tree
  - UB-Tree
- There are also:
  - Quadtrees (Point-, Region-)
  - ....

8

2.1 Grid-Files

- See blackboard.

9

2.2 R-Tree

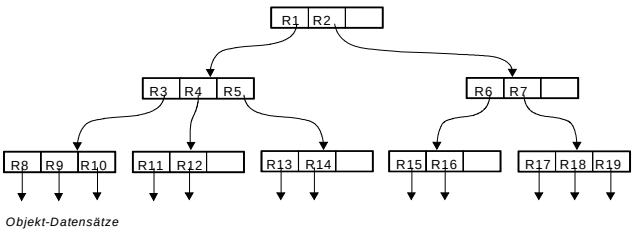
- Data structure for ***spatially expanded Objects***
  - **Extension of B*-tree for expanded objects (Gutman 1984)**
  - Main use: geographical objects (2D, 3D, n-D)
  - Transferable to general multidimensional data
- Suitable for
  - (spatial) match queries, range queries
- Characteristics:
  - Height balanced
  - Each node is the same size as a memory page
  - Differentiation between
    - Non-Leaf node (only for partitioning search space)
    - Leaf node (data object or reference to data object)

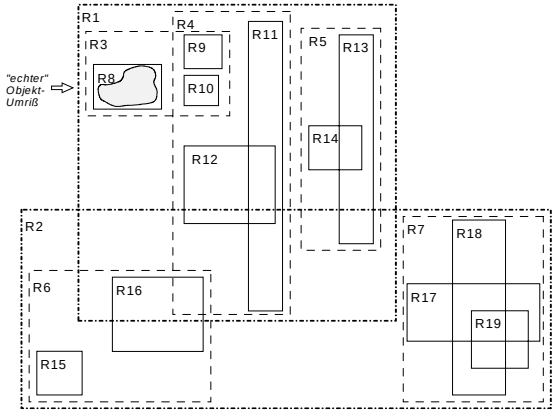
10

R-Tree Properties

- Approximation of spatial objects by the closest enclosing rectangle (in 2D cases, otherwise the smallest enclosing cuboid in 3D cases)
- Leaf nodes reference data objects and approximate this object
- Non-leaf nodes approximate the rectangles of their child nodes with the smallest (completely) enclosing rectangle
- Search space partitioning is generally not disjunct!
- Therefore a search in multiple partial spaces (i.e. paths) is required in certain cases
- Each node contains between m and M entries

11





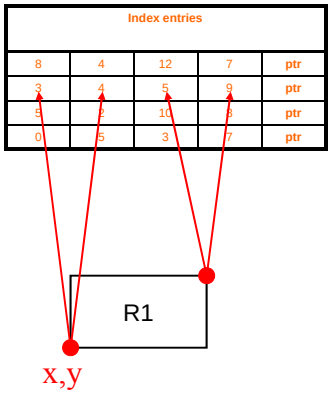
12

Properties of R-Trees

- Let M = maximum number of entries per node and
  m ≥ M / 2 the (given) minimum allocation per node
  - All leaf nodes (excluding Leaf = Root) contain between m and M index entries
  - Index entries in leaf nodes take the form (*I*,*OID*):
    I = the smallest enclosing rectangle for the object represented by *OID* (OID = Address of the object)
  - Each non-leaf node (excepting the root) has between m and M child nodes
  - For all index entries (*I*,*Child-Pointer*) in non-leaf nodes *I* is the smallest encosing rectangle for the rectangle of the child nodes
  - The root (excluding Root = Leaf) always has at least 2 child nodes
  - All leaf nodes are at the same height
  - With N index entries: Height(Tree) <= $\log_m(N)$ -1,
    as branching factor per level is at least *m*

13

Branching Degree M of the R-Tree



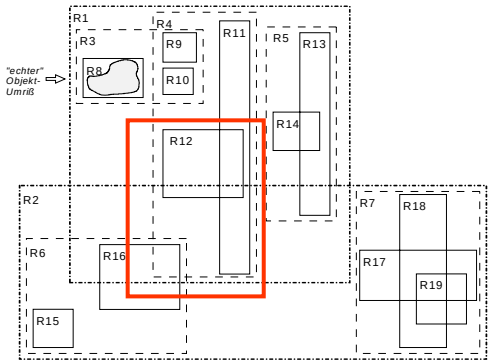
- A two dimensional rectangle can be represented by 4 numbers (x-min, y-min, x-max, y-max) of 4 Byte each.
- A pointer to a child node requires 4 Byte max.
- i.e. each index entries requires 20 Byte.

With1 kB pages: 50 Entries (=branching degree)
2 kB " 100 "
4 kB " 200 "

14

Search Space

- Searching
  - Similar to B-Tree, however more “keys” (here: search spaces) can be relevant per node
  - Search for all index entries (OIDs) whose rectangle overlaps the given search rectangle S
  - Starting point root W



Search Algorithm

- Algorithm:

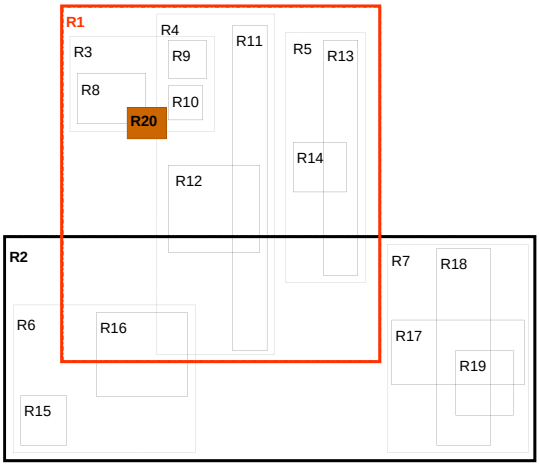
```
search(W,S,Result):
    { W: current search node, at start: root,
      S:   search rectangle
      E a currently viewed entry in node W }

    if W not leaf then
       ∀E : Rectangle(E.I) ∩ S ≠ ∅ ⇒ search(E.ptr,S,Result);
    if W leaf then  Result := { E.OID | E.I ∩ S ≠ ∅ }; return
```

Insertion

- Procedure
  - Similar to B-Tree
  - Insertions always happen in the leaf nodes
  - Unlike in B-Trees multiple leaf nodes (overlapping search spaces) are generally considered
  - Selection of suitable leaf node/sub-tree:
    Search rectangle of sub-tree which is not or minimally enlarged by insertio
  - On overflow split the node and insert an index entry in the parent node; thus maybe causing split of parent node, etc.

17

Example: Insert R20



- R1 contains R20 completely (as opposed to R2) ⇒ R1
- Analysis of all sub-trees of R1 to determine best insertion point

18

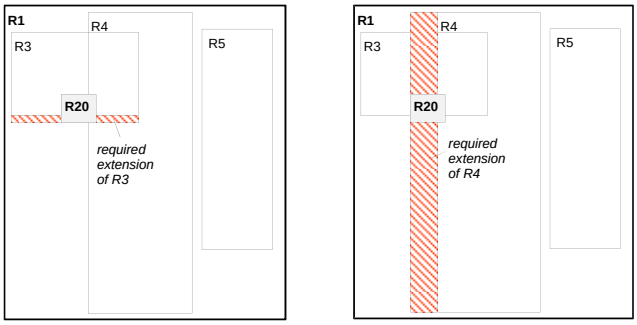
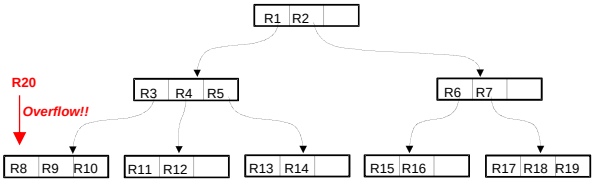


Fig.: Alternatives for inserting R20

- Determination of a suitable distribution of node entries
- Goal: compacte search spaces

  => *Minimisation of the enclosing rectangles (choose R3)*



19

Delete

- Procedure
  - Delete entry from leaf node
  - Recalculate the enclosing rectangle
  - If changes occur propagate changes to parent node (recursively if required)
  - If an underflow occurs combine nodes (similar to B-tree)
  - Difference: Combination possible with any sibling node (if not too many entries)
  - Selection criteria: smallest resulting enclosing rectangle

20

Algorithms

Note

In the following, let:

E.I be the associated rectangle to an index entry E and
E.p be the corresponding child pointer,
S be the search rectangle and
T be the root of the (sub-)tree.

21

Algorithm **Insert:**

I1. [Find position for new record.]
Invoke ChooseLeaf to select a leaf node *L* in which to place *E*.

I2. [Add record to leaf node.] If *L* has room for another entry, install *E*. Otherwise invoke SplitNode to obtain *L* and *LL* containing *E* and all the old entries of *L*.

I3. [Propagate changes upward.]
Invoke AdjustTree on *L*, also passing *LL* if a split was performed.

I4. [Grow tree taller.] If node split propagation caused the root to split, create new root whose children are the two resulting nodes.

Algorithm **ChooseLeaf :**

Select a leaf node in which to place a new index entry *E*.

CL1. [Initialize.] Set *N* to be the root node.

CL2. [Leaf check.] If *N* is a leaf, return *N*.

CL3. [Choose subtree.] If *N* is not a leaf, let *F* be the entry in *N*
whose rectangle F.I needs least enlargement to include E.I
Resolve ties by choosing the entry with the rectangle of smallest area.

CL4. [Descent until a leaf is reached.]
Set *N* to be the child node pointed to by *F.p* and repeat from CL2.

22

Algorithm **AdjustTree :**

Ascend from a leaf node *L* to the root, adjusting covering rectangles and propagating node splits as necessary.

AT1. [Initialize.] Set *N* = *L*. If *L* was split previously, set *NN* to the resulting second node.

AT2. [Check if done.] If *N* is the root, stop.

AT3. [Adjust covering rectangle in parent entry.] Let *P* be the parent node of *N*, and let E_N be the *N* 's entry in *P*. Adjust $E_N.I$ so that it tightly encloses all entry rectangles in *N*.

AT4. [Propagate node split upward.] If *N* has a partner *NN* resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.p$ pointing to *NN* and $E_{NN}.I$ enclosing all rectangles in *NN*. Add E_{NN} to *P* if there is room. Otherwise, invoke **SplitNode** to produce *P* and *PP* containing E_{NN} and all *P* 's old entries.

AT5. [Move up to next level.] Set *N* = *P* and set *NN* = *PP* if a split occurred. Repeat from AT2.

23

Algorithm **Delete :**

Remove index record E from an R-Tree

D1. [Find node containing record.] Invoke **FindLeaf** to locate the leaf node *L* containing *E*. Stop if the record was not found.

D2. [Delete record.] Remove *E* from *L*.

D3. [Propagate changes.] Invoke **CondenseTRee**, passing *L*.

D4. [Shorten tree.] If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm **FindLeaf :**

Given an R-tree whose root node is *T*, find the leaf node containing the entry *E*.

FL1. [Search subtrees.] If *T* is not a leaf, check each entry *F* in *T* to determine if *F.I* overlaps *E.I*. For each such entry invoke **FindLeaf** on the tree whose root is pointed to by *F.p* until *E* is found or all entries have been checked.

FL2. [Search leaf node for record.] If *T* is a leaf, check each entry to see if it matches *E*. If *E* is found return *T*.

24

Algorithm **CondenseTree :**

Given a leaf node L from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate nodes elimination upward as necessary. Adjust all covering rectangles on the path to the root, making them small if possible.

CT1. [Initialize.] Set *N* = *L*. Set *Q*, the set of eliminated nodes to be empty.
CL2. [Leaf check.] If *N* is a leaf, return *N*.

CT2. [Find parent entry.] If *N* is the root, go to CT6. Otherwise let *P* be the parent of *N*, and let E_N be *N*'s entry in *P*.

CT3. [Eliminate under-full node.] If *N* has fewer than m entries, delete E_N from *P* and add *N* to set *Q*.

CT4. [Adjust covering rectangle.] If *N* has not been eliminated, adjust E_N.I to tightly contain all entries in *N*.

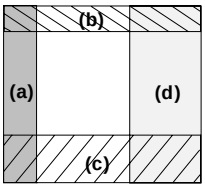
CT5. [Move up one level in tree.] Set *N* = *P* and repeat from CT2

CT6. [Re-insert orphand entries.] Re-insert all entries of nodes in set *Q*. Entries from eliminated leaf nodes are re-inserted in tree leaves as described in Algorithm **Insert**, but entries on higher-level nodes must be placed higher in the tree, so that leaves on their dependent subtrees will be on the same level as leaves of the main tree.

25

Procedure on Node Overflow

- Can sometimes be difficult to find a “good” distribution
  - Worst case: Afterwards identical search spaces in the parent node



  - In the above case: no real separation of the search space possible: No matter how one splits (with a 2-2 distribution), there are always two identical search spaces in the parent node.
  - Problem: **Large combinatory variety**
    - Number of possibilities ≈ 2^{M-1}
    - For realistic applications M = 50 .. 200 (see above)
    - Find optimal distribution = general mathematical optimisation problem
    - “Exact" solution in general zu costly ⇒ Heuristics
  - In general 2 procedures:
    - Quadratic Cost Algorithm
    - Linear Cost Algorithm

26

Heuristic 1: “Quadratic Cost Algorithm”

Algorithm **Quadratic Split :**

Divide a set of M+1 index entries into two groups. The cost is quadratic in M and linear in the number of dimensions.

QS1. [Pick first entry for each group.] Apply Algorithm **PickSeeds** to choose two entries to be the first elements of the groups. Assign each to a group.

QS2. [Check if done.] If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it in order for it to have the minimum number m, assign them and stop.

QS3. [Select entry to assign.] Invoke Algorithm **PickNext** to choose the next entry to assign. Add it to the group whose covering rectangle will have to be enlarged least to accomodate it. Resolve ties by adding the entry to the group with smaller area, then to the one with fewer entries, then to either. Repeat from QS2.

27

Algorithm **PickSeeds :**

Select two entries to be the first elements of the groups.

PS1. [Calculate inefficiency of grouping entries together.] For each pair of entries E_1 and E_2, compose a rectangle J including $E_1.I$ and $E_2.I$. Calculate
d = area(J) - area($E_1.I$) - area($E_2.I$) [22]

PS2. [Chose the most wasteful pair.] Choose the pair with the largest d.

Algorithm **PickNext :**

Select one remaining entry for classification in a group.

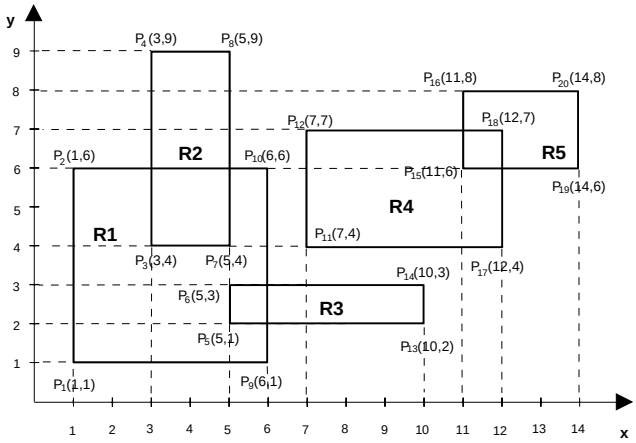
PN1. [Determine cost of putting each entry in each group.] For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include $E.I$. Calculate d_2 similarly for Group 2.

PN2. [Find entry with greatest preference for one group.] Choose any entry with the maximum difference beetween d_1 and d_2.

28

Example:

Given: following starting situation, M = 4



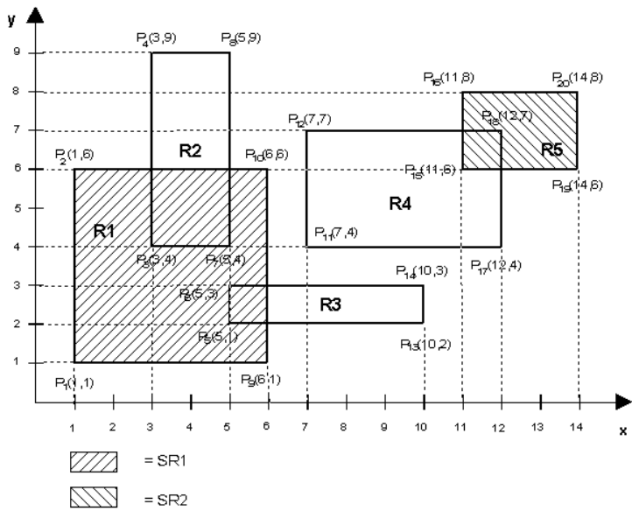
Entries to be distributed

Initialisation step:
search for maximally separated rectangles, these are then the seed for both partitions.

QS1:

area(R1) =	5 * 5	=	25
area(R2) =	2 * 5	=	10
area(R3) =	5 * 1	=	5
area(R4) =	5 * 3	=	15
area(R5) =	3 * 2	=	6

d(R1⊕R2)	=	total area - area(R1) -area(R2)	
	=	5 * 8 - 25 - 10	= 5
d(R1⊕R3)	=	9 * 5 - 25 - 5	= 15
d(R1⊕R4)	=	11 * 6 - 25 - 15	= 26
d(R1⊕R5)	=	13 * 7 - 25 - 6	= 60
d(R2⊕R3)	=	7 * 7 - 10 - 5	= 34
d(R2⊕R4)	=	9 * 5 - 10 - 15	= 20
d(R2⊕R5)	=	11 * 5 - 10 - 6	= 39
d(R3⊕R4)	=	7 * 5 - 5 - 15	= 15
d(R3⊕R5)	=	9 * 6 - 5 - 6	= 43
d(R4⊕R5)	=	7 * 4 - 15 - 6	= 7

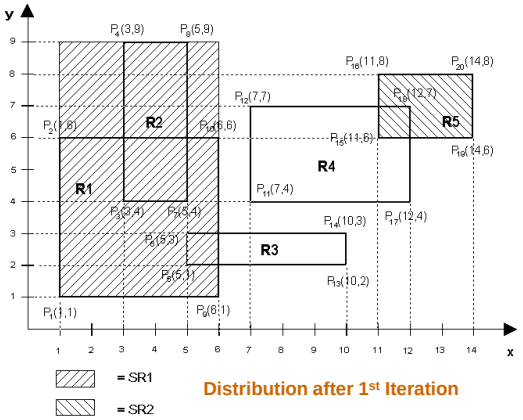


Initial distribution

Iteration step: assign the remaining rectangles to the most suitable partition (= minimal additional free area)

QS2/QS3:
1st Iteration: SR1 = { R1 }, SR2 = { R5 }
Entries still to be distributed: R2, R3, R4

If x =	SR1	SR2	Difference
d(R2 ⊕ x)	5*8 – 25 –10 = 5	11*5 – 10 – 6 = 39	34
d(R3 ⊕ x)	9*5 – 25 – 5 = 15	9*6 – 5 – 6 = 34	19
d(R4 ⊕ x)	11*6 – 25 –15 = 26	7*4 – 15 – 6 = 7	19

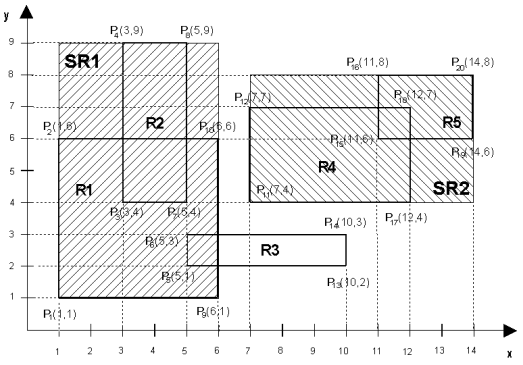


Distribution after 1st Iteration

2nd Iteration: SR1 = { R1, R2 }, SR2 = { R5 }
Entries still to be distributed: R3, R4

If x =	SR1	SR2	Difference
d(R3 ⊕ x)	72 - 40 - 5 = 27	54 - 5 - 6 = 43	16
d(R4 ⊕ x)	88 - 40 - 15 = 33	28 – 15 - 6 = 7	26

33



Distribution after 2nd Iteration

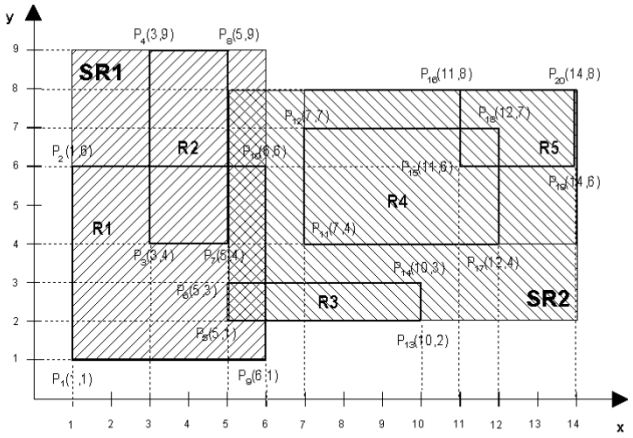
3rd Iteration: SR1 = { R1, R2 }, SR2 = { R4, R5 }
Entries still to be distributed: R3

If x =	SR1
d(R3⊕SR1)	72 - 40 - 5 = 27
d(R4⊕SR2)	54 - 5- 28 = 21

done!

Result: SR1 = { **R1, R2** }, SR2 = { **R3, R4, R5** }

34



Resulting distribution

Heuristic 2: „Linear Cost Algorithm“

Algorithm **Linear Split :**

This algorithm is linear in *M* and in the number of dimensions.
Linear Split is identical to **Quadratic Split** but uses
a different version of **PickSeeds** (see below).
PickNext simply chooses any of the remaing entries.

Algorithm **LinearPickSeeds :**

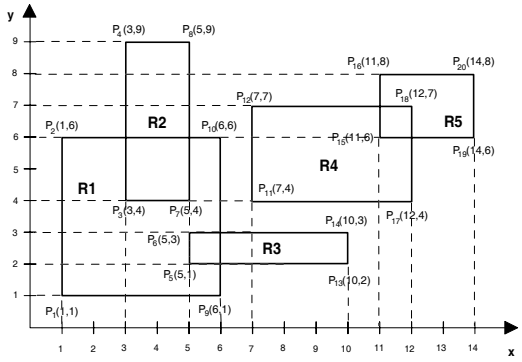
Select two entries to be first elements of the groups.

LPS1. [Find extreme rectangles along all dimensions.] Along each dimension, find the entry whose rectangle has the highest low side, and the one with the lowest high side. Record the separation.

LPS2. [Adjust for shape of the rectangle cluster.] Normalize the separations by dividing the widht of the entire set along the corresponding dimension.

LPS3 [Select the most extreme pair.] Choose the pair with the greatest normalized separation along any dimension.

Example for Linear Cost Algorithm:
Given: same starting situation as before, M = 4 (cf. Example for quadratic cost Algorithm)



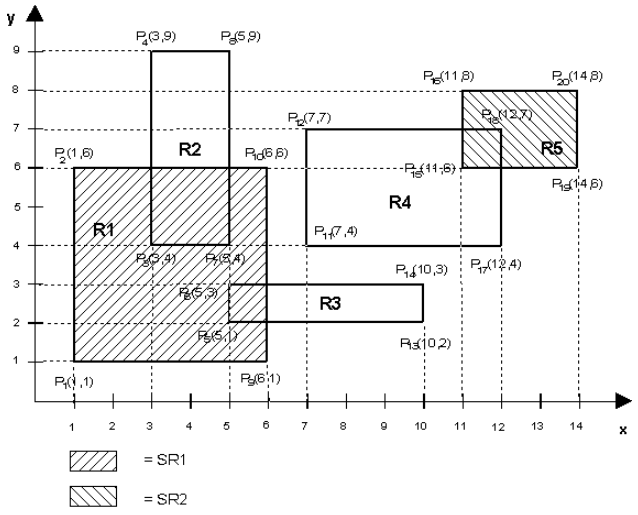
Entries to be distributed

Search for two rectangles as far from each other as possible:

	min(max(.)	max(min(.)	Diff.	normalised
x	6 (= R1)	12 (= R5)	6	6 : 13 = 0,462
y	3 (= R3)	6 (= R5)	3	3 : 8 = 0,375

37

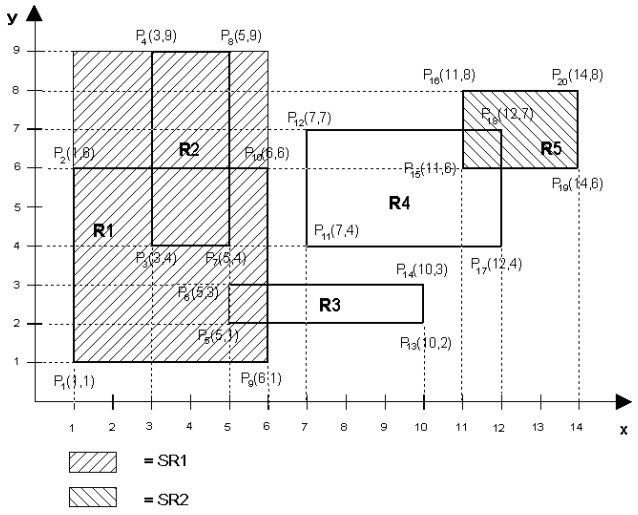
Initial distribution: SR1 = { R1 }, SR2 = { R5 }



Initial distribution

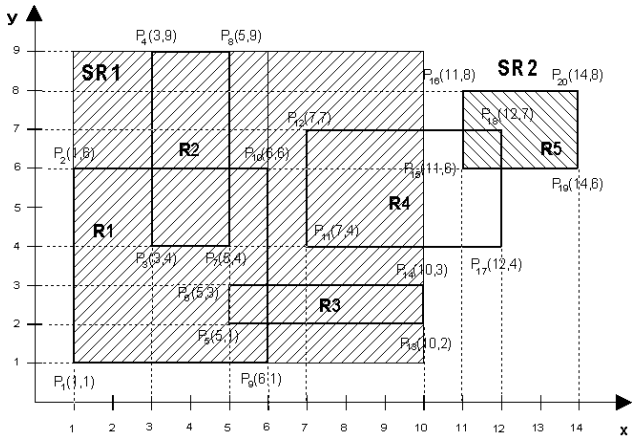
38

1st Iteration: SR1 = { R1 }, SR2 = { R5 }
Entries still to be distributed: R2, R3, R4
We select R2 ⇨ insert in SR1 (as smallest growth)



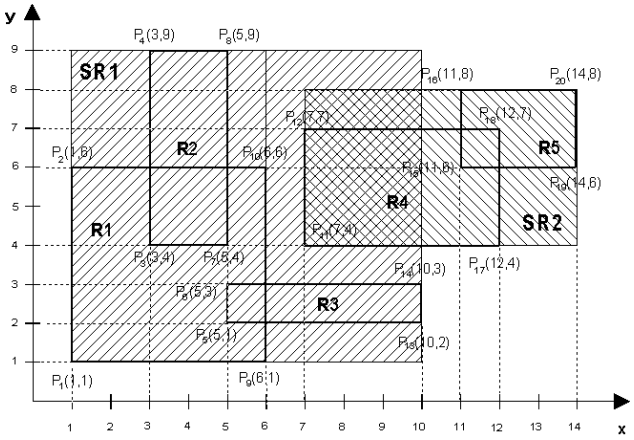
Distribution after 1st Iteration

2nd Iteration: SR1 = { R1, R2 }, SR2 = { R5 }
Entries still to be distributed: R3, R4
We select R3 ⇨ insert in SR1 (as smallest growth)



Distribution after 2nd Iteration

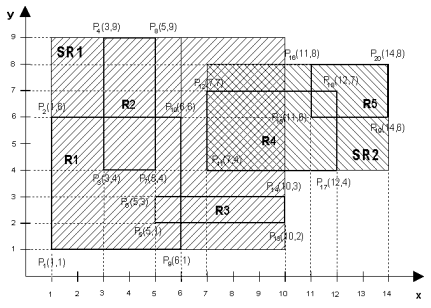
3rd Iteration: SR1 = { R1, R2, R3 }, SR2 = { R5 }
Entries still to be distributed: R4
R4 must be assigned to SR2, otherwise m ≥ M/2 not obtained
This results in:
Result: SR1 = { R1, R2, R3 }, SR2 = { R4, R5 }



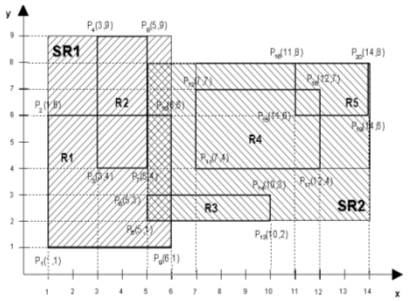
Resulting distribution

Result of Linear Cost Algorithm in Comparison to Quadratic Cost Algorithm:

Result from Linear Cost Algorithm:

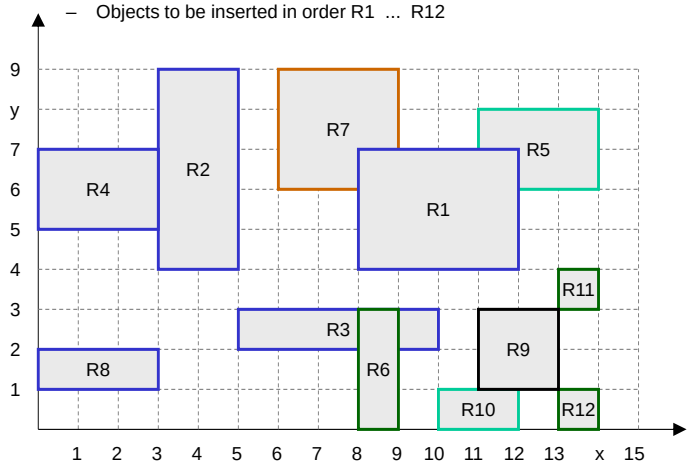


Result from Quadratic Cost Algorithm:



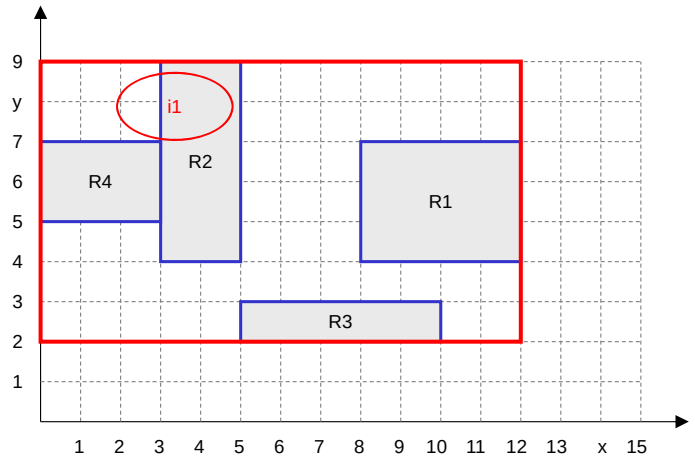
Example: R-Tree Construction
(using Linear Cost Algorithm)

- Given:
  - M = 4
  - Objects to be inserted in order R1 ... R12



43

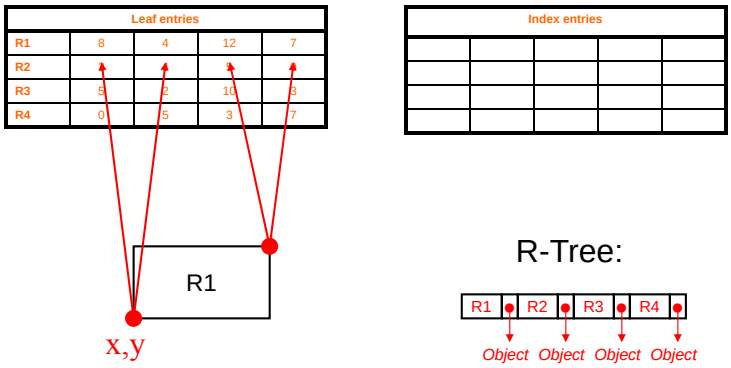
Starting situation:



Previously distributed objects

44

R-Tree after 4 insertions:

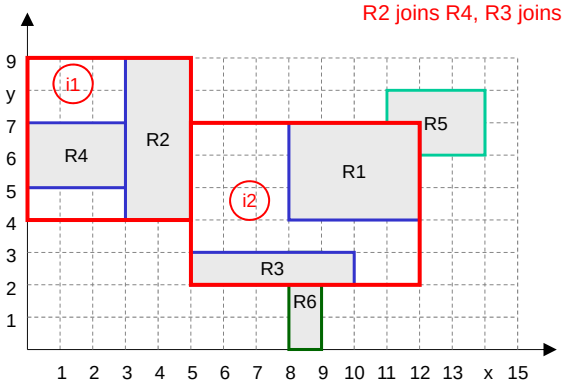


45

Node split (Linear-Split-Algorithm)

	min(x-max) resp. min(y-max)	max(x-min) resp. max(y-min)	Diff.	normalised
x	3 (=R4)	8 (=R1)	5	5 : 12 = 0,42
y	3 (=R3)	5 (=R4)	2	2 : 7 = 0,29

R2 joins R4, R3 joins R1



Index entries after the split

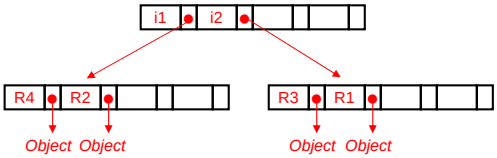
46

R-Tree after Split:

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7

Index entries				
i 1	0	4	5	9
i 2	5	2	12	7

R-Tree:



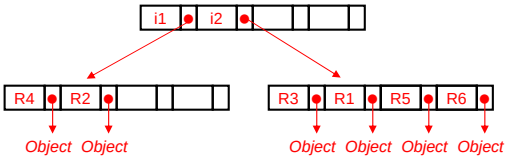
47

Insert R5 and R6:

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3

Index entries				
i 1	0	4	5	9
~~i 2~~	~~5~~	~~2~~	~~12~~	~~7~~
~~i 2~~	~~5~~	~~2~~	~~14~~	~~8~~
i 2	5	0	14	8

R-Tree:

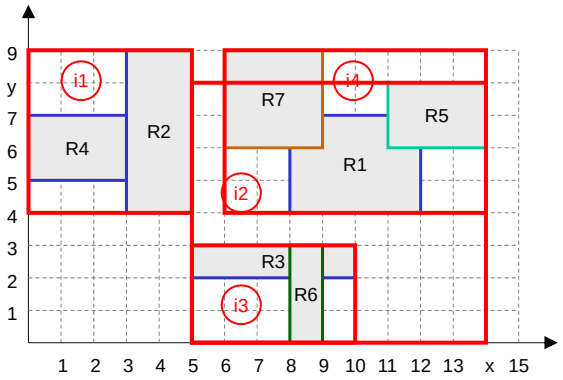


48

Insert R7

	min(max(.))	max(min(.))	Diff.	normalised
x	9 (=R6)	11 (=R5)	2	2 : 9 = 0,222
y	3 (=R3, R6)	6 (=R5)	3	3 : 8 = 0,375

Distribution (i2 is replaced): i3 := {R3, R6}, i4 := {R5, R1}
Insert R7: join i4



49

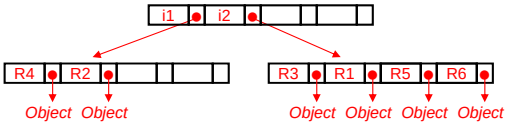
Insert R7:

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3
R7	6	6	9	9

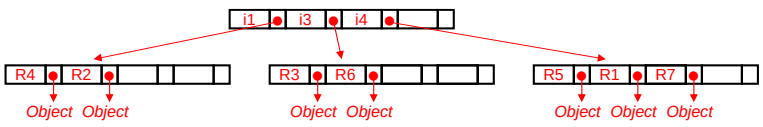
Index entries				
i 1	0	4	5	9
i 2	5	0	14	8
i 3	5	0	10	3
i 4	6	4	14	9

Neu-Adjustierung von i2!

R-Tree before inserting R7:



R-Tree after inserting R7:



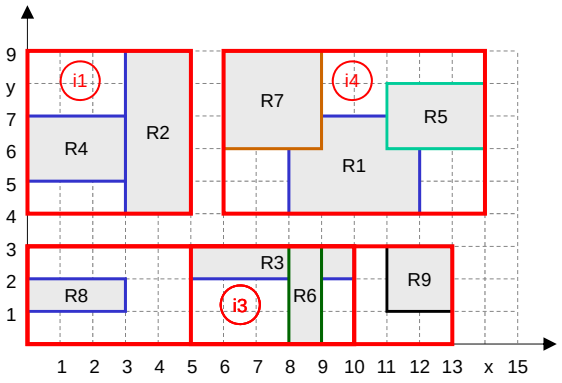
50

Insert R8 and R9:

Determine insertion point (sub-tree) for R8

- area enlargement when inserted in i1 : 5 * 3 = 15
- area enlargment when inserted in i3 : 5 * 3 = 15 ← arbitrary choice

Insertion point for R9 : i3



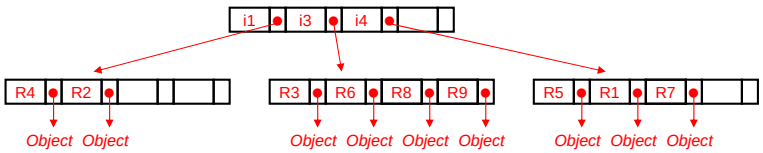
51

Insert R8 and R9

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3
R7	6	6	9	9
R8	0	1	3	2
R9	11	1	13	3

Index entries				
i 1	0	4	5	9
i 3	5	0	10	3
i 4	6	4	14	9
i 3	0	0	10	3
i 3	0	0	13	3

R-Tree



52

Prepare insertion of R10

Insertion point (subtree) for R10: i3 ⟶ Overflow!

- Index i3 needs to be split

	min(max(.))	max(min(.))	Diff.	normalised
x	3 (=R8)	11 (=R9)	8	
y	2 (=R8)	2 (=R3)	0	

First we consider R6 (arbitrarily chosen) – different solution if R3 were chosen!

Initialisation: (i3 is replaced by i5 and i6)

i5: {R8 } + R6 : area enlargment = 7 * 3 + 6 = 27

i6: {R9 } + R6 : area enlargment = 3 * 3 + 2 = 11 ⟵

Mapping: R6 ⟶ i6: { R9, R6 }
R3 ⟶ i5: { R8, R3 } (otherwise < m entries)

Insertion point (sub-tree) for R10: i6 / for R11: i6

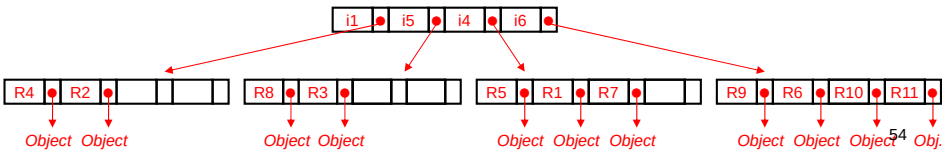
53

Insert R10 and R11

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3
R7	6	6	9	9
R8	0	1	3	2
R9	11	1	13	3
R10	10	0	12	1
R11	13	4	14	5

Index entries				
i 1	0	4	5	9
~~i 3~~	~~5~~	~~0~~	~~10~~	~~3~~
i 4	6	4	14	9
i 5	0	0	10	3
~~i 6~~	~~8~~	~~0~~	~~13~~	~~3~~
i 6	8	0	14	4

R-Tree

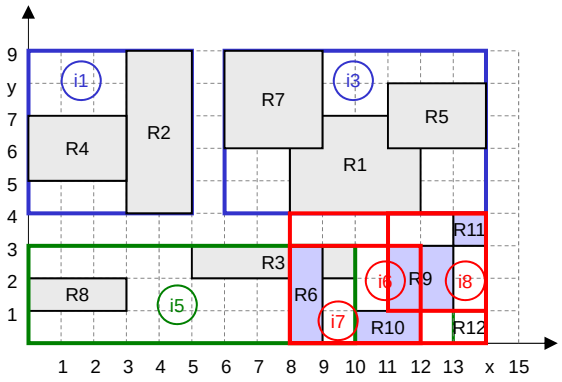


54

Prepare insertion of R12

	min(max(.))	max(min(.))	Diff.	normalised
x	9 (=R6)	13 (=R11)	4	4 : 6 = 0,67
y	1 (=R10)	3 (=R11)	2	2 : 4 = 0,5

i7: { R10, R6 } **+ R12: Enlargement = 6**
i8: { R11, R9 } **(otherwise < m entries)** **+ R12: Enlargement = 3**



55

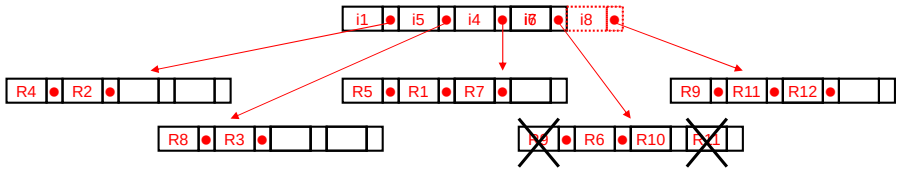
Insert R12

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3
R7	6	6	9	9
R8	0	1	3	2
R9	11	1	13	3
R10	10	0	12	1
R11	13	3	14	4
R12	13	0	14	1

Index entries				
i 1	0	4	5	9
i 4	6	4	14	9
i 5	0	0	10	3
~~i 6~~	~~8~~	~~6~~	~~14~~	~~4~~
i 7	8	0	12	3
~~i 9~~	~~11~~	~~1~~	~~14~~	~~4~~
i 8	11	0	14	4

Split of the root node!
Redistribution of entries i1, i5, i3, i7

R-Tree



56

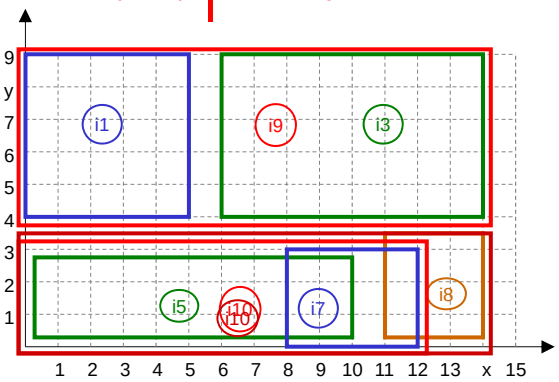
Root node split

	min(max(.))	max(min(.))	Diff.	normalised
x	5 (= i1)	8 (= i7)	3	3 : 14 = 0,214
y	3 (= i5, i7)	4 (= i1, i3)	1	1 : 9 = 0,111

Split the previous root node into:

i9 : { i1 }
i10: { i7 }
+ i3: enlargment = 9 * 5 = 45
+ i3: enlargment = 8 * 6 + 2 * 3 = 52

i9 : { i1, i3 }
i10: { i7, i5 }
+ i8: enlargment = 14 * 4 = 56
+ i8: enlargment = 12 * 2 + 2 * 5 = 34



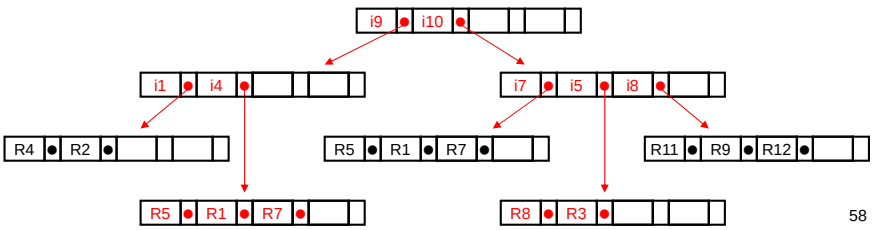
57

Insert i8

Leaf entries				
R1	8	4	12	7
R2	3	4	5	9
R3	5	2	10	3
R4	0	5	3	7
R5	11	6	14	8
R6	8	0	9	3
R7	6	6	9	9
R8	0	1	3	2
R9	11	1	13	3
R10	10	0	12	1
R11	13	3	14	4
R12	13	0	14	1

Index entries				
i 1	0	4	5	9
i 3	5	0	10	3
i 5	0	0	10	3
i 7	8	0	12	3
i 8	11	1	14	9
i 9	0	4	14	9
i10	0	0	14	4

R-Tree



58

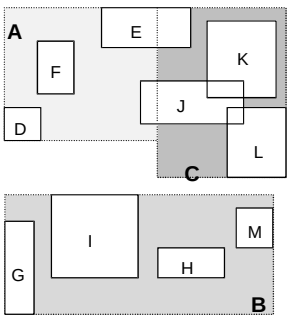
Evaluation R-Tree

- Properties
  - ⊕ dynamic, height-balanced structure
  - ⊕ suitable for storing large amount of data
  - ⊕ rapid access if data is distributed “well”
  - ⊖ Rectangle approximation is sometimes very imprecise
  - ⊖ Quality of search space paritioning dependent on the insertion order
  - ⊖ possibly largely overlapping search spaces
  - ⊖ In the worst case the whole index has to be searched
  - ⊖ Very costly update algorithms
  - ⊖ therefore not ideally suited for data with high dynamics

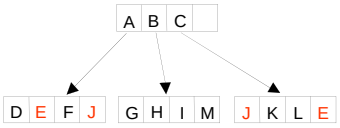
59

R^+-Tree

- Properties
  - Avoids overlapping search space regions in the leaf nodes, thereby achieving ***disjuncte search space partitioning***
  - ***Representative of the clipping approach***, however not true clipping
  - Objects which don’t completely fall in a region are stored multiply (redundantly) in the leaf nodes.



a) Rectangles to be managed



b) resulting R^+-Tree

60

- Search

  (with W = root of R^+(sub-)tree and R = search rectangle)

  If W not Leaf then [search in non-leaf nodes]
  forall Entries (p,RECT) of R do
  if RECT $\cap$ W $\neq \varnothing$ then Search($p\uparrow$, S $\cap$ RECT);
  else [search in leaf nodes]
  forall Object RECT in R do
  if RECT $\cap$ W $\neq \varnothing$ then return(Object);
- Insert
  - Where applicable (logical) “cutting” of the object rectangle along search space borders.
  - Insert in all search spaces resp. corresponding leaf nodes in which an overlap occurs.
- Node split/combination
  - More complex algorithms in comparison to R-Tree
- Overall
  - Due to redundant storage may require more storage for the index
  - The advantage of disjunct search spaces can be annulled.

61

R^*-Baum

- Variant of R-Tree
- Overlapping search spaces (wie R-Tree)
- Adapted insertion and split algorithms in order to obtain following minimisations:
  - Area of search space rectangles
  - Overlap of search space rectangles
  - Circumference of search space rectangles

62

Insertion

Algorithm **ChooseSubtree**

First of all, the node N where the entry is to be inserted is determined. If N is already full, then the **Split** algorithm needs to be executed.

CS1. N := Root

CS2. if (N is Leaf) then
return N
else
if (the child pointers in N point to leaf nodes) then
(* determine the minimal overlap costs*)
Select the entry in N, whose rectangle causes the least increase in overlap with other search space rectangles after inserting the new entry. If there are multiple similarly suitable candidates, insert the entry where the search space rectangle is least enlargened.

else
(* determine the minimal area costs*)
Select the entry in N, whose rectangle area increases least after inserting the new entry. If there are multiple suitable candidates, select the search space rectangle whose perimeter increases least.

End

CS3. N := Child node, which the selected entry in CS2 points to. Goto CS2.

63

Node Split

Algorithm **Split**

S1. Call **ChooseSplitAxis** to determine the axis a along which the split should be executed.

S2. Call **ChooseSplitIndex** to determine the best distribution of the entries in two groups along axis a.

S3. Distribute the entries in both groups.

Algorithm **ChooseSplitAxis**

CA1. for (each Axis) do
Sort the entries once by the lower value and once by the higher value of their rectangles. Split the entries in the middle (M/2 +1) and M. Determine perimeter U1 and U2 of the two resulting search space rectangles as well as S := U1 + U2.

end

CA2. Select the axis with minimal S as split axis.

Algorithm **ChooseSplitIndex**

CI1. Along the selected split axis, choose the distribution which results in a minimal overlap of the new search space rectangles. In case of equality choose the distribution with the smallest total area of the search space rectangles.

64

Forced Re-Insert

In order to attenuate the quality of search space distribution from the insertion order, the following **Insert**-Algorithm can be used. It removes *p* entries on a node split and reinserts them anew.

Algorithm **Insert**

I1. Call **ChooseSubtree** to determine a node in which the new entry E should be inserted. Input parameter for ChooseSubtree is the tree-level at which E should be inserted.

I2. if (N has less than M Entries) then
    put E in N;
    if (N has M Entries) then
    call **OverflowTreatment** with the level of N as input parameter

I3. if (**OverflowTreatment** caused Split) then
    propagate **OverflowTreatment** upwards, if needed

I4. Adapt all search space rectangles in the input path so that they become smallest enclosing rectangles for their child rectangles again.

65

Algorithm **OverflowTreatment**

OT1. if (InputLevel ≠ Root and first call of OverflowTreatment for this tree-level)
    then
    call ReInsert
    else
    call Split
    end

Algorithm **ReInsert**

RI1. For all M+1 entries of a node N, calculate the distances between the centres of ist rectangles and the centre of their enclosing rectangle.

RI2. Sort the entries descending *according* to the distances determined in RI1.

RI3. Remove the first *p* entries from N from the entries determined in RI2 and refit the enclosing rectangle correspondingly.

RI4. Call **Insert** to re-insert the p entries anew. Start with the sorting according to the largest distance (= far insert) or the smallest distance (= close insert).

66

2.3 UB-Baum

Developed by Prof. R. Bayer and the Mistral-Group, TU München.
In the following the original slides (!) (in more detail as Book)

Design Goals:

- clustering tuples on disk pages while preserving spatial proximity
- efficient incremental organization
- logarithmic worst-case guarantees for insertion, deletion and point queries
- efficient handling of range queries
- good average memory utilization

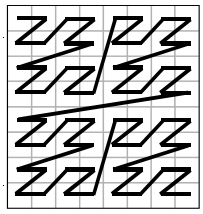
67

The design goals on this slide are generic for designing an access method, both for single-attribute or multi-attribute access methods. However, most multi-attribute methods do not fulfill all of the design goals. R-Trees, for instance, do not offer a good average memory utilization. Most multidimensional access methods like R*-Trees, Grid-Files or kd-B-Trees do not allow for efficient incremental organization, e.g., by requiring forced reinsertion, Grid-Splits or even complex reorganizations. Since UB-Trees (as we will see) rely on standard B-Trees for their storage organization of so-called Z-regions, they inherit all of the above properties from the underlying B-Tree structure.

68

Z-Ordering

$$Z(x) = \sum_{i=0}^{s-1} \sum_{j=1}^{d} x_{j,i} \cdot 2^{i \cdot d + j - 1}$$



(a)

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(b)

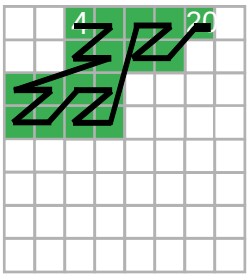
69

- Z(x) is a bijective function that computes for every tuple x its Z-address, i.e., its position on the space filling Z-curve. The slide presents the Z-addresses (or Z-values) for an 8x8 universe. Z-values are efficiently computed by bit-interleaving as described e.g. by Orenstein and Merret in 1984. An additional animation
- `http://mistral.in.tum.de/results/presentations/ppt/zaddress.ppt`
  on the Mistral Web Site describes bit-interleaving.

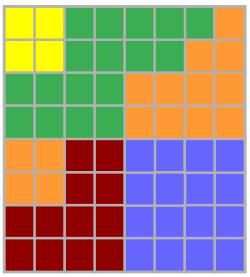
70

Z-regions/UB-Trees

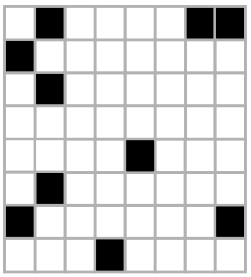
A *Z-region* [α : β] is the space covered by an interval on the Z-curve and is defined by two Z-addresses α and β.



Z-region [4 : 20]



UB-Tree partitioning: [0 : 3],[4 : 20], [21 : 35], [36 : 47], [48 : 63]



point data creating the UB-Tree on the left for a page capacity of 2 points

71

- A Z-region is the space covered by an interval on the Z-curve. Thus a Z-region has two meaningful interpretations, a linear interpretation as an interval as well as a spatial interpretation. The left part of this animation shows the Z-region [4:20] and its spatial interpretation. The spatial extent of the Z-region becomes clearer if we draw the Z-region into the picture. Please note that bit-interleaving is an efficient means to calculate the Z-value for a tuple (or the inverse, i.e., the tuple values for a given Z-value). Thus we can arbitrarily switch between the linear Z-space and the geometric interpretation.
- The middle part shows a Z-region partitioning (or also called UB-Tree partitioning) which is a disjoint set of Z-regions whose union covers the entire multidimensional space. In this picture the partitioning consists of 5 Z-regions. Most Z-regions preserve spatial proximity, i.e., neighboring points of a given point are in the same region with a high probability. The orange region [21 : 35] consists of two disconnected parts. If a Z-region could consist of many disconnected parts, this would prevent Z-regions from being suitable for clustering. However, [Mar99] gives a proof that regardless of the dimensionality of the Z-ordered space (i.e., not only for 2d) the number of not connected parts of a Z-region is at most two.

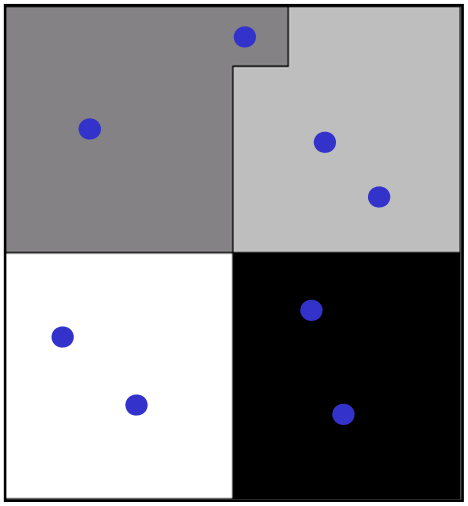
72

- We can consider a Z-region corresponding to a disk page, i.e., being a container of a fixed or variable capacity storing tuples which fall into the spatial extent of the Z-region. The right part of the animation shows a point distribution of 10 points which with a page capacity of two points per page might be stored in the Z-region partitioning of the middle picture of this slide .
- We use a B-Tree to store the upper limit of the Z-value of each Z-region and call the corresponding B-Tree storing the Z-region organization Universal B-Tree (UB-Tree) [Bay96, Mar99]

73

74

UB-Tree Insertion 1/2/3/4



75

- UB-Tree disk pages correspond to Z-regions. Each tuple is stored on a disk page corresponding to the Z-region that this tuple spatially belongs to. Each Z-region can store a certain capacity of tuples and must be split into two Z-regions during insertion if the page capacity of a Z-region is exceeded. Details can be found in [Bay96] and [Mar99]

76

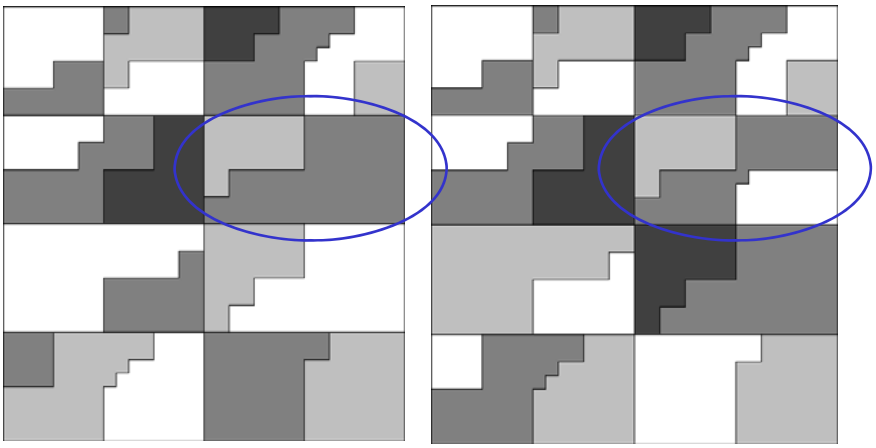
- This animation shows how data is inserted into a two-dimensional UB-Tree assuming a page (Z-region) capacity of two tuples (or points). We start with an empty UB-Tree consisting of a single Z-region corresponding to one disk page (the root node of the UB-Tree). Now points are inserted into the universe (indicated by red circles). As soon as the third point is inserted, the Z-region covering the the entire universe must be split into two Z-regions. This is done by choosing a separator Z address on that page which ensures that 50% of the tuples stored already on that page will have a lower value (in Z-order) than the separator. Choosing this separator for the split ensures a page utilization of 50%. To optimize the geometric shape of Z-regions, an additional heuristics exploiting the remaining freedom of choice for the separator may be used to avoid fringes (however, this optimization is not used in the animation above). In addition, the geometric shape of the region may be improved (be more rectangular) by lowering the page utilization and thereby offering a greater freedom of choice for the separator selection. However, in practice one can remain with the 50% choice.

77

- After the (in this case horizontal) split, the UB-Tree consists of two Z-regions. Inserting further points causes further splits each time the page capacity of a Z-region is exceeded. This animation continues until a UB-Tree consisting of 4 Z-regions storing 8 tuples has been created.
- In the enclosed file
  `http://mistral.in.tum.de/results/presentations/ppt/insert.ppt`
  one can find another animation of the UB-Tree insertion, using our standard view with a screen split into three sections CODE, Z-SPACE and GEOMETRIC SPACE (in this presentation the range query animation uses that layout) which allows to see the algorithm running view the code, the events happening in geometric space as well as the events happening in linear Z-space as stored in the B-Tree of the UB-Tree.

78

UB-Tree Insertion 18/19



79

- This slide shows that insertion into UB-Trees is a local operation and thus allows for efficient incremental updates. We have inserted further points into the UB-Tree which induced further splits into 18 Z-regions (left UB-Tree visualization). The UB-Tree on the right side was created by inserting further points into the last Z-region (in Z-order) of the second quadrant (in the UB-Tree on the left). It is important to note that only this single Z-region is split, resulting in one update and one write of B-Tree leaf pages (plus possible further splits on the upper B-Tree levels). This is a major difference to other multidimensional access methods like R-Trees or Grid-Files.

80

Multidimensional Range Query

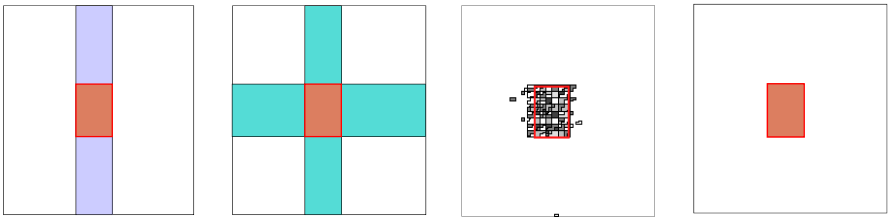
SELECT * FROM table
WHERE (A_1 BETWEEN a_1 AND b_1) AND
(A_2 BETWEEN a_2 AND b_2) AND
.....
(A_n BETWEEN a_n AND b_n)

81

- The goal of a multidimensional access methods is to efficiently answer multidimensional range queries which are created by the SQL query template above.

82

Theoretical Comparison of the Range Query Performance



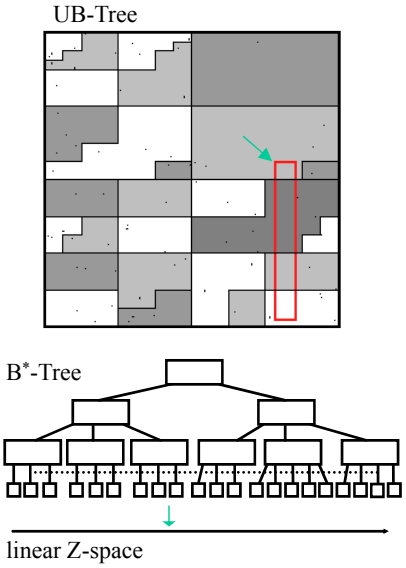
composite key clustering B-Tree	multiple B-Trees, bitmap indexes	multidimensional index	ideal case
s_1*P	$s_1*i_1+s_2*i_2+s_1*s_2*T$	$s_1^{\uparrow}*s_2^{\uparrow}*P$	s_1*s_2*P

83

- This slide illustrates how range queries are processed with access methods that are standard in today's relational DBMS. For simplification of our illustration we assume uniformly distributed data as well as independence of the dimensions. We assume a table consisting of P disk pages and a query box with the selectivities s1 and s2. In the ideal case we thus have to retrieve s1*s2*P disk pages to answer the query. With a composite key B-Tree, however, only the leading dimension of the composite key can be utilized, resulting in reading s1*P disk pages in the blue stripe. The result set is then determined by post filtering the tuples in main memory after retrieval. With bitmap indexes or multiple B-Trees, index intersection results in reading row ids or bitmaps with sizes s1*i1 and s2*i2 for index sizes of i1 respectively i2 pages. After this intersection, the result set tuples are retrieved by random access, resulting in s1*s2*T page reads, if T tuples are stored in the table. Note the difference between T (the number of tuples in the table) and P (the number of pages in the table). With an average of 30 tuples (empirical value from our project partners) per page bitmap indexes or multiple B-Trees are immediately more than 30 times worse than the ideal case. In contrast to that, a multidimensional index clusters the data more symmetrically with respect to all dimensions. Since this clustering or partitioning is discrete, there is always an overhead. However, with large database sizes the overhead gets smaller. In general this means that multidimensional indexes approximate the ideal case with some kind of ceiling function for each selectivity.

84

```
rangeQuery(Tuple ql, Tuple qh)
{
  Zaddress start  = Z(ql);
  Zaddress cur    = start;
  Zaddress end    = Z(qh);
  Page page = {};

  while (1)
  {
    cur = getRegionSeparator(cur);
    page = getPage(cur);
    outputMatchingTuples(page, ql, qh);
    if ( cur >= end ) break;
      cur = getNextZAddress(cur, start, end);
  }
}
```

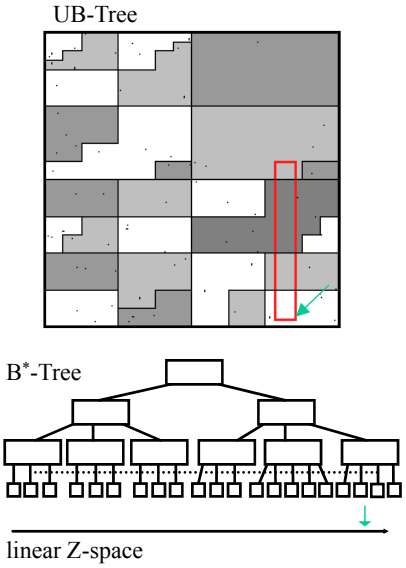


85

- This slide and the following slides explain the UB-Tree range query algorithm. The screen is divided into 4 sections: The left part shows the code, the right part shows the geometrical interpretation of the UB-Tree (UB-Tree), the linear B*-Tree as well as the linear Z-space. Please note that the UB-Tree and the B*-Tree on the linear Z-space are merely two different interpretations or visualizations of the same data set. In the algorithms we can arbitrarily switch between both representations. In the following we show how the read query box defined by the tuples ql and qh with (ql, qh) is processed by the range query algorithm.

86

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

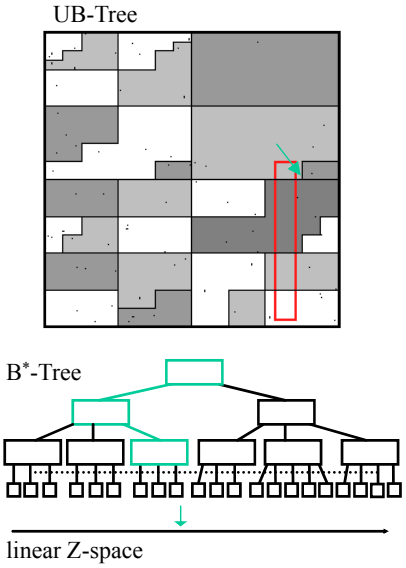


87

- First the algorithm calculates the start and the end Z-values of the query box coordinates ql and qh.

88

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
       cur = getNextZAddress(cur, start, end);
   }
}
```

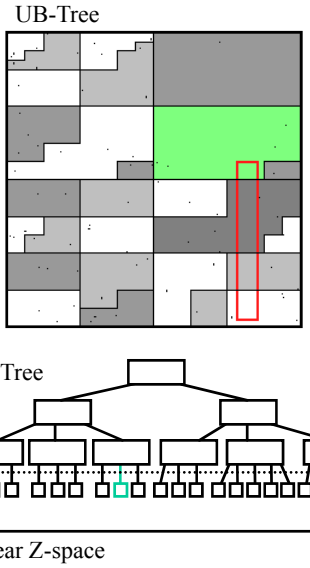


89

- First the region separator of the Z-region where the start point of the query box is located is determined by a single B-Tree point search (in SQL: SELECT min(Z) where Z>cur) as illustrated by the blue path through the B-Tree.

90

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

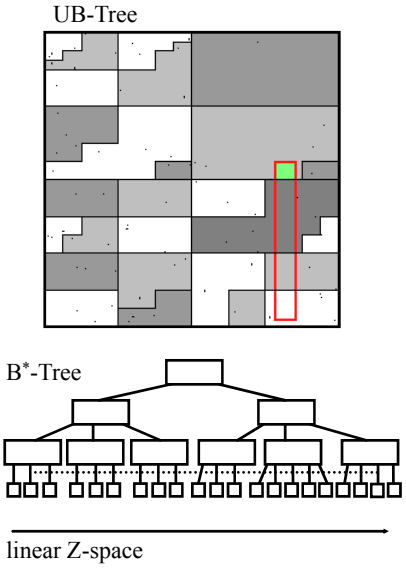


91

- Then the leaf page corresponding to that Z-region is retrieved as illustrated by the blue leaf page in the B-Tree as well as the green colored space in the UB-Tree.

92

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```

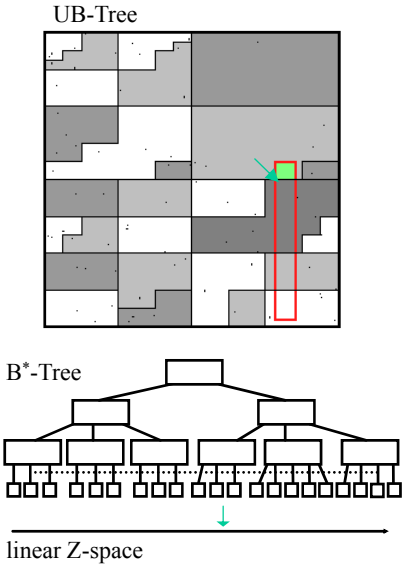


93

- Then all tuples of that page that are inside the query box are returned.

94

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
     cur = getNextZAddress(cur, start, end);
   }
}
```

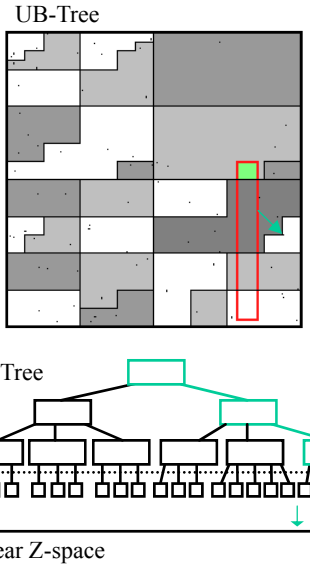


95

- Now the next Z-address intersecting the query box is computed, i.e.,
  cur = min { Z(x) where x in [[ql, qh]] and Z(x) > cur)
  This is achieved by an algorithm that only requires O(n) bit operations (copy and compare) where n is the number of bits (i.e., length) of the Z-address cur.
- The details of that algorithm can be found in [Mar99] and [RMF+00].

96

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
       cur = getNextZAddress(cur, start, end);
   }
}
```

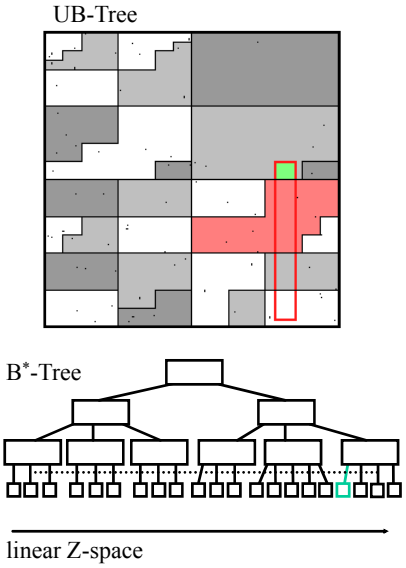


97

- After that the algorithm proceeds in the same way as described before until the entire query box has been processed, i.e., cur > end.

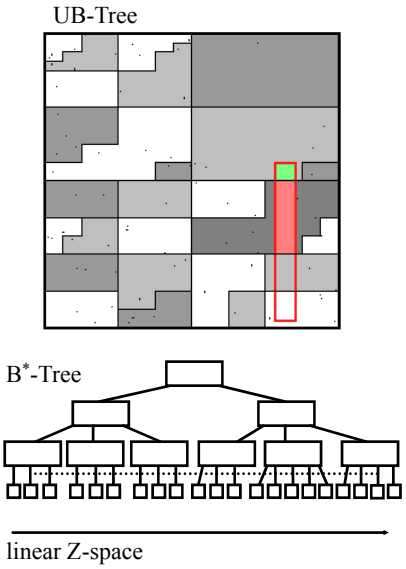
98

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



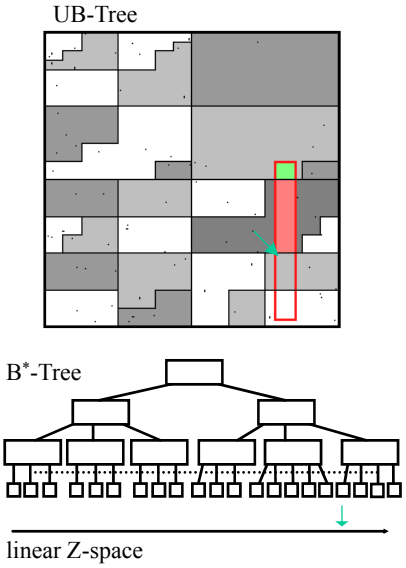
99

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



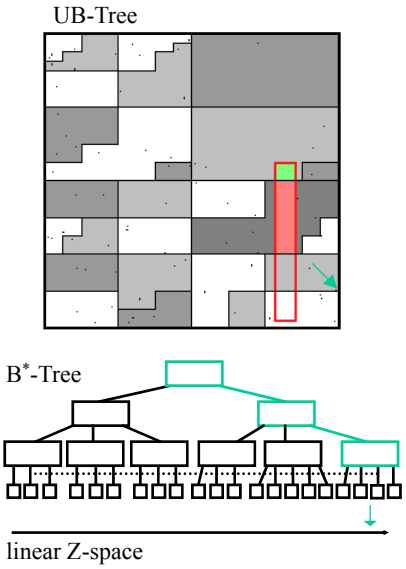
100

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
     cur = getNextZAddress(cur, start, end);
   }
}
```



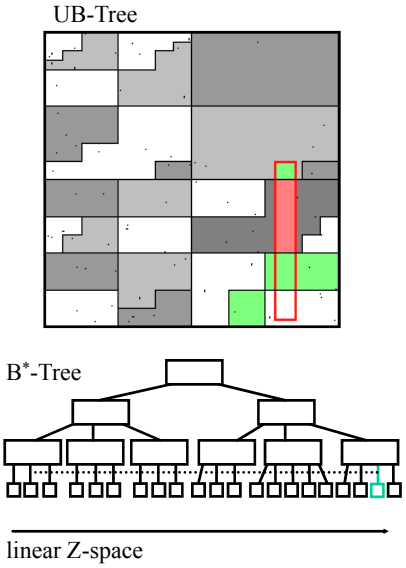
101

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
     cur = getNextZAddress(cur, start, end);
   }
}
```



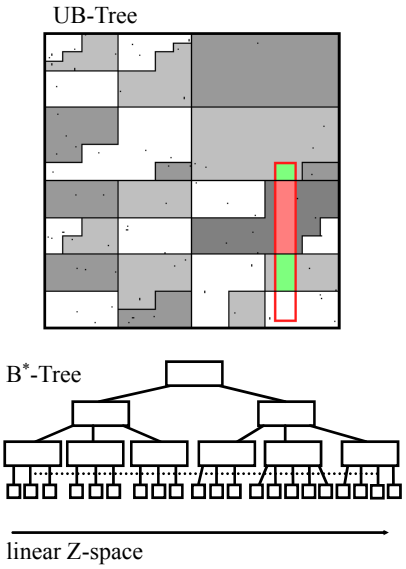
102

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



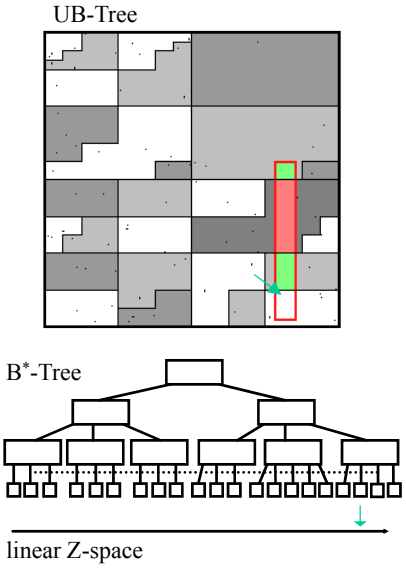
103

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



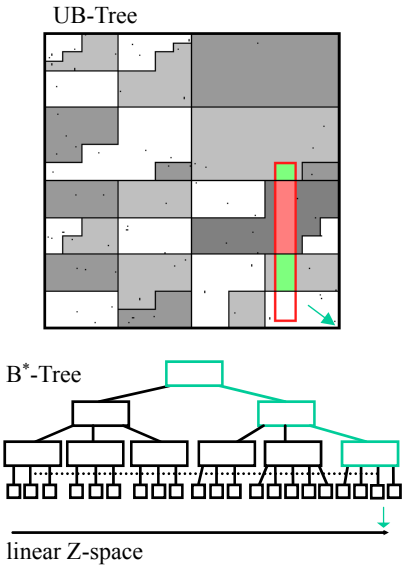
104

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
     cur = getNextZAddress(cur, start, end);
   }
}
```



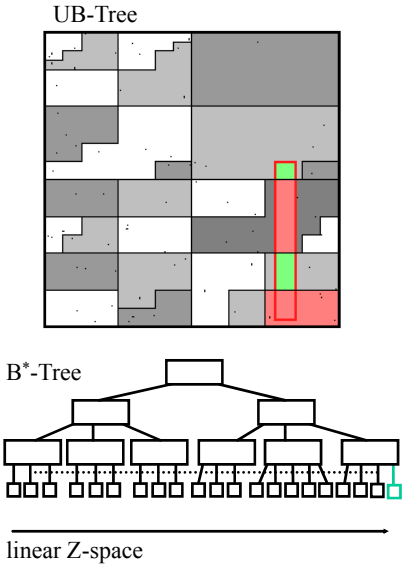
105

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
     cur = getNextZAddress(cur, start, end);
   }
}
```



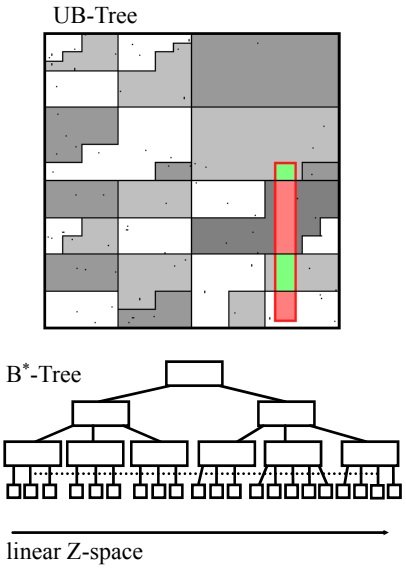
106

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



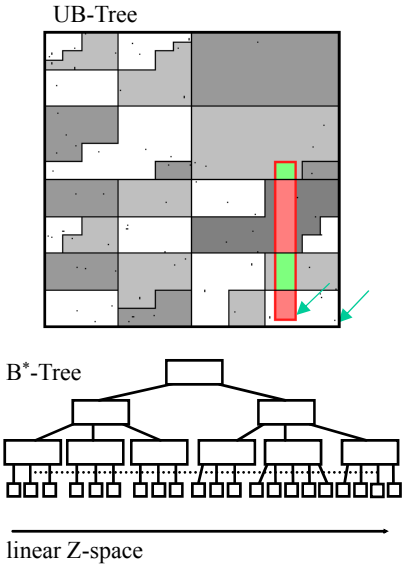
107

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
        cur = getNextZAddress(cur, start, end);
   }
}
```



108

```
rangeQuery(Tuple ql, Tuple qh)
{
   Zaddress start  = Z(ql);
   Zaddress cur    = start;
   Zaddress end    = Z(qh);
   Page page = {};

   while (1)
   {
     cur = getRegionSeparator(cur);
     page = getPage(cur);
     outputMatchingTuples(page, ql, qh);
     if ( cur >= end ) break;
       cur = getNextZAddress(cur, start, end);
   }
}
```

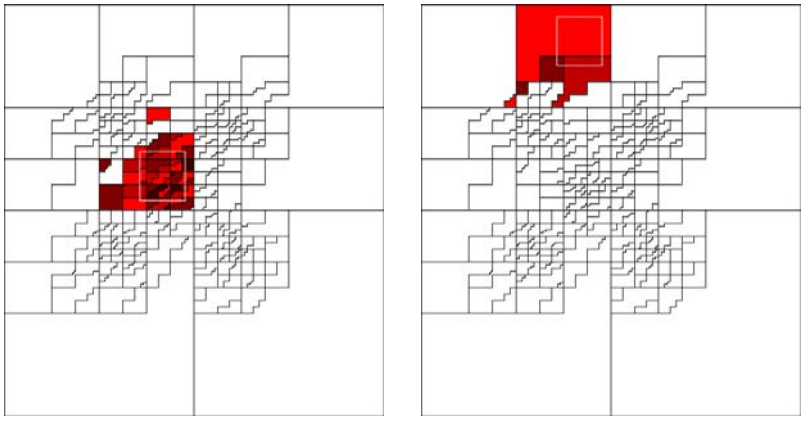


109

- Now the end condition is satisfied, since the largest address of the Z-region is larger than the end address of the query box. Thus, the algorithm can terminate now.

110

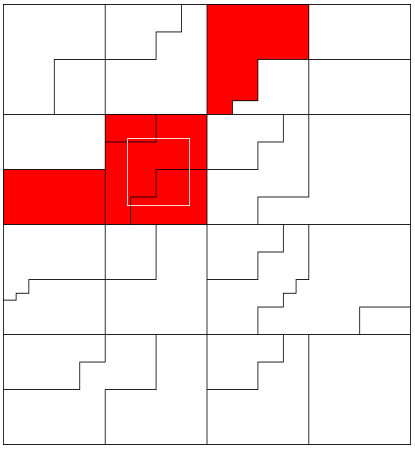
Range Queries and Data Distributions



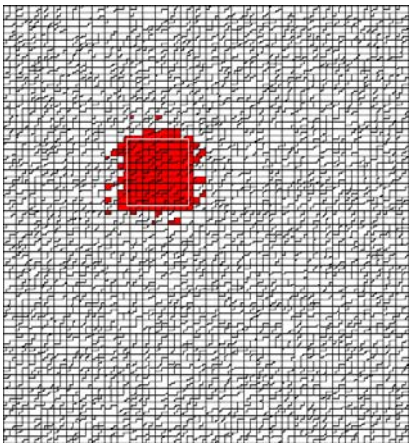
111

112

Growing Databases



1000 tuples



50 000 tuples

113

- This slide shows range queries in sparsely and densely populated parts of the universe. The data of this UB-Tree consists of 5 clusters (please note that small Z-regions denote parts of the space that are densely populated since each Z-region stores about the same number of tuples). Whereas a query in a densely populated part of the space (left side) retrieves a lot of Z-regions, a query in the sparsely populated part (right side) retrieves only 3 Z-regions. This means that the number of pages retrieved is correlated with the results set size, resulting in a very desirable response time behavior.

114

Summary UB-Trees

- 50% storage utilization, dynamic updates
- Efficient Z-address calculation (bit-interleaving)
- Logarithmic performance for the basic operations
- Efficient range query algorithm (bit-operations)
- Prototype UB/API above RDBMS (Oracle 8, Informix, DB2 UDB, TransBase, MS SQL 7.0) using ESQL/C

Patent application

115

116

Standard Query Pattern

SELECT * FROM table
WHERE (A_1 BETWEEN a_1 AND b_1) AND
(A_2 BETWEEN a_2 AND b_2) AND
.....
(A_n BETWEEN a_n AND b_n)
ORDER BY A_i, A_j, A_k, ...
(GROUP BY A_i, A_j, A_k, ...)

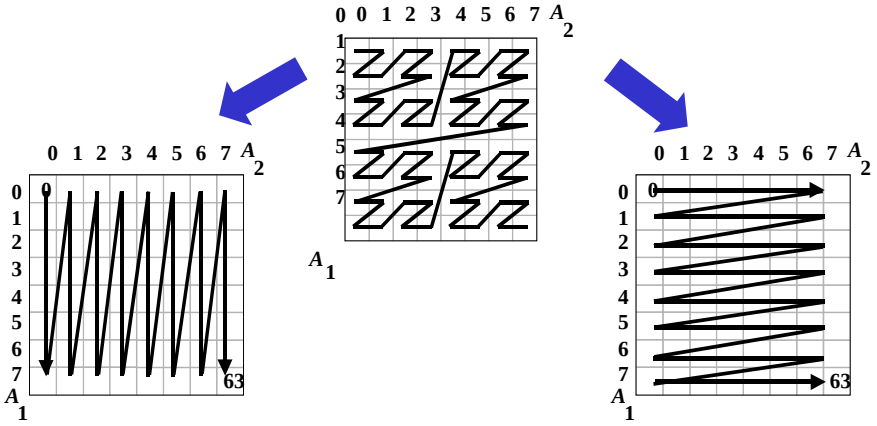
117

- In addition to range queries, sorting is a very important and frequent operation in relational databases. Sorting is not only used for ordering, but also provides a basis for efficient algorithms for duplicate elimination or join operations as well as for group by operations.

118

Z-Order/Tetris Order

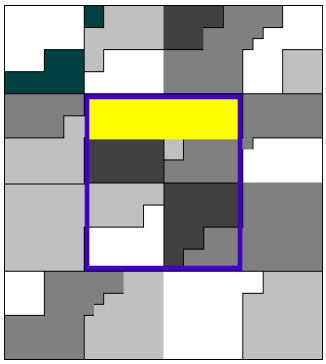
$$T_j(x) = x_j \circ Z(x_1,...,x_{j-1},x_{j+1},...,x_d)$$



- Sorting a Z-ordered space means to introduce a Tetris order, an ordering that extracts a single attribute out of a Z-ordered space. The Z-regions are read in this Tetris order as opposed to the Z-order that the range query uses to retrieve Z-regions. Please confer to [MZB99] for detailed information about Tetris order and the Tetris algorithm.

The Tetris Algorithm

sort direction



121

Summary Tetris

- Combines sort process and evaluation of multi-attribute restrictions in one processing step
- I/O-time linear w.r. to result set size
- temporary storage sublinear w.r. to result set size
- Sorting no longer a “blocking operation”

Patent application

122

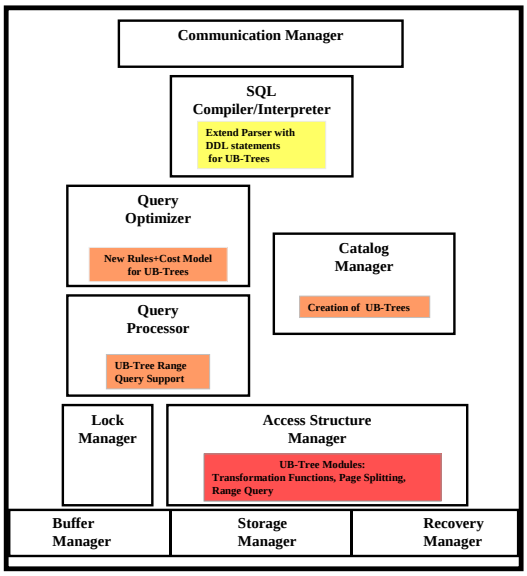
Integration Issues

- Starting point with TransBase:
  - clustering B*-Tree
  - appropriate data type for Z-values: variable bit strings
- Modifications to B*-Tree in TransBase:
  - support for computed keys:
    - Z-values are only stored in the index, not together with the tuples
    - tuples are stored in Z-order
  - generalization of splitting algorithm:
    - computed page separators for improved space partitioning

123

- In the MDA project between FORWISS, GfK, and TransAction Software the UB-Tree was integrated seamlessly into the RDBMS TransBase. The resulting product TransBase HyperCube is shipping since Systems 1999 and was awarded the 2001 IT-Prize by EUROCASE and the European Commission.
- The TransBase RDBMS already provided clustering B*-Trees and a bitstring datatype, which are a pre-requisite for a UB-Tree implementation. These implementations had to be slightly modified and enhanced in order to store the Z-addresses used by the UB-Tree.

124



- Minor extensions:
- Major extensions:
- New modules:
- **NO changes for:**
  - DML
  - Multi-user support, i.e., locking, logging facilities → handled by underlying B*-Tree

125

- This slide shows the extensions that had to be made to TransBase in order to incorporate the UB-Tree. It is important to note that UB-Trees rely on an underlying B-Tree; thus locking, caching and recovery did not need to be modified. Further details about the kernel integration can be found in [RMF+00].

126

Summary Integration

- Integration of the UB-Tree has been achieved within one year
- TransBase HyperCube is shipping since Systems 1999 and was awarded the 2001 IT-Prize by EUROCASE and the European Commission
- UB-Trees speed up relational DBMS for multidimensional applications like Geo-DB and data warehouse up to two orders of magnitude
- Speedup is even more dramatic for CD-ROM databases (archives)

127

Application Fields of the UB-Tree

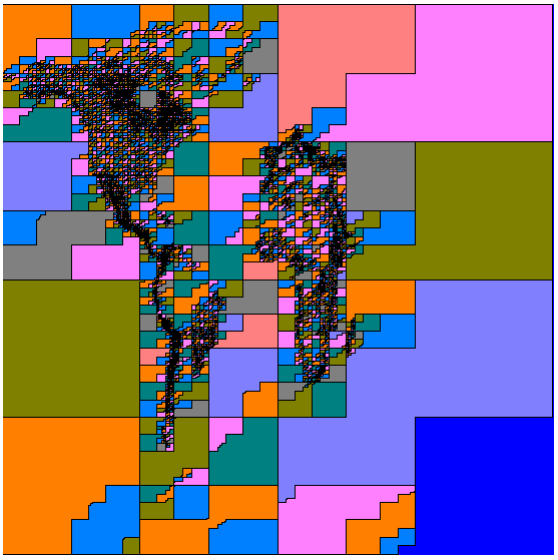
- Data Warehouses
  - Measurements with SAP BW Data
    - UB-Tree/API for Oracle
    - UB-Tree ***on top of*** Oracle outperforms conventional B-Tree and Bitmap indexes ***in*** Oracle!
  - Measurements with the GfK Data Warehouse
    - UB-Tree in TransBase HyperCube
    - significant performance increases (Factor of 10)
- Geographic Databases
- „Multidimensional Problems“
  - Archiving Systems, Lifecycle-Management, Data Mining, OLAP, OLTP, etc.

128

- The UB-Tree can be applied to any large record set that is queried and retrieved by multidimensional range queries.
- A typical application is data warehousing: Measurements with an UB-Tree API on top of Oracle compared to built-in Oracle indexes (including bitmap and IOT) showed speed-ups in favor of the UB-Tree, sometimes of more than two orders of magnitude. Similar results have been achieved with TransBase HyperCube compared to native TransBase indexes.
- The product TransBase is also used for GIS database, for instance for tracking the signal quality of the cells of a mobile phone network. Further application areas for TransBase HyperCube include archiving systems, data ming, and lifecycle management. Due to the good update and multi-user characteristics, UB-Trees can also be used to organize OLTP databases.
- Performance measurements details and applications of the UB-Tree and TransBase HyperCube can be found under http://mistral.in.tum.de and http://www.transaction.de

129

The UB-Tree



Z-region partitioning for a GIS database storing point data for Africa, Europe and the Americas.

130