ifis

Institut für Informationssysteme
Technische Universität Braunschweig

Data Warehousing & Mining Techniques

Wolf-Tilo Balke
Muhammad Usman
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de

Summary

- Last week:
  - Logical Model: Cubes, Dimensions, Hierarchies, Classification Levels
  - Physical Level
    - Relational: **Star-**, **Snowflake-schema**
    - Multidimensional (array based storage): **linearization**, problems e.g., **order of dimensions**, dense and sparse cubes
- This week:
  - Indexes

4. Indexes

4.1 Tree based indexes

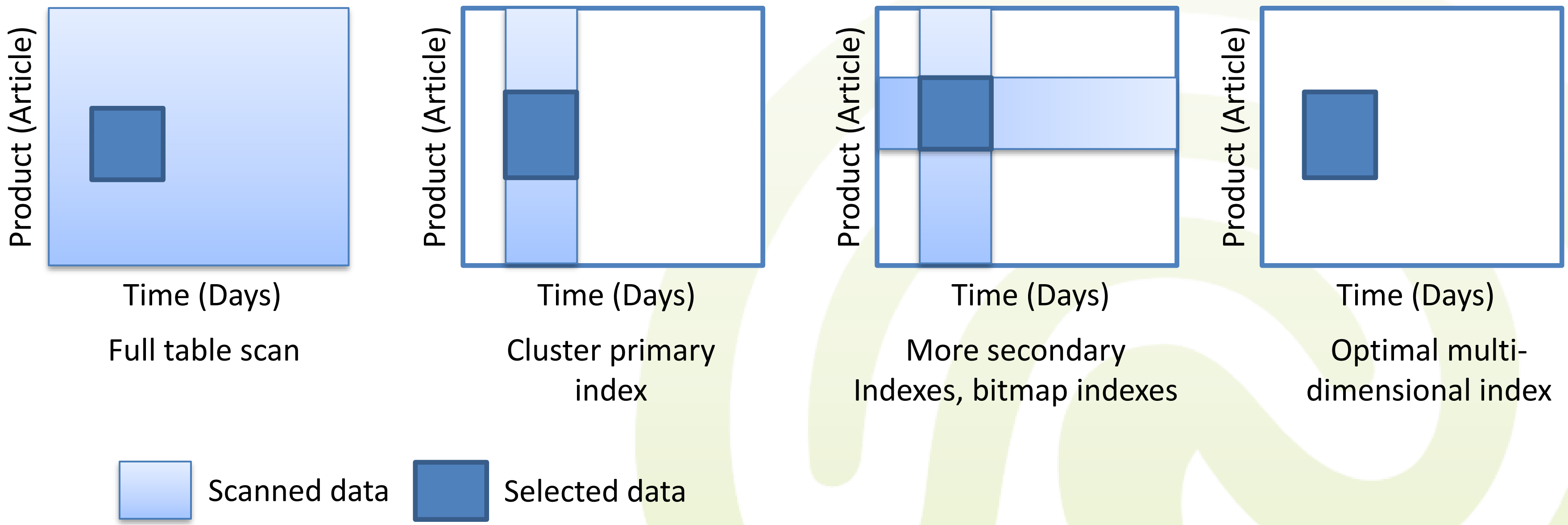
4.2 Bitmap indexes

4.0 Indexes

- Why index?
  - Consider a 100 GB table; at 100 MB/s read speed we need 17 minutes for a full table scan
  - Query for the number of Bosch S500 washing machines sold in Germany last month
    - Applying restrictions (product, location) the selectivity would be strongly reduced
      - If we have 30 locations, 10000 products and 24 months in the DW, the selectivity is $1/30 * 1/10000 * 1/24 = 0{,}00000014$
  - So…**we read 100 GB for 1,4KB of data**

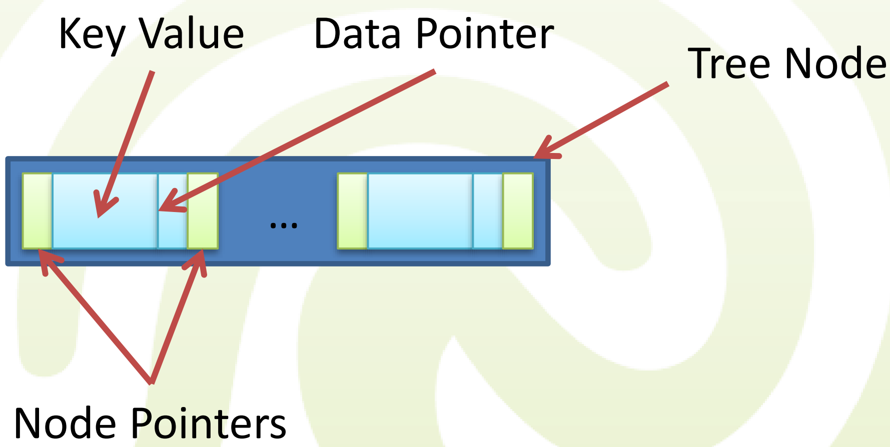
4.0 Indexes

- **Reduce the size of read pages to a minimum with indexes**



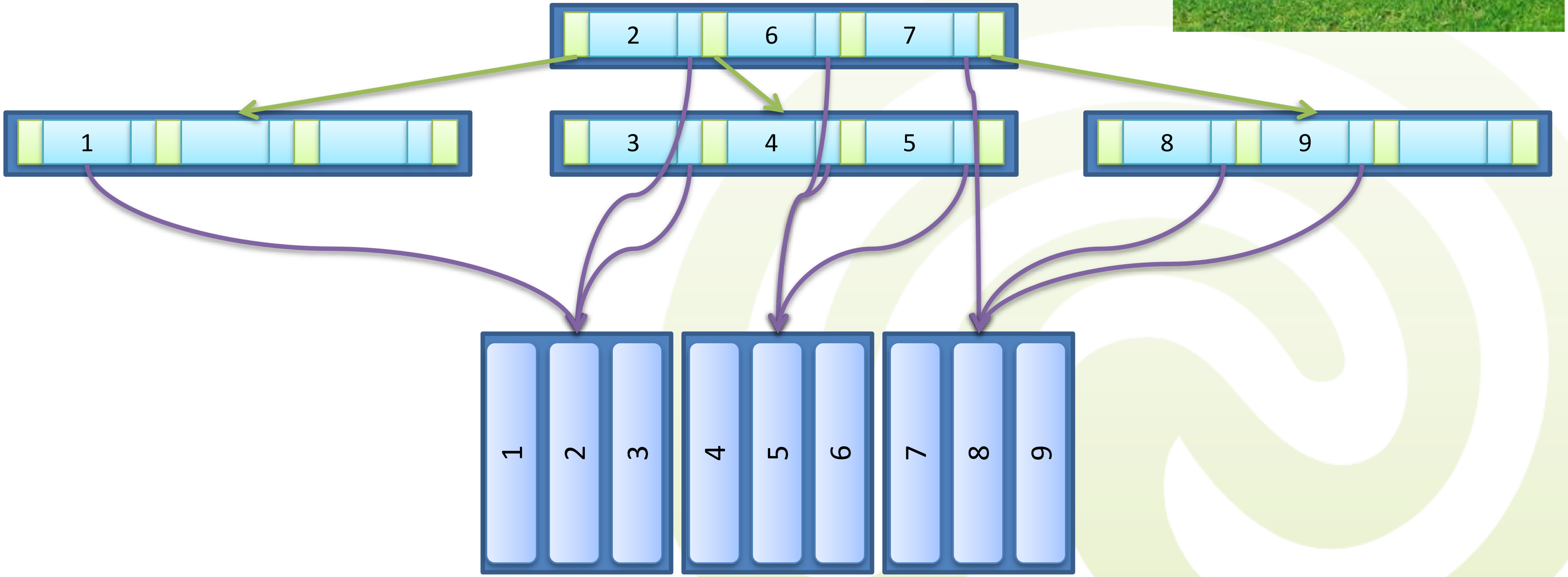
4.1 Tree Based Indexes

- In the beginning…there were **B-Trees**
  - Data structures for storing sorted data with amortized run times for insertion and deletion
  - Basic structure of a node



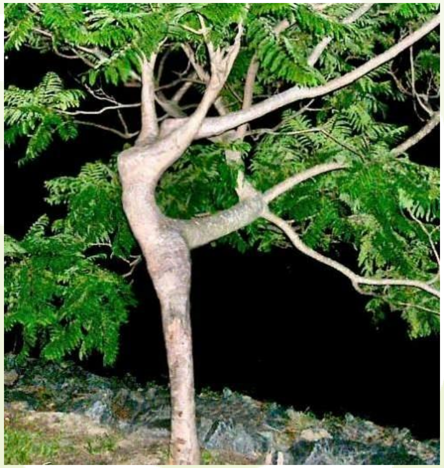
4.1 Tree Structures

- Search in database systems
  - **B-tree** structures allow exact search with logarithmic costs



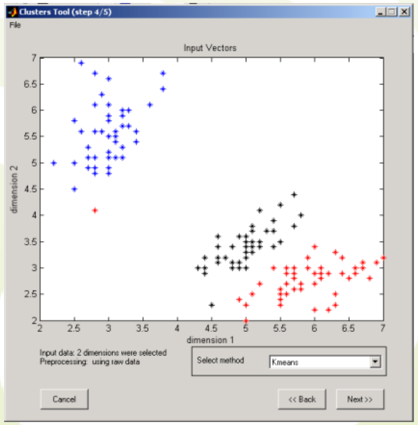
4.1 Tree Structures

- **Search in DWs**
  - The data is **multidimensional,** B-trees however, support only one-dimensional search
- Are there any possibilities to extend tree functionality for multidimensional data?



4.1 Tree Structures

- **The basic idea** of multidimensional trees
  - Describe the sets of points through **geometric regions**, which contain (clusters of) data points
  - Only clusters are considered for the actual search and not every individual point
  - Clusters can contain each other, resulting in a **hierarchical structure**

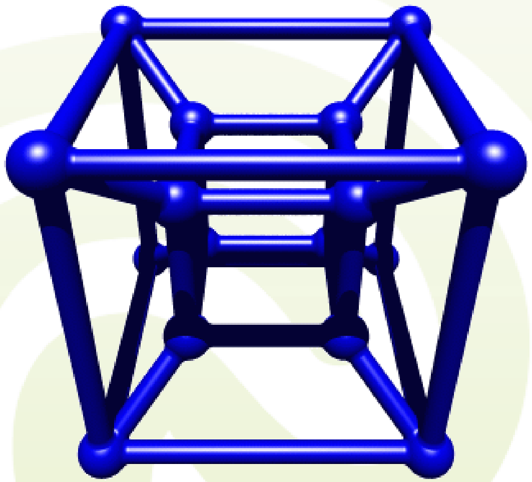


4.1 Tree Structures

- Differentiating criteria for tree structures:
  - **Cluster construction:**
    - Either completely fragmenting the space
    - Or grouping data locally
  - **Cluster overlap:**
    - Overlapping or
    - Disjoint
  - **Balance:**
    - Balanced or
    - Unbalanced

4.1 Tree Structures

- **Object storage:**
  - Objects can be in leaves and nodes, or
  - Objects are only in the leaves
- **Geometry:**
  - Hyper-sphere,
  - Hyper-cube,
  - ...



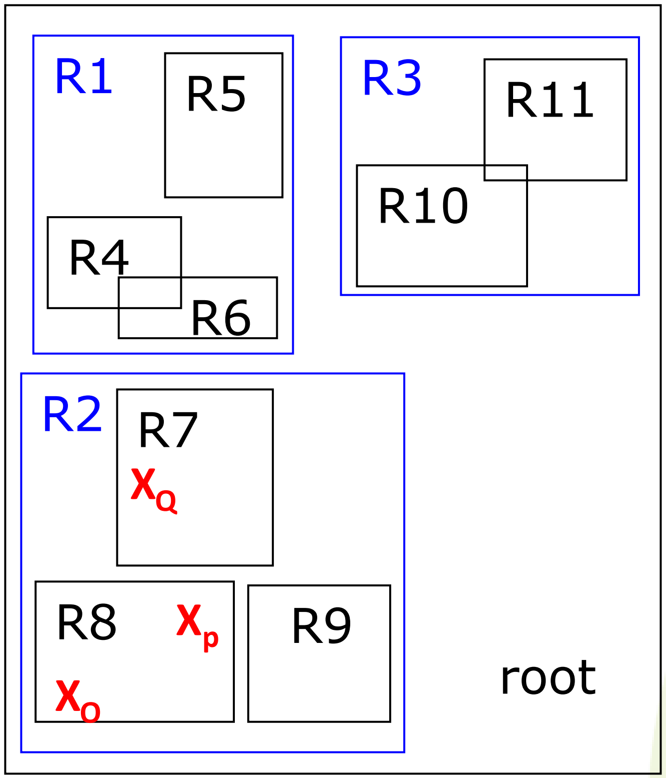
4.1 R-Trees

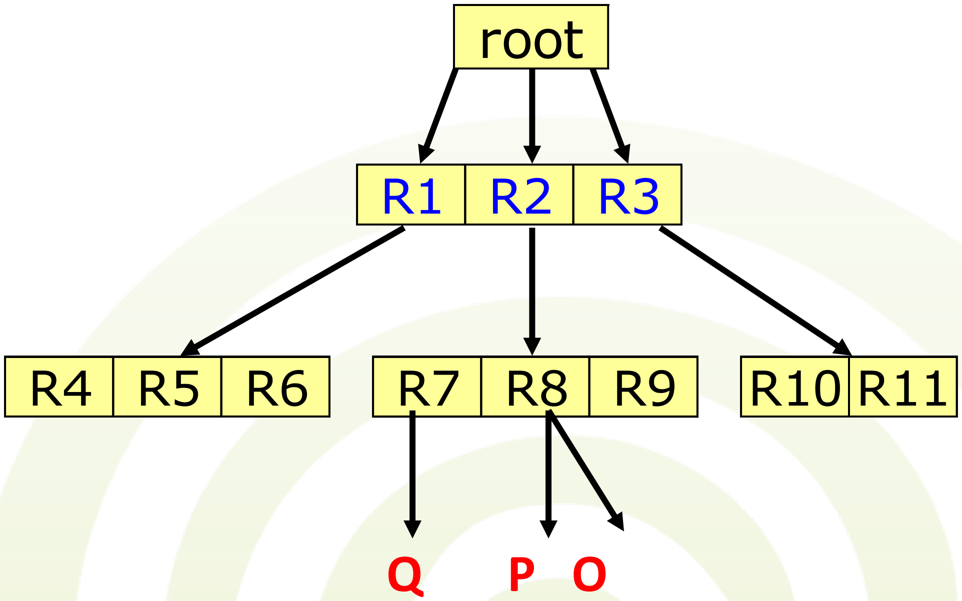
- The **R-tree** (Guttman, 1984) is a multi-dimensional extension of the classical B-trees
  - Frequently used for low-dimensional applications (up to about 10 dimensions), such as geographic information systems
- More scalable versions: R^+-Trees, R*-Trees and X-Trees
  - each up to 20 dimensions for uniformly distributed data

4.1 R-Tree Structure

- **Dynamic Index Structure** (insert, update and delete are possible)
- Data structure
  - **Data pages** are leaf nodes and store clustered point data and data objects
  - **Directory pages** are the internal nodes and store directory entries
  - Multidimensional data are structured with the help of **Minimum Bounding Rectangles** (MBRs)

4.1 R-Tree Example





4.1 R-Tree Characteristics

- **Local grouping** for clustering
- **Overlapping** clusters
  - the more clusters overlap the **less efficient** the index
- **Height-balanced** tree structure
  - therefore all the children of a node in the tree have about the same number of successors
- Objects are **stored** only in the leaves
  - Internal nodes are used for navigation
- MBRs are used as **geometry**

4.1 R-Tree Properties

- The root has **at least two children**
- Each internal node has between m and M children
- M and $m \geq M / 2$ are pre-defined parameters
- All the leaves in the tree are on the **same level**
- All leaves have between m and M index records
- Internal nodes: (I, child-pointer) where I is the **smallest bounding rectangle** that contains the rectangles of the child nodes
- Leaf nodes: (I, tuple-id) I is the smallest bounding rectangle that contains the data object (with ID tuple-id)

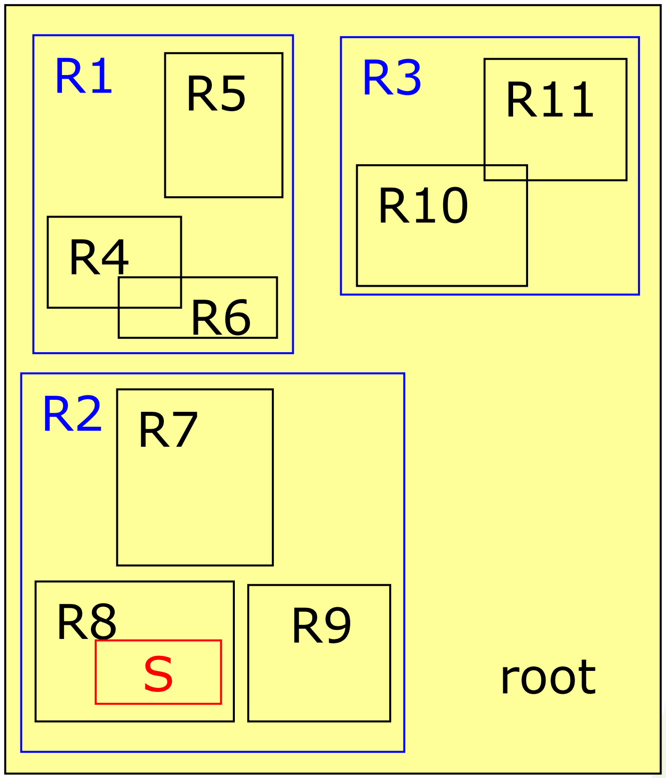
4.1 Operations on R-Trees

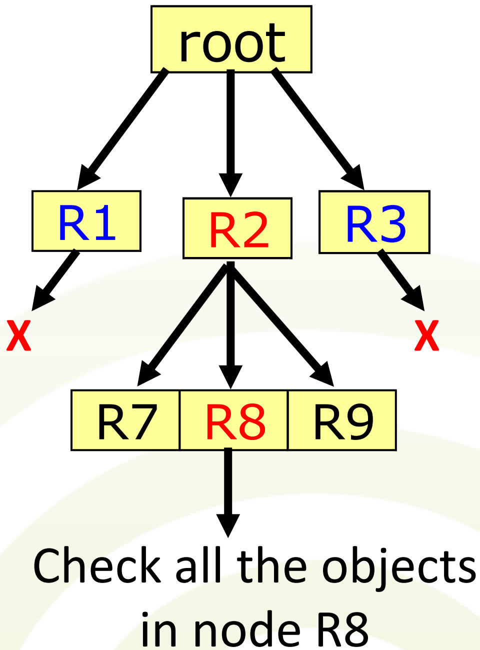
- The essential operations for the use and management of an R-tree are
  - Search
  - Insert
  - Updates
  - Delete
  - Splitting

4.1 Searching in R-Trees

- The tree is searched **recursively** from the root to the leaves
  - One path is selected
  - If the requested record has not been found in that sub-tree, the next path is traversed
- The path selection is arbitrary

4.1 Example





- Check only 7 nodes instead of 12

4.1 Searching in R-Trees

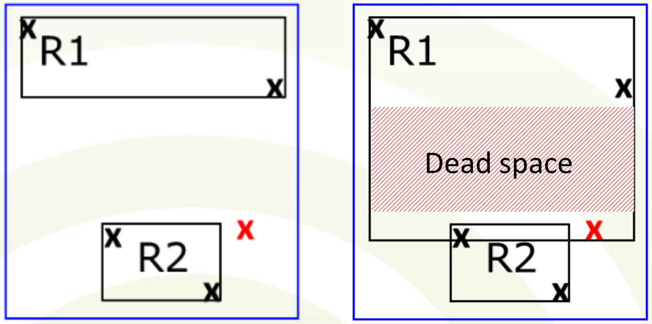
- **No guarantee** for good performance
- In the worst case, all paths must traversed (due to overlaps of the MBRs)
- Search algorithms try to exclude as many irrelevant regions as possible (“pruning”)

4.1 Search Algorithm

- All the index entries which intersect with the **search rectangle (query)** are traversed
  - **The search in internal nodes**
    - Check each MBR for intersection with query
    - For all intersecting MBRs continue the search with their children
  - **The search in leaf nodes**
    - Check all the data points to determine whether they intersect the query
    - Take all correct objects into the result set

4.1 Insert

- Choose the **best leaf** page for inserting the data
  – The beast leaf is the leaf that needs the **smallest volume growth** to include the new object
- Why smallest volume growth?
  – Enlarging R1 produces a large portion of unoccupied space in R1(dead space)



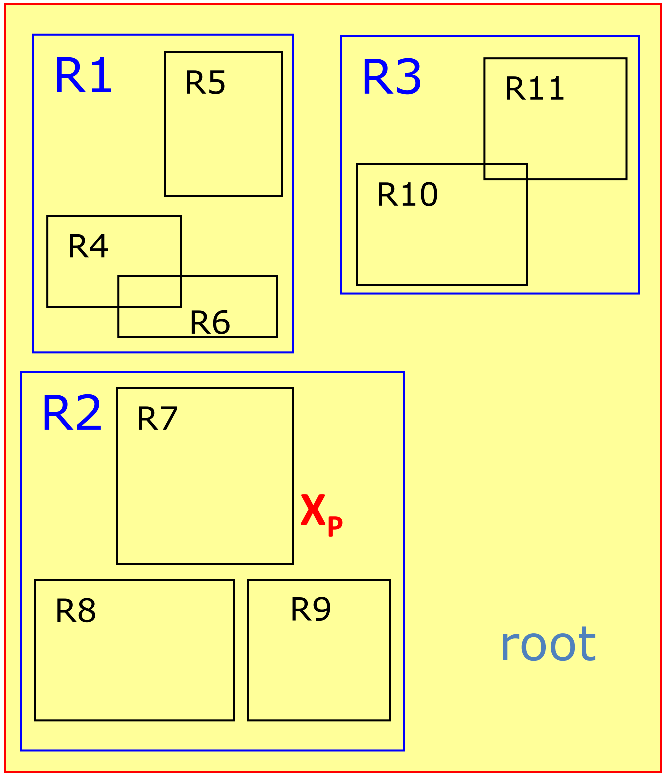
  – Since R1 occupies now large portions of space, the probability of a query intersecting with R1 is bigger, but the probability of hitting real data is low

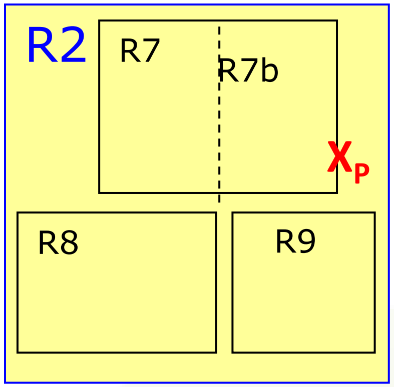
4.1 Heuristics

- An object is always inserted into nodes, where it produces the smallest increase in volume
  - If it falls in the **interior** of some MBR, no enlargement is needed
  - If there are several possible nodes, select the one with the **smallest overall** volume

4.1 Insert

- After the leaf is chosen, the object is inserted whenever there is still space (remember: number of objects in each node ≤ M)
  - Otherwise it is considered a case for **overflow handling** and the leaf node is divided
  - The interval (the bounding rectangle) of the parent node must be adapted to the new object
  - Divisions can cascade, if the parent was also full
  - If the root is reached by division, then create a new root whose children are the two split nodes of the old root

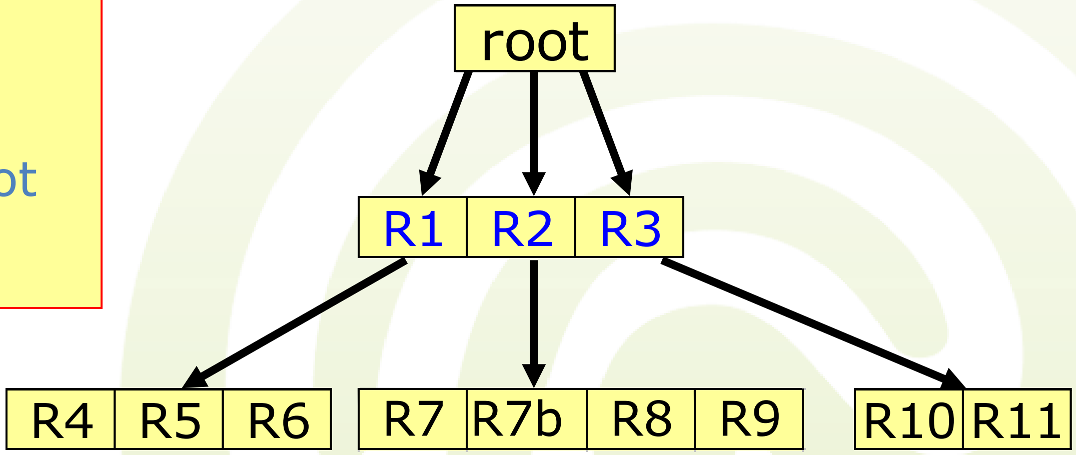




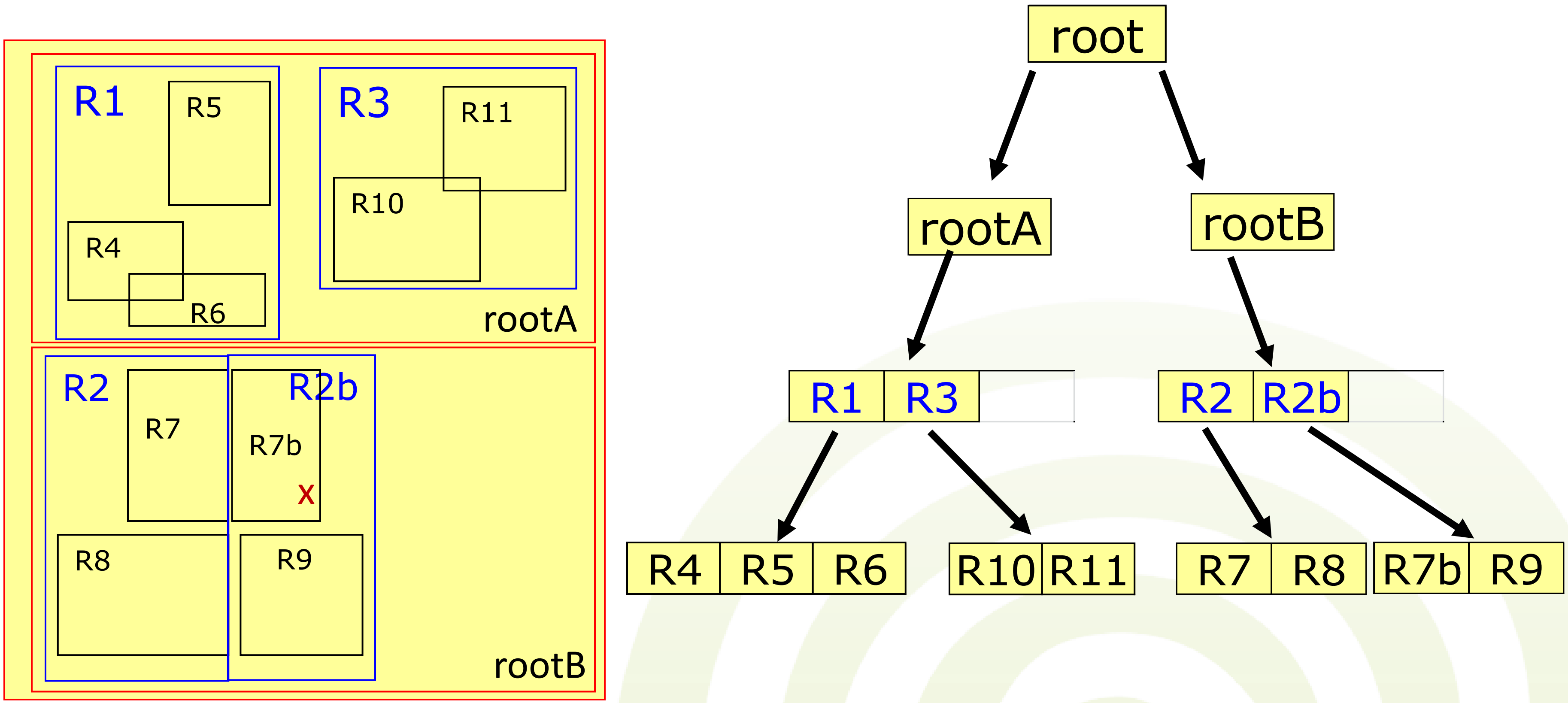
M = 3

R7 is full but is chosen as leaf for insertion

R2 is also full

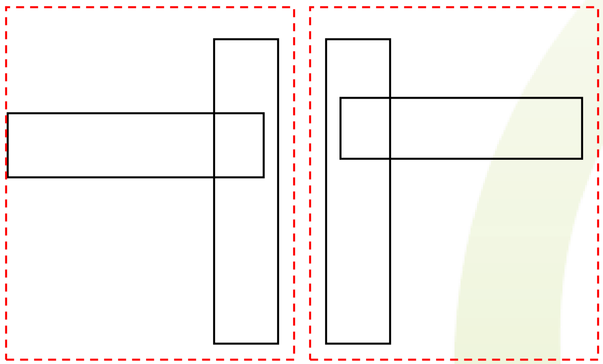


4.1 Insert with Overflow

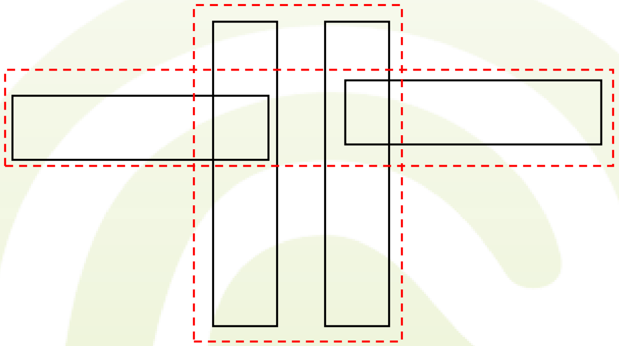


4.1 Splitting a Node

- The goal in splitting is that it should rarely be needed to **traverse both resulting nodes** on subsequent searches
  - Therefore use **small** MBRs. This leads to minimal **overlapping** with other MBRs



Bad split



Better Split

4.1 Overflow Problem

- Deciding on how exactly to perform the splits is **not trivial**
  - All objects of the old MBR can be divided in different ways on two new MBRs
  - The volume of both resulting MBRs should remain as small as possible
  - The naive approach of checking checks all splits and calculate the resulting volumes is not possible
- Two approaches
  - With **quadratic cost**
  - With **linear cost**

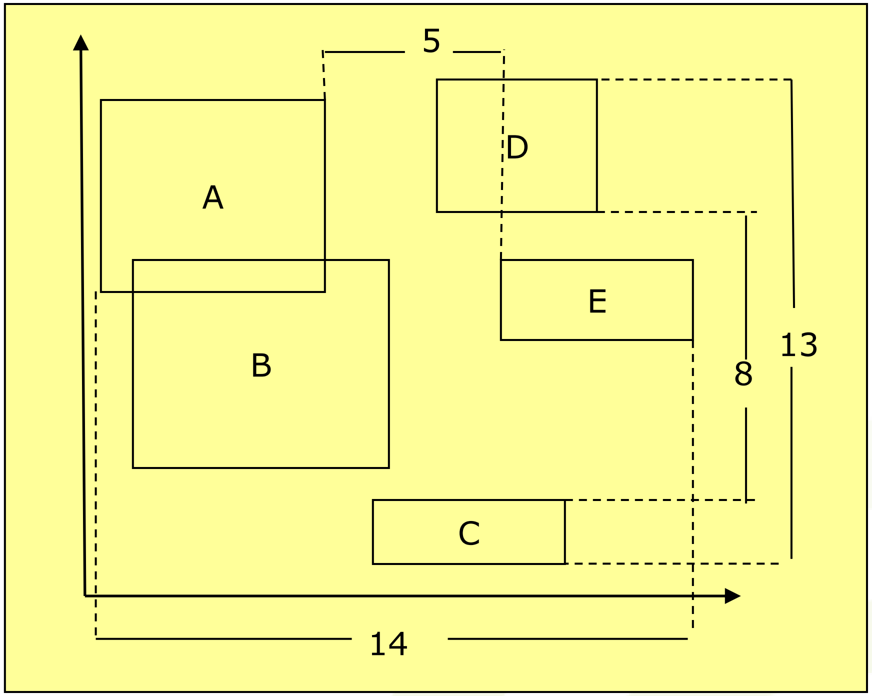
4.1 Overflow Problem

- Procedure with **quadratic cost**
  - For each 2 objects compute the necessary MBR and choose the pair with the largest MBR
  - Since these two objects should never occur in some MBR, they will be used as **starting points** for two new MBRs
  - For all other objects compute the **difference** of the necessary **volume increase** to insert them in either one of the starting points
  - Insert the object with the **smallest difference** in the corresponding MBR and compute the MBR again
  - Repeat this procedure for all non-allocated objects

4.1 Overflow Problem

- Procedure with **linear cost**
  - In each dimension:
    - Find the rectangle with the **highest minimum** coordinates, and the rectangle with the **smallest maximum** coordinates
    - Determine the distance between these two coordinates, and normalize it on the size of all the rectangles in this dimension
  - Determine the two starting points of the new MBRs as the two objects with the highest normalized distance

4.1 Example



- x-direction: select A and E, as $d_x = diff_x/max_x = 5 / 14$
- y-direction: select C and D, as $d_y = diff_y/max_y = 8 / 13$
- Since $d_x < d_y$, C and D are chosen for the split

4.1 Overflow Problem

- Iterativel insert remaining objects in the MBR with the smallest volume growth

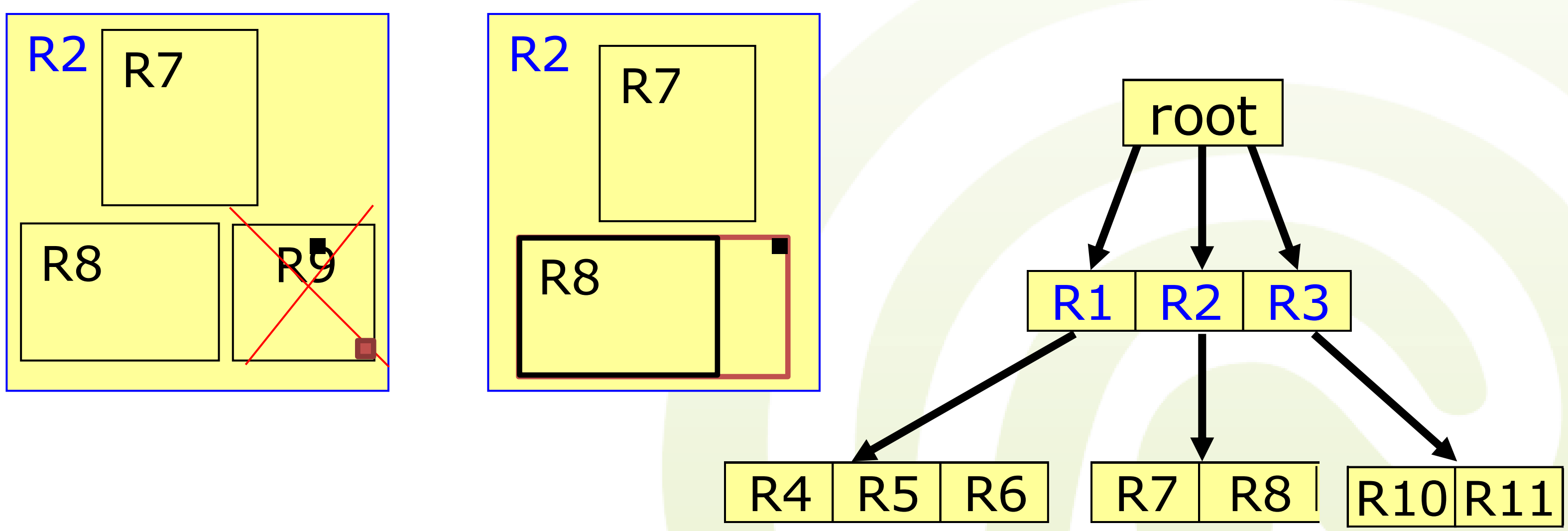
- The linear process is a simplification of the quadratic method
- It is usually sufficient providing similar quality of the split (minimal overlap of the resulting MBRs)

4.1 Delete

- **Procedure**
  - Search the leaf node with the object to delete
  - Delete the object
  - The tree is condensed if the resulting node has less than m objects
  - When **condensing**, a node is completely erased and the objects of the node which should have remained are **reinserted**
  - If the root remains with just one child, the child will become the new root

4.1 Example

- An object from R9 is deleted
  (1 object remains in R9, but m = 2)
  - Due to few objects R9 is deleted, and R2 is condensed

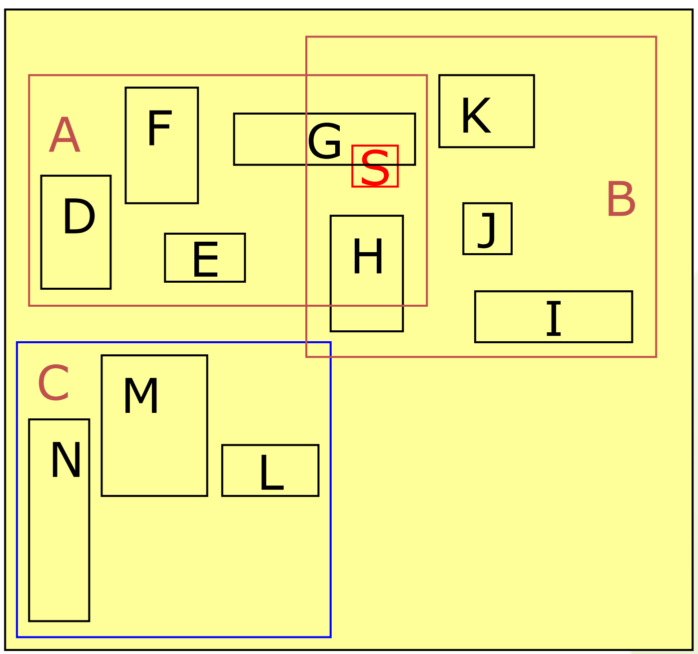


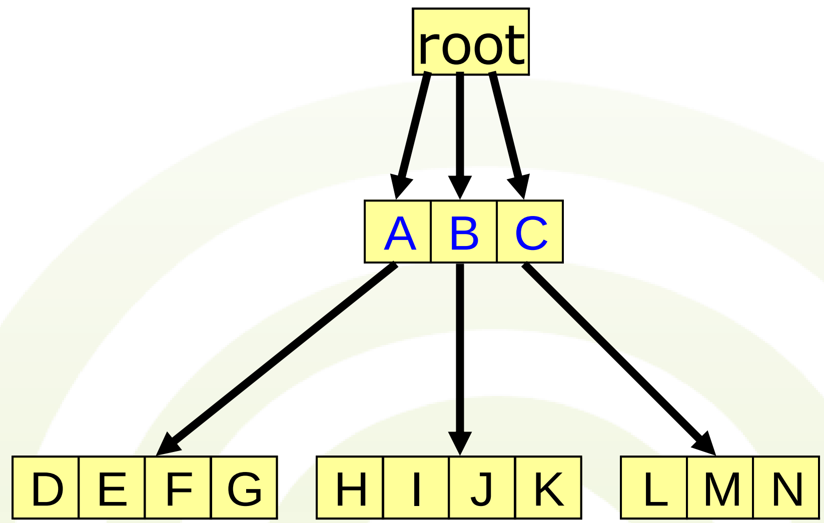
4.1 Update

- If a record is updated, its surrounding rectangle can change
- The index entry must then be deleted updated and then re-inserted

4.1 Block Access Cost

- The most efficient search in R-trees is performed when the **overlap** and the **dead** space are minimal





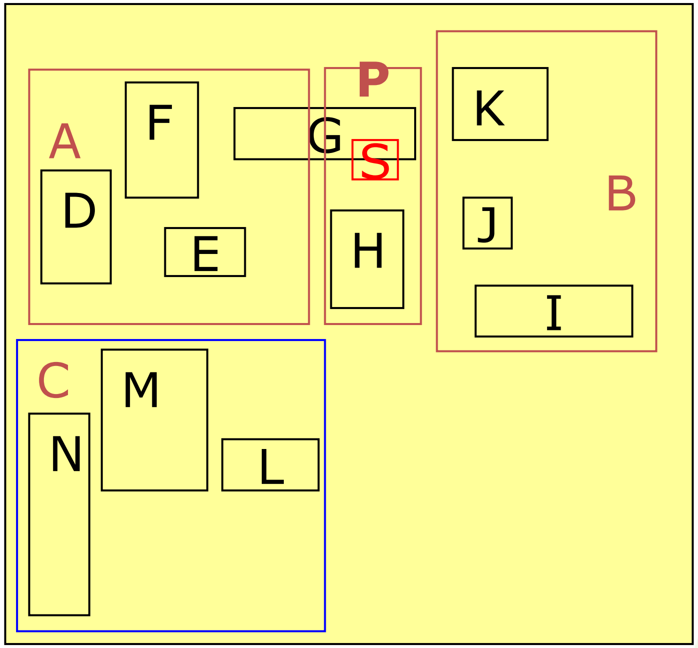
Avoiding overlapping is only possible if data points are known in advance

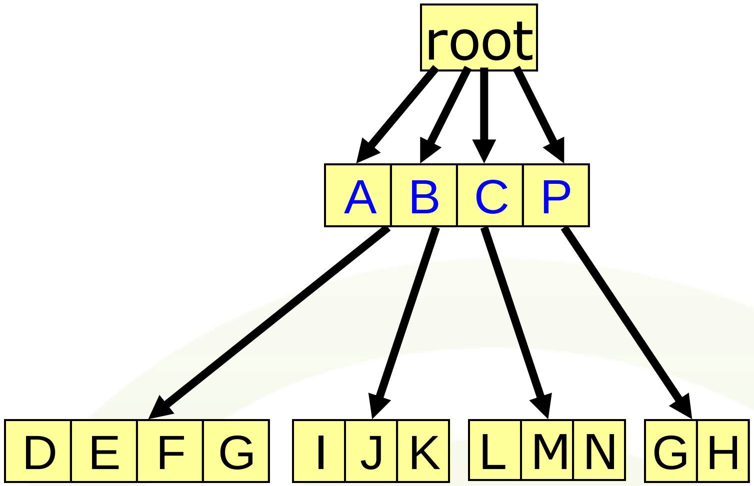
4.1 Improved Versions of R-Trees

- Why may R-trees be inefficient?
  - They **allow overlapping** between neighboring MBRs

- R^+-Trees (Sellis and others, 1987)
  - **Overlapping of neighboring MBRs is prohibited**
  - This may lead to identical leafs occurring more than once in the tree

4.1 R⁺-Trees





- **Overlaps are not permitted** (A and P)
- Data rectangles are divided and may be present (e.g., G) in **several leafs**

4.1 Performance

- The **main advantage** of R^+-trees is to improve the search performance
  - Especially for point queries, this may save up to 50% of access time
- **Drawback** is the low occupancy of nodes resulting from many splits
  - R^+-trees often **degenerate** with the increasing number of changes
  - Actually **scalability** is similar to R-trees

4.1 More Versions

- **R*- trees** and **X-trees** improve the performance of the R$^+$-trees (Kriegel and others, 1990/1996)
  - Improved split algorithm in R*-trees
  - “Extended nodes“ in X-trees allow sequential search of larger objects
  - Scalable up to 20 dimensions

4.1 UB-Trees

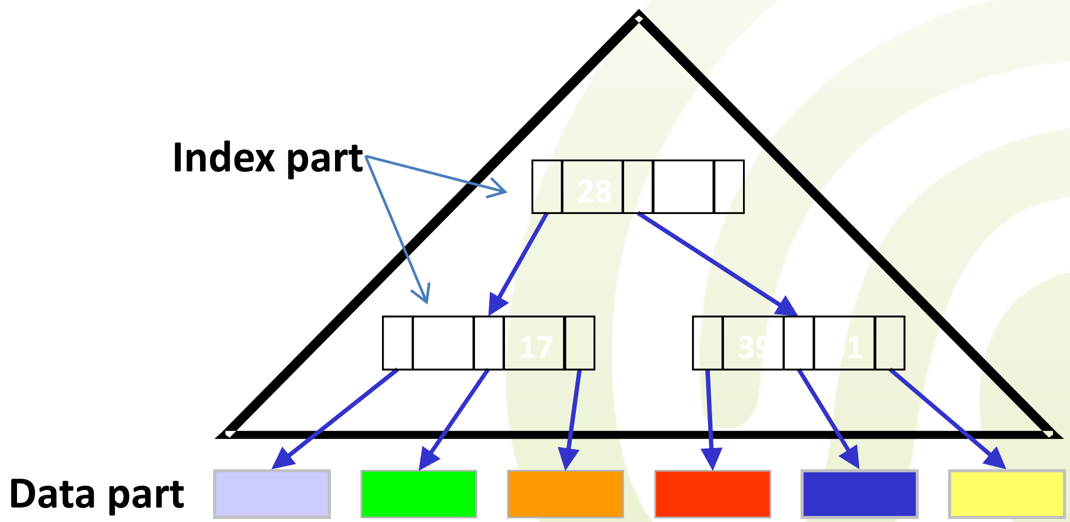
Detour

- B-Trees are great for indexing uni-dimensional data, but in the DW the data is stored **multidimensional**

- Idea: represent multidimensional data with just one dimension without information loss
  - How? Like in the case of MOLAP, with **linearization**

4.1 UB-Trees

- Universal B-Tree (UB-tree) are a combination of B^+-Tree and Z-curve as linearization function
  - Z-curve is used to map multidimensional points to one-dimensional values (Z-values)
  - Z-values are used as keys in B^+-Tree



4.1 Z-Curve Function

Detour

- **Z-Value address representation**
  - Calculate the z-values such that neighboring data is clustered together
  - Calculated through bit **interleaving** of the coordinates of the tuple
  - In order to localize a value with coordinates one has to perform de-interleaving

Tuple = 51, x = 4, y = 5

y \ x	0	1	2	3	4	5	6	7
0	1	2	5	6	17	18	21	22
1	3	4	7	8	19	20	23	24
2	9	10	13	14	25	26	29	30
3	11	12	15	16	27	28	31	32
4	33	34	37	38	49	50	53	54
5	35	36	39	40	51	52	55	56
6	41	42	45	46	57	58	61	62
7	43	44	47	48	59	60	63	64

For Z-value 51, we have
51 - offset = 50
50 in binary is 110010

Z-value = 110010

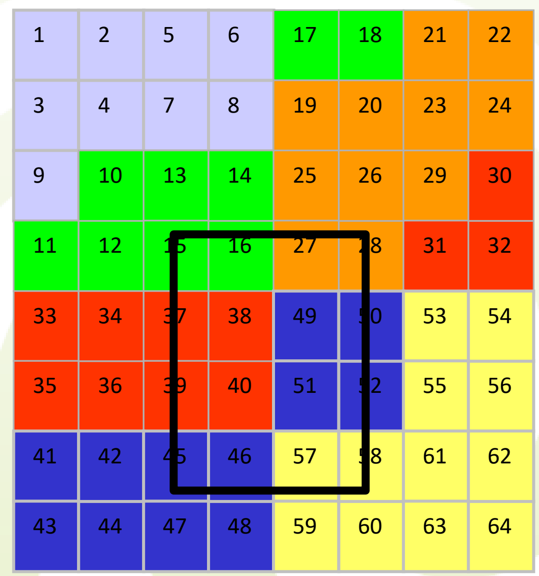
Y = 101 = 5 X = 100 = 4

4.1 UB-Trees

- **Z-Regions**
  - The space covered by an interval on the Z-Curve
    - E.g. [1:9], [10, 18], [19, 28]…
  - Each Z-Region maps exactly onto **one page** on **secondary storage**
    - i.e., to one leaf page of the B$^+$-Tree
  - This allows for very efficient processing of **multidimensional range queries**

1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

- Range queries (RQ) in UB-Trees
  - Each query can be specified by 2 coordinates
    - q_a (the upper left corner of the query rectangle)
    - q_b (the lower right corner of the query rectangle)
  - RQ-algorithm
    1. Starts with q_a and calculates its Z-Region
       1. Z-Region of q_a is [10:18]



4.1 UB-Trees

Detour

- Range queries (RQ) in UB-Trees
  2. The corresponding page is loaded and filtered with the query predicate
     - E.g. value 10 has after de-interleaving x=1 and y=2, which is outside the query rectangle
  3. After q_a all values on the Z-curve are de-interleaved and checked by their coordinates
     - The data is only accessed from the disk The next jump point on the Z-curve is 27
  4. Repeat steps 2 and 3 until the end-address of the last filtered region is bigger than q_b

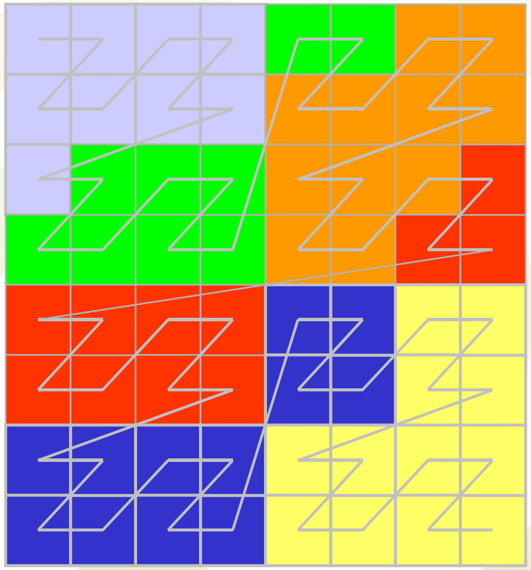
1	2	5	6	17	18	21	22
3	4	7	8	19	20	23	24
9	10	13	14	25	26	29	30
11	12	15	16	27	28	31	32
33	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

4.1 UB-Trees

Detour

- The critical part of the algorithm is **calculating the jump point** on the Z-curve which is inside the query rectangle
  - If this takes too long it eliminates the advantage obtained through optimized disk access
  - Calculating the jump point mostly involves
    - Performing **bit operations** and comparisons
    - 3 points: q_a, q_b and the current Z-Value

- Advantages of UB-Trees
  - B^+-Trees provide for high node filling degree (at least 50%)
    - Logarithmical complexity at **search, insert and delete**
  - The Z-curve provides for good performance for **range queries!**
    - Consecutive values on the Z-curve index similar data
    - Similarity by means of neighborhood



4.2 Bitmap Indexes

- Database indexes that use **bitmaps**
- Bitmaps (bit arrays) are array data structures that store individual bits
- Bitmap indexes are primarily intended for DW
  - Users query data rather than update it
- Bitmap indexes work well for data which has a small number of distinct values
  - E.g. gender data, or dimensions

Identifier	Gender	Bitmaps	
		F	M
1	Female	1	0
2	Male	0	1
3	Male	0	1
4	Unspecified	0	0
5	Female	1	0

4.2 Bitmap Indexes

- Let's assume fact table Sales and dimension Geography with granularity Shops

Nr	Shop_ID	Sum
1	1	150
2	2	65
3	3	160
4	2	45
5	1	350
6	2	80

Nr	Shop
1	Saturn
2	Real
3	P&C

  – A bitmap index on the fact table for dimension geography on attribute Shop looks like this

Value	Bitmap
P&C	001000
Real	010101
Saturn	100010

4.2 Bitmap Indexes

- A **bitmap index** for an attribute is:
  - A **collection of bit-vectors**
  - The number of bit-vectors represents **the number of distinct values** of the attribute in the relation
  - The **length of each bit-vector** is called the cardinality of the relation
  - The bit-vector for value 'Saturn' has 1 in position 5, if the 5th record has 'Saturn' in attribute Shop, and it has 0 otherwise
- Records are allocated permanent numbers
- There is a mapping between record numbers and record addresses

4.2 Bitmap Indexes

- Let's assume that…
  - There are n records in the table
  - Attribute A has m distinct values in the table
- The size of a bitmap index on attribute A is m*n
- Significant number of 0's if m is big, and of 1's if m is small
  - Opportunity to compress
    - Run Length Encoding (RLE)
    - Gzip (Lempel-Ziv, LZ)
    - Byte-Aligned Bitmap Compression (BBC): variable byte length encoding (Oracle patent)

4.2 Bitmap Indexes

Nr	Shop	Sum
1	Saturn	150
2	Real	65
3	P&C	160
4	Real	45
~~5~~	~~Saturn~~	~~350~~
6	Real	80

- Handling modification
  - Assume **record numbers are not changed**
- Deletion
  - **Tombstone** replaces deleted record (6 doesn't become 5!)
  - Corresponding bit is set to 0

Before

Value	Vector
P&C	001000
Real	010101
Saturn	100010

After

Value	Vector
P&C	001000
Real	010101
Saturn	100000

4.2 Bitmap Indexes

Nr	Shop	Sum
1	Saturn	150
2	Real	65
3	P&C	160
4	Real	45
5	Saturn	350
6	Real	80
7	REWE	23

- Inserted record is assigned the next record number
  - A bit of value 0 or 1 is appended to each bit vector
  - If new record contains a new value of the attribute, add one bit-vector
    - E.g., insert new record with REWE as shop

Before

Value	Vector
P&C	001000
Real	010101
Saturn	100010

After

Value	Vector
P&C	0010000
Real	0101010
Saturn	1000100
REWE	0000001

4.2 Bitmap Indexes

Nr	Shop	Sum
1	Saturn	150
2	REWE	65
3	P&C	160
4	Real	45
5	Saturn	350
6	Real	80

- Performing updates
  - Change the bit corresponding to the old value of the modified record to 0
  - Change the bit corresponding to the new value of the modified record to 1
  - If the new value is a new value of A, then insert a new bit-vector: e.g., replace Shop for record 2 to REWE

Before

Value	Vector
P&C	001000
Real	010101
Saturn	100010

After

Value	Vector
P&C	001000
Real	000101
Saturn	100010
REWE	010000

4.2 Bitmap Indexes

Nr	Shop	Sum
1	Saturn	150
2	Real	65
3	P&C	160
4	Real	45
5	Saturn	350
6	Real	80

- Performing selects
  - Basic AND, OR bit operations:
    - E.g., select the sums we have spent in either Saturn or P&C

Saturn	OR	P&C	=	Result
1		0		1
0		0		0
0		1		1
0		0		0
1		0		1
0		0		0

Value	Vector
P&C	001000
Real	010101
Saturn	100010

  - Bitmap indexes should be used when selectivity is **high**

4.2 Bitmap Indexes

- Advantages
  - Operations are efficient and easy to implement (directly supported by hardware)
- Disadvantages
  - For each new value of an attribute a new bitmap-vector is introduced
    - If we bitmap index an attribute like birthday (only day) we have 365 vectors: 365/8 bits ≈ 46 Bytes for a record, just for that
    - Solution to such problems is **multi-component** bitmaps
  - Not fit for range queries where many bitmap vectors have to be read
    - Solution: **range-encoded** bitmap indexes

4.2 Bitmap Indexes

- **Multi-component bitmap indexes**
  - Encoding using a different numeration system
    - E.g., for the month attribute, between 0 and 11 values can be encoded as $x = 4*y+z$, where $0 \leq y \leq 2$, and $0 \leq z \leq 3$, called <3,4> basis encoding

Month	Dec	Nov	Oct	Sep	Aug	Jul	Jun	Mai	Apr	Mar	Feb	Jan
M	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
5	0	0	0	0	0	0	1	0	0	0	0	0

    - $5 = 4*1+1$

X	Y			Z			
M	$A_{2,1}$	$A_{1,1}$	$A_{0,1}$	$A_{3,0}$	$A_{2,0}$	$A_{1,0}$	$A_{0,0}$
5	0	1	0	0	0	1	0

4.2 Multi-Component Bitmap Indexes

- Advantage **of multi-component bitmap indexes**
  - If we have 100 (0..99) different days to index we can use a multi-component bitmap index with basis of <10,10>
  - The storage is reduced from 100 to 20 bitmap-vectors (10 for y and 10 for z)
  - The read-access for a point (1 day out of 100) query needs however 2 read operations instead of just 1

4.2 Bitmap Indexes

- **Range-encoded bitmap indexes:** Persons born between March and August
  - For normal encoded bitmap indexes read 6 vectors

	Dec	Nov	Oct	Sep	Aug	Jul	Jun	Mai	Apr	Mar	Feb	Jan
Person	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	0	0	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	1	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0

  - Idea: set the bits of all bitmap vectors to 1 if they are higher or equal to the given value

4.2 Bitmap Indexes

- **Query:** Persons born between March and August
  - So persons which didn't exist in February, but existed in August!
  - **Just 2 vectors** read: $((\text{NOT } A_1) \text{ AND } A_7)$

	Dec	Nov	Oct	Sep	Aug	Jul	Jun	Mai	Apr	Mar	Feb	Jan
Person	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
1	1	1	1	1	1	1	1	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	0	0	0
3	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0

4.2 Range-encoded Bitmap Indexes

- If the query is limited only on one side, (e.g., persons born in or after March), **1 vector** is enough (NOT A_1)
- For point queries, **2 vector reads** are however necessary!
  - E.g., persons born in March: $((\text{NOT } A_1) \text{ AND } A_2)$

4.2 Advantages

- Bitmap indexes are great for indexing the dimensions
  - Fully indexing tables with a traditional trees can be expensive - the indexes can be several times larger than the data
  - Bitmap indexes are typically only a fraction of the size of the indexed data in the table.
- They…
  - reduced response time for large classes of ad hoc queries
  - bring dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory

Summary

- B-Trees are not fit for multidimensional data
- R-Trees
  - MBR as geometry to build multidimensional indexes
  - Operations: select, insert, overflow problem, node splitting , delete
  - Inefficient because they **allow overlapping** between neighboring MBRs
  - R^+-trees - **improve** the search performance

Summary

- UB-Trees
  - Reduce multidimensional data to one dimension in order to use B-Tree indexes
  - Z-Regions, Z-Curve, use the advantage of bit operations to make optimal jumps
- Bitmap indexes
  - Great for indexing tables with set-like attributes e.g., Gender: Male/Female
  - Operations are efficient and easy to implement (directly supported by hardware)
  - **Multi component** reduce the storage while **range encoded** allow for fast range queries

Next lecture

- Optimization
  - Partitioning
  - Joins
  - Materialized Views