Part VII

Index Structures for Data Warehouses

Index Structures for Data Warehouses

Motivation

- Fact tables in data warehouses can become very large, such that a full table scan becomes unadvantageous
- Example: Scan over 10 GB table at 10 MB/s $=$ ca. 17 minutes
- Queries just affect a relatively small part of the available data:
  - Depending on the restrictions on individual dimensions the result set of a query may affect a few percent or per mille (or even less) of the data
- Use of index structures to minimize the number of required page accessess

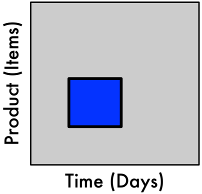
Classification of Index Structures

Classification of index structures

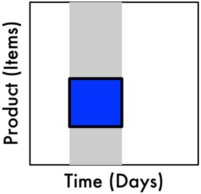
- Clustering: data that are likely to often processed together will also be stored physically close to each other
  - *Tuple Clustering*: Storing of tuples on the same physical page
  - *Page Clustering*: storing related pages close together in secondary storage (allows *prefetching*)
- Dimensionality: specify how many attributes (dimensions) of the underlying relation for calculation of the index key can be used
- Symmetry: If the performance is independent of the order of the index attributes, we have a symmetrical index structure, otherwise asymmetrical
- Tuple references: Type of tuple references within the index structure
- Dynamic behavior: Effort to update the index structure for insert, update and delete; (and possibly problem of "degeneration")

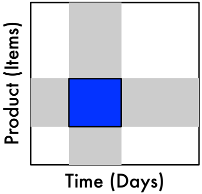
Comparison of Index Structures

Full Table Scan

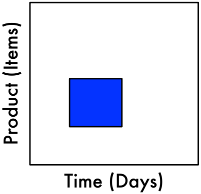


Clustered Primary Index





Several Secondary Indexes,
Bitmap Indexes

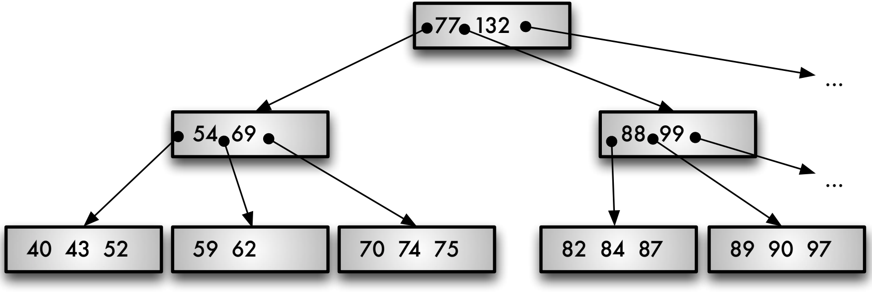


Multi-dimensional Index

B-Trees

One dimensional tree structures

- B-Tree [Bayer/McCreight 1972]



B-Tree

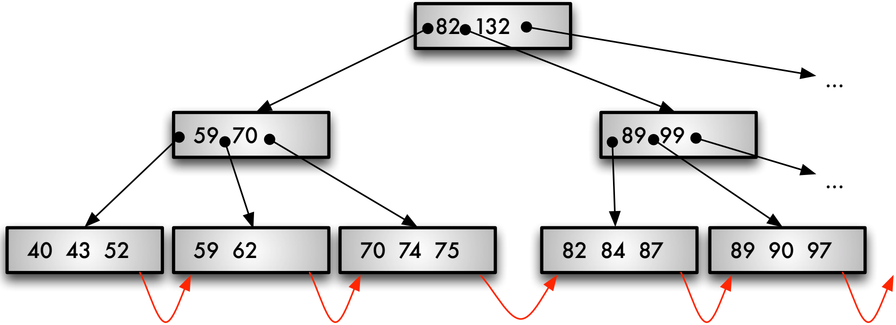
- Order of a B Tree: min. number of entries on the index pages (except for the root page)
- Definition: Index tree is a B Tree of order m, if
    - Each page contains at most $2m$ elements
    - Each page except for the root page contains at least m elements
    - Each page is either a leaf side without successors or it has $i+1$ successors (i: number of elements)
    - All leaf pages are on the same level

B-Tree: Properties

- n records in the main file
  $\rightarrow$ $log_m(n)$ page accesses from the root to the leaf
- Balance criterion leads to almost complete evenness
- Insert, delete, search with $O(log_m(n))$
- Memory space utilization: at least mindestens 50% (except for root)

B^+-Tree

- B^+ Tree (variant of the B-Tree): Tupels/TIDs only in the leaves; leaves are chained for a sequential (range) traversal



Properties of B- and B^+-Trees

- One-dimensional structure (index on an attribute)
- As a primary index tuples can be stored directly in the tree (allowing simple clustering, especially in the B^+-Tree)
- As a secondary index only TIDs are stored in the tree
- Balanced tree (Path from root to leaf has everywhere the same length); balancing requires more effort in reorganization in case of updates on the data
  - For data warehouses is of minor importance
- B^+–tree especially suited for range queries (by concatenation on leaf-level)

Use of B-/B^+-Trees

- B- and B^+-Trees: one-dimensional structures:
  - Only insufficient support of multidimensional queries
- Possible applications with multi-dimensional queries
  - For each attribute involved, there is a B- or B^+-Tree as a secondary index
  - Then for each attribute the set of TIDs is determined depending on whether they fulfill the query restriction
  - Now the intersection of the independently from each other determined TID sets is taken, the corresponding tuples form the query result

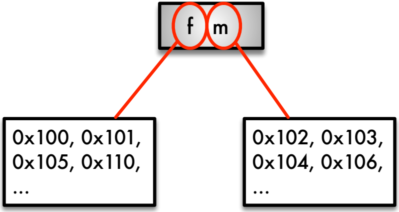
B^+-Tree: Conclusion

- Robust, generic data structure
- Independent of data type (only order required)
- Efficient update algorithms
- Compact
- "Working Horse" of all RDBMS
- Problems
  - Attributes of low cardinality $\rightarrow$ degenerated trees
  - Composed Indexes $\rightarrow$ order sensitive

Degenerated B-Trees

- Example:
  - Table `Customer`
  - Attributes: among others `Gender (m, f)`
  - Index

```
CREATE INDEX s_idx ON Customer(Gender)
```



Order Dependency

- Composed Index
  - Indexing of concatenated attribute values
  - Problem: Adjust the order of the query predicates
- Example:
  - Table: `Customer`
  - Attributes: `cclass, gender, profession`
  - Index:

```
CREATE INDEX csp_idx
   ON Customer(cclass, gender, profession)
```

  - Queries:

```
SELECT ...  FROM ...
   WHERE cclass=1 AND
      gender='m' AND profession='Lecturer'
```

B^+-Tree-Tricks: Oversized Index

- Query:

```
SELECT AVG(Age)
FROM Customer
WHERE cclass=1 AND gender='m'
   AND profession='Lecturer'
```

- Index usage
  - Searching the value "‘1||m||Lecturer"’
  - Access to a block of the relation `Customer` over TID for the value of `Age`
- Better:

```
CREATE INDEX csp_idx
ON Customer(cclass, gender, profession, age)
```

B^+-Tree-Tricks: Calculated Indexes

- Calculation of indexed values by using a function
- Example: Index over `Customer(name)`
  - QUery:

```
SELECT * FROM Customer
WHERE name="Müller" OR name="mueller"
   OR name="Mueller" ...
```

  - Index usage not possible
- Better

```
CREATE INDEX n_idx ON Customer(upper(name))
CREATE INDEX n_idx ON Customer(soundex(name))
```

Oracle9i: Special Features

- Index Skip Scan: uses composed indexes also when the is not in the condition
- Example:
  - Table `Customer`, Index on `(status, registration#)`
  - Index usable for a query with

```
...  WHERE registration# = 4245
```

- Searching the secondary index for all **`DISTINCT`** values of the first attribute
- Only useful if the first attribute has a low cardinality
- Infos: `http://otn.oracle.com/products/oracle9i/daily/apr22.html`

Oracle9i: Special Features (2)

- Index-organized tables
  - Tuples are additionaly stored in a B^+-Tree
  - No indirection via TID necessary
- User-defined indexes
  - Implementation of own index structures for user-defined data types
  - Transparent use
  - Specify own cost estimates

Bitmap Indexes

Bitmap Indexes

- Idea: Bit-Array to encode the tuple attribute value mapping
- Comparison with tree-based index structures:
  - Avoids degenerated B-trees
  - Insensitive towards higher number of dimensions
  - Easier support of queries, where only some (the indexed) dimensions are restricted
  - But for generally higher update costs
    - In data warehouses mostly unproblematic due to the majority of read-only accesses unproblematisch

Bitmap Index: Implementation

- Principle: Replacement of TIDs (rowid) for a key value in the B^+-Tree by bit list
- Node structure:

B: 010010...01	F: 101000...10	O: 000101...00

- Advantage: lower storage consumption
  - Example: 150.000 Tuples, 3 different key values, 4 Bytes for a TID
    - B^+-Tree: 600 KB
    - Bitmap: $3 \cdot 18750$ Byte $= 56$KB
- Disadvantage: higher update effort

Bitmap Index: Implementation (2)

- Definition in Oracle

```
CREATE BITMAP INDEX orderstatus_idx
ON Order(status);
```

- Use particularly for Star-Query transformation (Join between dimension- and fact table)
- Storage in compressed form
- Furthermore: Bitmap-based join indexes

Standard Bitmap Index

- Each Dimension is stored separately
- For each attribute value, an individual bitmap vector is created:
  - For each tuple, there is a corresponding bit that is set to 1 when the indexed attribute in the tuple contains the reference value of this bitmap vector
  - The number of resulting bitmap vectors per dimension corresponds to the number of different values that occur for the attribute

Standard Bitmap Index (2)

- Example: Attribute Gender
  - 2 Feature values (m/f)
  - 2 Bitmap vektors

PersId	Name	Gender	Bitmap-f	Bitmap-m
007	James Bond	M	0	1
008	Amelie Lux	F	1	0
010	Harald Schmidt	M	0	1
011	Heike Drechsler	F	1	0

Standard Bitmap Index (3)

- Selection of tuples can be achieved by linking bitmap vectors
- Example: Bitmap index over the attributes gender and month of birth
  - 2 Bitmap vectors B-f and B-m for gender
  - 12 Bitmap vectors B-1, ..., B-12 for the months, if all months occur
- Query: "all women born in march"
  - Calculation: B-f $\wedge$ B-3 (bitwise conjunctively linked)
  - Result: all tuples, at whose position in the bitmap vectors a 1 is given in the result

Multicomponent Bitmap Index

- For the Standard Bitmap Indexes many bitmap vectors are created for attributes with many feature values
- $< n, m >$ Multicomponent Bitmap Index allow to index $n \cdot m$ possible feature values by using $n + m$ bitmap vectors
- Each value $x (0 \leq x \leq n \cdot m - 1)$ can be represented by y and z:

$$x = n \cdot y + z \text{ mit } 0 \leq y \leq m - 1 \text{ und } 0 \leq z \leq n - 1$$

  - At maximum m bitmap vektors for y and n bitmap vektors for z
  - Storage overhead is reduced from $n \cdot m$ to $n + m$ vektors
  - However, point queries require reads of 2 bitmap vectors

Multicomponent Bitmap Index (2)

- Example: Two Component Bitmap Index
- For $M = 0..11$ for instance $x = 4 \cdot y + z$
- y values: B-2-1, B-1-1, B-0-1
- z values: B-3-0, B-2-0, B-1-0, B-0-0

x	y			z			
M	B-2-1	B-1-1	B-0-1	B-3-0	B-2-0	B-1-0	B-0-0
5	0	1	0	0	0	1	0
3	0	0	1	1	0	0	0
0	0	0	1	0	0	0	1
11	1	0	0	1	0	0	0

Example: ZIP-Codes

- Encoding of ZIP-Codes
- Values from 00000 to 99999
- Direct Implementation: 100.000 columns
- Two Component Bitmap Index (first 2 digits + 3 last digits): 1.100 columns
- Five Components: 50 columns
  - Suitable for range queries "‘`ZIP 39***`"’
- Binary encoded (to 2^{17}): 34 columns
  - Only for point queries!
- *Note: Encoding to the base 3 results in 33 columns....*

Range Encoded Bitmap Index

- Standard and Multicomponent Bitmap Indexes
  - Well suited for point queries
  - Inefficient for large ranges, because many bitmap vectors need to be linked
- Idea of range encoded bitmap indexes:

  In the bitmap vector a bit is set to 1 if the value of the attribute is smaller or equal to the given value.

- Range query $2 \leq attr \leq 7$ requires only 2 bitmap vectors: B-1 and B-7.
- Resulting bitmap vektor is $((\neg \texttt{B-1}) \wedge \texttt{B-7})$.
  - For range queries at maximim 2 bitmap vectors need to be read (only one for one-side restricted ranges)
  - For point queries exactly 2 bitmap vectors need to be read

Range Encoded Bitmap Index

Month M	Dec B-11	Nov B-10	Oct B-9	Sep B-8	Aug B-7	Jul B-6	Jun B-5	May B-4	Apr B-3	Mar B-2	Feb B-1	Jan B-0
Junq - 5	1	1	1	1	1	1	1	0	0	0	0	0
April - 3	1	1	1	1	1	1	1	1	1	0	0	0
Jan. - 0	1	1	1	1	1	1	1	1	1	1	1	1
Feb. - 1	1	1	1	1	1	1	1	1	1	1	1	0
April - 3	1	1	1	1	1	1	1	1	1	0	0	0
Dec. - 11	1	0	0	0	0	0	0	0	0	0	0	0
Aug. - 7	1	1	1	1	1	0	0	0	0	0	0	0
Sept. - 8	1	1	1	1	0	0	0	0	0	0	0	0

- Range query $February \leq Date \leq August$ requires B-0 and B-7.
- Resulting bitmap vector is $((\neg \mathtt{B-0}) \wedge \mathtt{B-7})$

Multicomponent Range Encoded Bitmap Index

- Combination of both techniques
- First, a multicomponent bitmap index is created
- On each built group of bitmap vectors the range encoding is applied
  - Due to the multicomponent technique less storage consumption (because of the smaller number of bitmap vectors)
  - The range encoding allows for efficient support of range queries
  - Because of the range encoding there is always one bitmap vector (component) dispensable in each group (representing the value $n-1$ and $m-1$, respectively, since there always all bits have to be set to 1 da dort immer alle Bits auf 1); hence only $n+m-2$ bitmap vectors are needed

Example MCREBMI

- Example Multicomponent Range Encoded Bitmap Index
  - $\texttt{B-0-1}' = \texttt{B-0-1}$
  - $\texttt{B-1-1}' = \texttt{B-1-1} \vee \texttt{B-0-1}'$
  - $\texttt{B-2-1}' = \texttt{B-2-1} \vee \texttt{B-1-1}' = \texttt{B-2-1} \vee \texttt{B-1-1} \vee \texttt{B-0-1} = 1$

M	B-1-1'	B-0-1'	B-2-0'	B-1-0'	B-0-0'
5	1	0	1	1	0
3	1	1	0	0	0
0	1	1	1	1	1
11	0	0	0	0	0

Interval Encoded Indexing

- Principle: each Bitmap vector represents a defined interval
- Example: Intervals $\mathtt{I-0} = [0,5], \mathtt{I-1} = [1,6], \mathtt{I-2} = [2,7], \mathtt{I-3} = [3,8], \mathtt{I-4} = [4,9], \mathtt{I-5} = [5,10]$

M	I-5	I-4	I-3	I-2	I-1	I-0
5	1	1	1	1	1	1
3	0	0	1	1	1	1
0	0	0	0	0	0	1
11	0	0	0	0	0	0
10	1	0	0	0	0	0

- Query: $(2 \leq \mathrm{M} \leq 8) \rightsquigarrow$ Evaluation of $\mathtt{I-2} \vee \mathtt{I-3}$

Selection of Index Structures

- In Data Warehouses normally multidimensional range queries
- Selection of index structure dependent on query profile
  - Is a specific attribute predominantly restricted?
    - ⇒ For asymmetrical index structures, the order of the index attributes shall be selected according to their frequency in the query profile
  - When no attribute can be identified as particularly important or many ad-hoc queries occur, symmetrical structures (secondary indexes, multidimensional indexes) are advantageous

Selection of Index Structures (2)

- Standard Bitmap Index
  - Fast, efficient implementation
  - Much storage space needed for a large number of feature values
- Multicomponent Bitmap Index
  - For point queries smalles number of read operations
- Range Encoded Bitmap Index
  - One-side restricted range queries
- Interval Encoded Bitmap Index
  - Two-side restricted range queries

Join Index

- Accelerating join computations by indexing attributes of "foreign" relations
- Precomputaion of the join and storing as an index structure
- Join/Grouping partially without access to foreign relation (e.g., dimension table) possible

Join Index: Example

Sales

V_ROWID	GeoID	TimeID	Sales	...
0x001	101	11	200	
0x002	101	11	210	
0x003	102	11	190	
0x004	102	11	195	
0x005	103	11	100	
0x006	103	11	95	

Geography

G_ROWID	GeoID	Branch	City	...
0x100	101	Allee-Center	Magdeburg	
0x101	102	Bördepark	Magdeburg	
0x102	103	Anger	Erfurt	
0x103	104	Erfurter Str.	Ilmenau	

```
CREATE INDEX joinidx ON Sales(Geography.GeoID)
USING Sales.GeoID = Geography.GeoID
```

0x100: { 0x001, 0x002, ...}
0x101: { 0x003, 0x004, ...}
0x103: { ...}

Bitmap Join Index

- So far:
  - Predicates for bitmap indexes not applied on foreign keys
  - Join has to be still executed
- Bitmap indexes only for Star Join optimization helpful
- Combination of
  - Bitmap Index and
  - Join Index

Bitmap Join Index with Oracle

- Definition

```
CREATE BITMAP INDEX join_idx
   ON Sales(Geography.GeoID)
      FROM Sales, Geography
      WHERE Sales.GeoID = Geography.GeoID
```

- Makes joins redundant (no access to `region` needed)
- Linking with other bitmap indexes on table `sales` possible

```
SELECT SUM(Sales.sales)
FROM Sales, Geography
WHERE Sales.GeoID = Geography.GeoID AND
   Geography.Stadt = 'Magdeburg'
```

- Infos: `Oracle Dokumentation 11g2 - Part E25789-01`

Indexed Views

- SQL Server 2008: Indexing Views
- Materialization of affected data
- Automated update for changes in the base data → materialized views

```
CREATE VIEW Sales2009 AS
SELECT City, Sales, S.TimeID, S.GeographyID
FROM Sales S, Time T, Geography G
WHERE S.TimeID = T.TimeID AND T.Year = 2009
   AND S.GeographyID = G.GeographyID;

CREATE UNIQUE CLUSTERED INDEX V2009_IDX
   ON Sales2009(TimeID, GeographyID);
```

Multidimensional Index Structures

Multidimensional Index Structures

- Hash-based Structures
  - Grid Files
  - Multidimensional Dynamic Hashing
- Tree-based Structures
  - kdB-Tree
  - R-Tree [Gutman 1984]
  - UB-Tree [Bayer 1996]

Grid File

- Multidimensional form of data organization
  - Combination of key transformation elements (Hash approaches) and Index Files (Tree approaches)
- Idea
  - Dimension refinement: Equal distribution of the multidimensional space of the chosen dimension through a complete cut (insertion of a hyperplane)

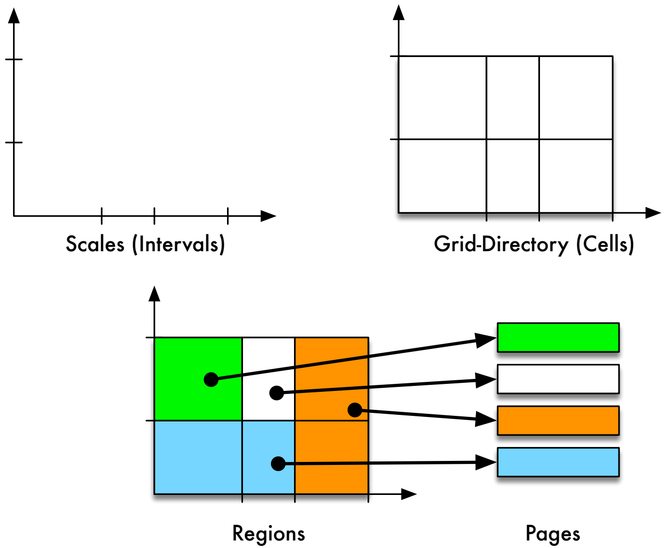
Grid File: Principles

- Separation of the data space in squares (search region: k-dimensional cuboids)
- Neighborhood preservation: Storage of similar objects on the same page
- Symmetrical treatment of all space dimensions: partial match queries
- Dynamic adjustment of the structure during inserts and deletes
- Principle of 2 disc accesses for exact match queries

Grid File: Structure

- Grid: k one dimensional fields (Scales), each scale represents an attribute
- Skales: consists of partitions from the mapped value space in intervals
- Grid Directory: consists of Grid cells, which disect the data space in squares
- Grid Cells: form a Grid region, that is assigned to exactly one record page
- Grid Region: k-dimensional, convex constuction (pairwise disjoint)

Grid File: Structure (2)



Grid File: Operations

- Beginning state: Cell = Region = one record page
- Page overflow
  - Dividing of pages
  - If there is just one cell in the region belonging to a page: Segmentation of the interval on a scale
  - If region consists of multiple cells: Division of those cells in separate regions
- Page underflow
  - Subsumption of two regions if the result is a convex region

Multidimensional Hashing – MDH

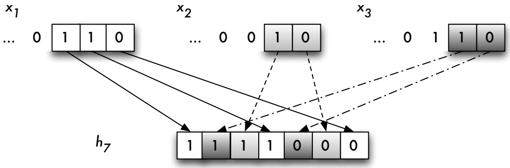
- Idea: Bit Interleaving
- Calculate in diverging order the address bits of the access attributes
- Example: two dimensions

	*0*0	*0*1	*1*0	*1*1
0*0*	0000	0001	0100	0101
0*1*	0010	0011	0110	0111
1*0*	1000	1001	1100	1101
1*1*	1010	1011	1110	1111

Idea MDH

- MDH builds upon *linear Hashing*
  - Dynamic Hash Method
  - Bit sequence prefixes address storage blocks
- Hash values are bit sequences, each having a beginning section serving as a current hash value
  - For binary numbers: often inverting the bit representation before prefix computation
- Compute one bit string per involved attribute
- Traverse beginning sections according to the principle of bit interleaving in a cyclic manner
- Hash values are composed of the surrounding bits of the individual values
- Family of Hash functions h_i for bit sequences of length i
  - Dynamic growth: go from i to $i+1$

MDH Illustration



- Clarifies composition of the hash function h_i fo three dimensions and the value $i = 7$
- At $i = 8$ a further bit of x_2 is used (more specific: of $h_{8_2}(x_2)$)
- MDH Complexity
  - Exact Match Queries: $O(1)$
  - Partial Match Queries, t of k attributes set, Effort $O(n^{1-\frac{t}{k}})$
  - Follows from the number of pages when certain bits are "unknown"
  - Special cases: $O(1)$ for $t = k$ and $O(n)$ for $t = 0$

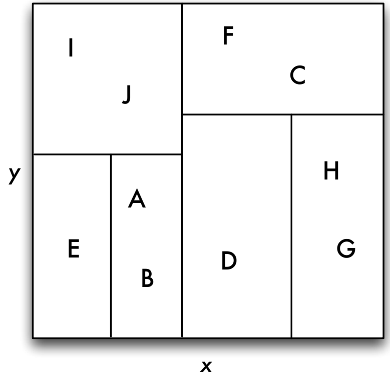
kdB-Tree

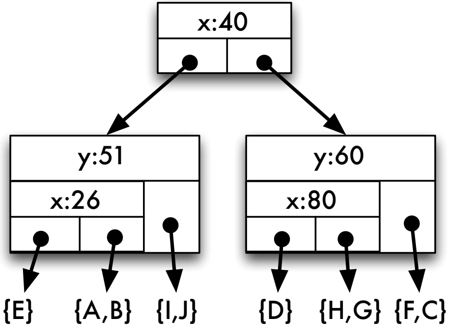
- k-dimensional index trees
  - kd-Tree: binary Tree for multidimensional basic structure; Main memory storage algorithms [Bentley 1975]
  - kdB-Tree: Combination of kd-Tree and B-Tree (higher branching degree)
  - kdB-Tree: Improving the kd-Tree
- Idea of the kdB-Tree
  - On each index page a subtree is presented, which branches after multiple subsequent attributes
  - Effort: exact-match $O(\log n)$, partial match better than $O(n)$

kdB-Tree: Structure

- kdB-Tree of type (b, t)
- Range pages (inner nodes): contain kd-Tree with max. b inner nodes
  - kd-Tree with with split elements and two pointers
  - Split element: Access attribute and value
  - Left pointer: smaller access attributes
  - Right pointer: larger access attributes
- Record pages (leaves): contain up to t tuples of the stored relation

kdB-Tree: Example



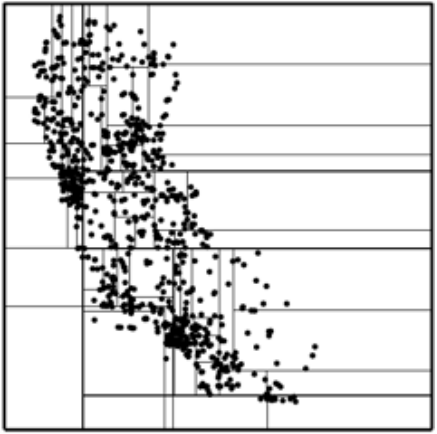


kdB-Tree: Split attributes

- Order
  - Cyclic
  - Consideration of selectivity: Access attribute with high selectivity ideally early and used more often than split element einsetzen
- Split attribute value
  - Finding a suitable average of the value space given distribution information

k[dD](B)-Tree: Conclusion

- Stores also bad distributed data
- Difficult to handle for more than three dimensions



R-Tree

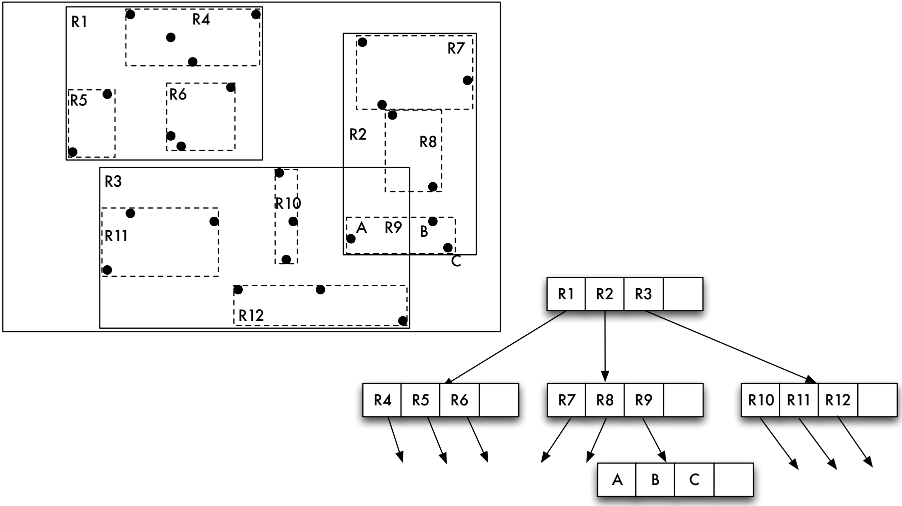
- Each node of the R-Tree stores max. m index entries
- Each node (except for the root) contains at least n entries
- d-dimensional R-Tree uses d-dimensional intervals (rectangles, squares) for indexing the data space
- Entry on leaf level: (I, tid) with I a d-dimensional interval and tid tuple identifier, that references the respective tuple
- Entry in inner nodes: (I, cp) with I the d-dimensional interval, that contains all intervals of the child node entries (minimum bounding box) and cp pointers to this child node (child pointer)

R-Tree (2)

Special Features (in comparison to the B^*-Tree)

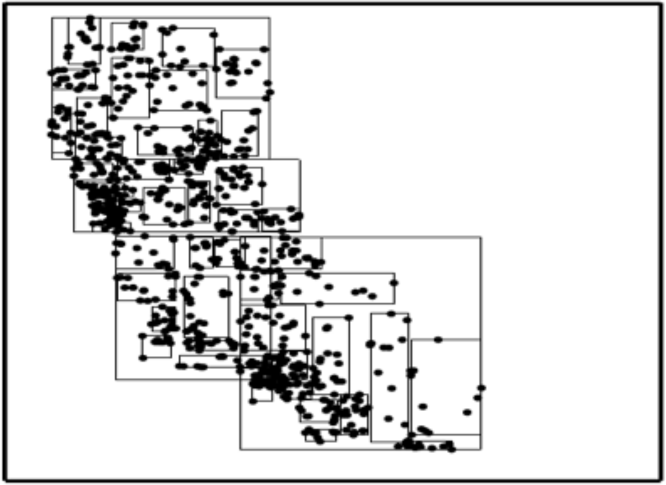
- Search: if different regions of the nodes on the same level overlap, multiple descendants may need to be traversed even for point queries
- Insert: attempt of finding an interval that does not need to be extended, otherwise the interval that has to be extended the least
- The deletion of data usually does not play a role in Data Warehouses; inserts only in big time intervals (but in turn often with many new tuples)
  - Efficient method important to built the R-Tree structure in a bottom-up manner

R-Tree: Example



R-Tree: Conclusion

- Better adjustment of the regions to the data



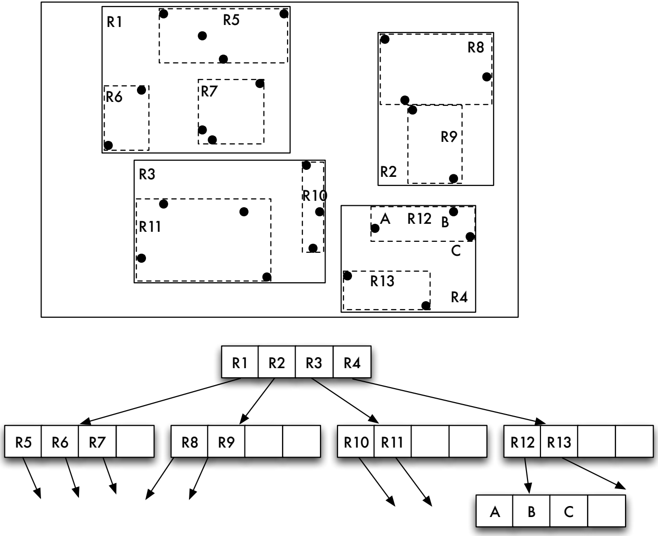
R+-Tree

- Basic idea: Forbid overlaps

$\rightarrow$ Adjustment of multiple nodes

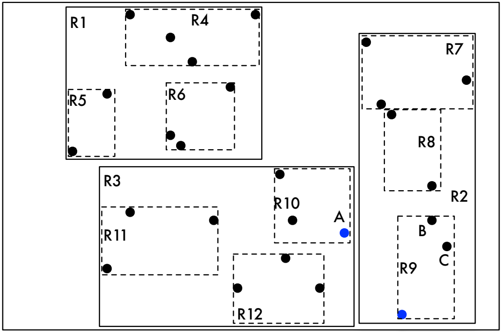
- Adjustment implies disections in smaller MBRs without previous overflow $\rightarrow$ nodes with few tuples (unused capacity) $\rightarrow$ many nodes (Degeneration)
- Clipping for storing geometric objects

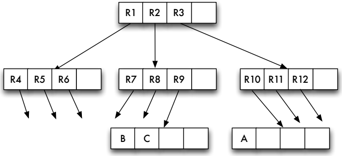
R^{+}-Baum: Example



R*-Tree

- Minimizing overlaps
- However, not forbidden!





R_a^*-Tree

- Support for OLAP
- Predominant for: Aggregation functions

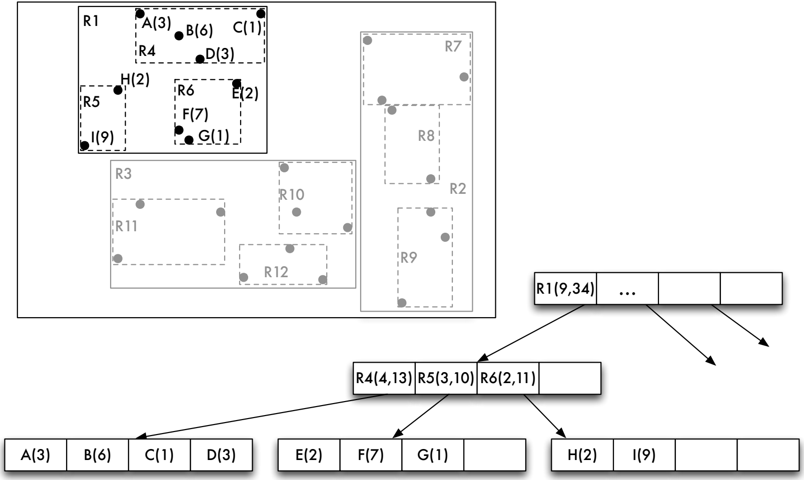
→ SUM, MAX, AVG, COUNT, ...

- R_a^*-Tree stores selected aggregated values of lower nodes in each inner node (materialized views)

→ *a* stands for *aggregated*

R_a^*-Tree II

R_a^*-Tree for COUNT and SUM

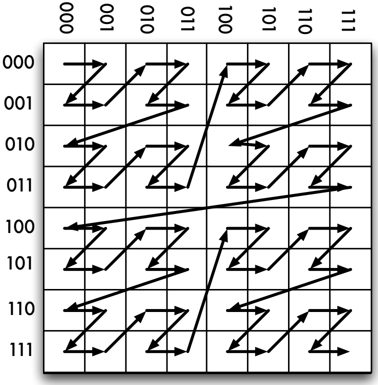


UB-Tree

- Data space is disected in disjoint subspaces using a space-filling curve (often the so-called Z-Curve)
- Each point of the attributes to be indexed within the multidimensional space is projected to a scalar value. the Z-Value
- Z-Values are used as keys in a standard B^+-Tree

UB-Tree (2)

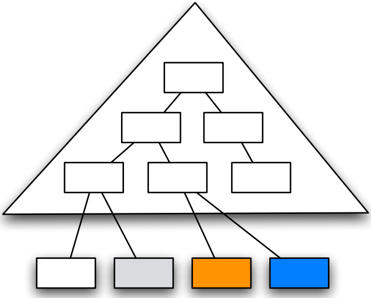
- Disection of a 2-dimensional space with the Z-Curve:

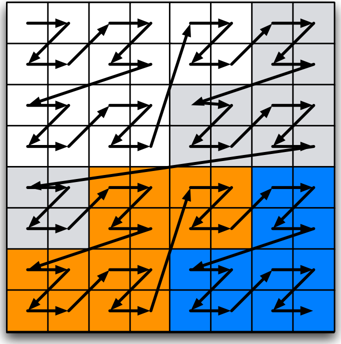


UB-Tree (3)

- Z-Values can be computed efficiently (in linearer time):
  - Per dimension the basic intervals are binary numbered;
  - By interleaving the bits, the respective Z-Value is obtained.
- Z-Region: is determined by an interval $[a, b]$ of Z-Values
  - Z-Regions of a UB-Tree are adjusted dynamically so that the objects within a Z-Region fit exactly within a page of a B^+-Tree
  - With that a B^+-Tree can be used as a basic structure

UB-Tree (4)





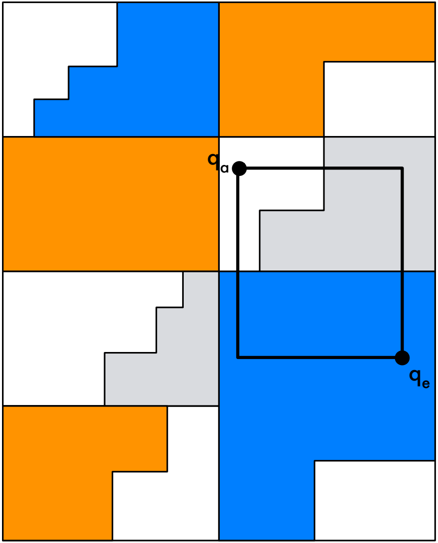
UB-Tree: Region Search (RQ-Algorithm)

- Each range query is determined by 2 tuples q_a and q_e which (visually) specify the left upper and right lower edge of the query region
  1. Begin with q_a and compute the respective Z-Region
  2. Load the respective page and apply the query predicate to all tuples within the page
  3. Compute the next region of the Z-Kurzve, that lies within the query region
  4. Repeat 2. and 3. until the end address of the edited Z-Region is bigger than q_e (and also contains the end point of the query region)

UB-Tree: Region Search

- Step 3. (Computation of the intersection points of the Z-Curve with the query region)
  - Appears critical at first glance;
  - In fact efficiently solved by using "a few" bit operations (and without disk accesses) in linear time (linear in the length of the Z-Values)

UB-Tree: Visual Region Search



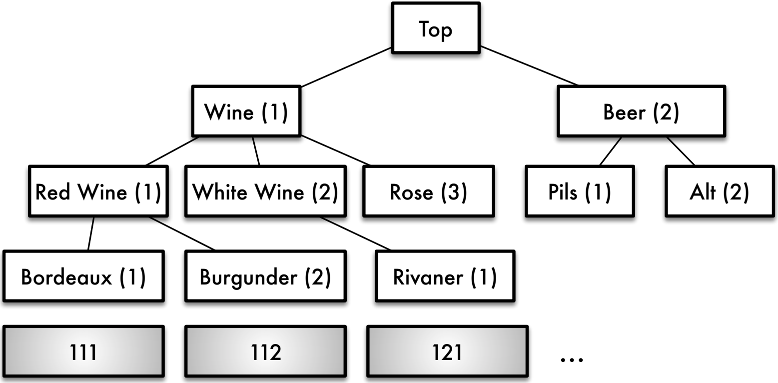
UB-Tree and MHC

- Extensions of the UB-Tree for Data Warehouses:
  - Multidimensional Hierarchical Clustering (MHC)
  - Supports hierarchically organized dimensions, so that all advantages of the UB-Tree remain

MHC: Principle

- Total order for hierarchy
- Assignment of a unique number to each leaf element of the hierarchy
- Elements of the same subtree contain grouped numbers (Clustering)
- Computation:
  - Each element of a hierarchy level contains a number (surrogate)
  - For leaf element: linking the surrogates together (binary representation) $\rightarrow$ multicomponent surrogate

MHC Example


Top
Wine (1)
Beer (2)
Red Wine (1)
White Wine (2)
Rose (3)
Pils (1)
Alt (2)
Bordeaux (1)
Burgunder (2)
Rivaner (1)
111
112
121
...

MHC Usage

- Multicomponent surrogate as
  - Key for tuples in the fact table
  - Index attribute for UB-Tree
- Example: Range query
  - Minimum und maximum multicomponent surrogate as interval for restriction

Summary

- Index structures allow improved multidimensional queries
- One dimensional index structures do not suffice
- B-Tree, Hash approaches and extensions are one dimensional
- Tree- and Hash approaches can be adjusted for multidimensionality
  - kdB-Tree, UB-Tree, R-Tree
  - Grid Files, MDH