Datenbanksysteme II: Implementation of Database Systems

Multidimensional Indexing

Material von
Prof. Johann Christoph Freytag
Prof. Kai-Uwe Sattler
Prof. Alfons Kemper, Dr. Eickler
Prof. Hector Garcia-Molina
Content of this Lecture

- Introduction to multidimensional indexing
- Partitioned Hashing
- Grid files
- kdb Trees
- R trees
Multidimensional Indexing

- Access methods so far
  - Support access on attribute(s) for
    - **Point query**: Attribute = const (Hashing and B-Tree)
    - **Range query**: const₁ ≤ Attribute ≤ const₂ (B-Tree)

- More complex queries
  - Point query on more than one attribute
    - Combined through AND (intersection) or OR (union)
  - Range query on more than one attribute
  - Queries for objects with size
    - "Sale" is a point in a multidimensional space
      - Time, location, product, …
    - Geometric objects have size: rectangle, cubes, polygons, …
Geometric Objects

• GIS (geographic information system) store rectangles
  \( \text{RECT} (X_1, Y_1, X_2, Y_2) \)
  (\(X_1, Y_1\) lower left corner and (\(X_2, Y_2\) upper right corner)

• Queries
  - Box query: All rectangles containing point (5,6)
    \[
    \text{SELECT} \ast \ \text{FROM} \ \text{RECT} \\
    \text{WHERE} \ \ X_1 \leq 5 \ \text{and} \ \ Y_1 \leq 6 \ \text{and} \\
    \ \ \ \ X_2 \geq 5 \ \text{and} \ \ Y_2 \geq 6
    \]
    • Similar to range query – all points within a given rectangle
  - Partial match query: Rectangles containing points with X=3
    \[
    \text{SELECT} \ast \ \text{FROM} \ \text{RECT} \\
    \text{WHERE} \ \ X_1 \leq 3 \ \text{and} \ X_2 \geq 3
    \]
  - All rectangles with non-empty intersection with rectangle Q
    \[
    \text{SELECT} \ast \ \text{FROM} \ \text{RECT} \\
    \text{WHERE} \ \ ...
    \]
Composite Indexes

- Imagine composite index on (X, Y)
- Box queries: efficiently supported
- Partial match query
  - All points/rectangles with X coordinate between ...
    - Efficiently supported
  - All points/rectangles with Y coordinate between ...
    - Not efficiently supported

<table>
<thead>
<tr>
<th>Point</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>2.5</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>4.5</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>4.7</td>
<td>6.5</td>
</tr>
<tr>
<td>P5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>P6</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>P7</td>
<td>8,3</td>
<td>3</td>
</tr>
</tbody>
</table>
Composite Index

• One index with two attributes \((X, Y)\)

• General
  - Prefix of attribute list in index must be present in query
  - The longer the prefix in a query, the more efficient

• Alternatives
  - Also index \((Y, X)\)
    • Combinatorial explosion for more than 2 attributes
  - Use independent indexes on each attribute
Independent Indexes

- One index per attribute

- Partial match query on one attribute: supported
- Partial match query on many attributes or box query
  - Compute TID lists for each attribute
  - Intersect
Example – Independent versus Composite Index

- Consider 3 dimensions of range 1,...,100
  - 1,000,000 points, uniformly (randomly) distributed
  - Index blocks holding 50 keys or records
  - Index on each attribute has height 4
- Find points with $40 \leq x \leq 50$, $40 \leq y \leq 50$, $40 \leq z \leq 50$
  - Using x-index, we generate list $|X| \sim 100.000$
  - Using y-index, we generate list $|Y| \sim 100.000$
  - Using z-index, we generate list $|Z| \sim 100.000$
  - For each index, we have $4 + 100.000/50 = 2004$ IO
    - TIDs are sorted in sequential blocks, each holding 50 TIDs
    - Hopefully, we can keep the three lists in main memory
    - Intersection yields app. 1,000 points with 6012 IO
      - Why 1000 points??
- Using composite index (X,Y,Z)
  - Number of indexed points doesn’t change
  - Key length increases – assume blocks hold only 30 (10) keys or records
  - Index has height 5 (6)
    - This is worst case – index blocks only 50% filled
  - Total: $5 \cdot 6 + 1000/30 (10) \sim 38$ IO (104)
Generalization

- Assume d dimensions, n records, k keys
- Assume query selectivity in each dimension s
- Independent indexes
  - Each independent index has height $\log_k(n)$
  - We find $s*n$ TIDs in $(s*n)/k$ blocks
  - All together: $C_1 = d*(\log_k(n)+(s*n)/k)$
- Composite index
  - Index has height $\log_r(n)$ for some $r<k$
  - We find $s^d*n$ TIDs in $(s^d*n)/r$ blocks
  - All together: $C_2 = \log_r(n)+(s^d*n)/r$
- For $d=5$, $n=1.000.000$, $k=50$, $r=30$, $s=0.1$
  - $C_1 = 20+10000$, $C_2 = 4+0$
    - On average, the result will already be empty
- For $d=8$, $n=1E9$, $k=50$, $r=10$, $s=0.01$
  - $C_1 = 48+1.600.000$, $C_2 = 9+0$
Conclusion 1

- **We want composite indexes**
  - Much less IO
  - Things get worse for bigger d
    - TID lists don’t fit into main memory – paging, more IO
    - Reading large TIDs list again and again is more work than scanning relation once
    - Linear scanning of relation might be faster
  - Advantage grows “exponentially” with number of dimensions and selectivity of predicates
  - Things get complicated if data is not uniformly distributed
    - Dependent attributes (age – weight, income, height, …)
    - Clustering of points
    - Histograms (later more)
Conclusion 2

• But: To support partial match queries, we need to index all combinations
  - Impossible

• Solution: Use **multidimensional indexes**
  - General: Improvement, but no solution
  - “Curse of dimensionality” still valid
    • Most md indexes somehow degrade for many dimensions
    • Trees difficult to balance, very bad space usage, excessive management cost, expensive insertions/deletions, …
  - Commercial databases use **bitmap indexes**
    • Very small memory footprint
  - Multidimensional indexes are used for geometric objects
    • Oracle has R tree in spatial extender
Multidimensional Indexes

• All dimensions are equally important
• Neighbors in space are (hopefully) stored on nearby blocks
  – That is the clue
  – Difficult to achieve
• Supported types of objects
  – With size
  – Without size (points)
• Supported queries
  – Exact match point queries
  – Partial match point queries
  – Box queries (range queries)
  – Nearest neighbor queries
    • In multidimensional space
Geographic Information Systems
Data Warehousing

- More dimensions: customer, logistic centre, supplier, company division, ...
Multimedia Databases

• Map object into feature vector
  - Here: Tumor images
  - Feature vector are derived from mathematical morphology
    • Can be computed in varying granularity (different length of vector vectors)
    • Filling / bordering picture using differently coarse brushes
• Compute nearest neighborhood queries in feature space
  - Filters away most false positives
  - Usually, final costly check on real object still necessary (but on few)
Content of this Lecture

- Introduction to multidimensional indexing
- Partitioned Hashing
- Grid files
- kdb Trees
- R trees
Partitioned Hashing

- Partitioned Hashing
  - Let \( A_1, A_2, \ldots, A_k \) be search keys
  - Define a hash function for each \( A_i \); interpret result as bit string
  - Global hash key: concatenation of the attribute bit strings
  - Definition
    - Let \( h(A_i) \) map each \( A_i \) into a integer with \( b_i \) bit
    - Let \( b = \sum b_i \) (length of global hash key in bits)
    - The global hash function
      \[
      h(v_1, v_2, \ldots, v_k) \rightarrow [0, \ldots, 2^b-1]
      \]
      is defined as
      \[
      h(v_1, v_2, \ldots, v_k) = h_1(v_1) \oplus h_2(v_2) \oplus \ldots \oplus h_k(v_k)
      \]
    - We need \( B = 2^b \) buckets
      - Static address space - dynamic structures later
Example

• We want to store points
  - (3,6), (6,7), (1,1), (3,1), (5,6), (4,3), (5,0), (6,1), (0,4), (7,2)
• Let hash function $h_1$, $h_2$ be
  \[
  h_i (v_i) = \begin{cases} 
  0 & \text{if } 0 \leq v_i \leq 3 \\
  1 & \text{otherwise}
  \end{cases}
  \]
• Thus, there are 4 buckets with address 00, 01, 10, 11

\[
\begin{array}{|c|c|}
\hline
0 & 1 \\
\hline
0 & (1,1) & (3,1) & (3,6) & (0,4) \\
1 & (4,3) & (5,0) & (6,1) & (6,7) & (5,6) \\
\hline
\end{array}
\]
Queries with Partitioned Hashing

• Exact point queries
  - Direct access to bucket possible

• Partial match queries
  - Only parts of the global hash key are determined
  - Use those as filter; scan all buckets passing the filter
  - Let $c = \sum b_i$ be the number of unspecified bits
    • Then $2^c$ buckets must be searched
    • These are certainly not ordered (ordered on what?) – random IO

• Range queries
  - Not supported, if hash function doesn’t preserve order
  - Example of order-preserving hash function??
Order Preserving Hash Functions

• Not order preserving: modulo
• Order preserving: division
• Example
  - Suppose 3 dimensions, each with range 1..1024 (10 bits)
  - Use 3 highest bits as hash key in each dimension
    • Equal to division by 64 (right-shift 7 times)
  - Global hash key: 9 bit, hence $2^9=512$ buckets
  - Partial range query: points with $200<y<300$ and $z<600$
    • $h_y(200)=001$, $h_y(300)=010$, $h_z(600)=100$
    • Scan buckets with
      - X-coordinate: ?
      - Y-coordinate: between 001 and 010 (001, 010)
      - Z-coordinate: <100 (000, 001, 010, 011,100)
    • We need to scan $8 \times 2 \times 4 = 64$ buckets
• But: Very vulnerable to not-uniformly distributed data
  - Data with Gauss distribution (weight, height, age, …) is clustered in the centre of each dimension
  - Use Modulo instead – and lose order-preservation
Conclusions

• Can only store \textit{point objects}
• Has \textit{static address} space as described here
  – Can be combined with extensible/linear hashing
  – Hash keys of different partitions grow/shrink independently
  – Directory in extensible hashing can grow quite large
    • Must be buffered; more IO
• No adaptation to clustered data – overflow buckets or large directories
Grid File

• Probably the most classic multidimensional index structure
  – “Quite” simple: searching, indexing, deleting
  – Good for uniformly distributed data, cannot handle skewed data well
  – Many variations (we will point to different options)

• Design goals
  – Index point objects
  – Support exact, partial match, and neighbor queries
  – Guarantee “two IO” access to each point
    • Under some assumptions
  – Do not prefer any dimension
  – Adapt dynamically to the number of points
Principle

- Partition each dimension into disjoint intervals, called scales
- The intersection of all intervals defines all grid cells
  - Convex d-dimensional hypercubes
  - Grid cells hold pointer to all data objects in that cell
  - When cell overflows – split (no overflow blocks)
  - Each point falls into exactly one grid cell
  - Grid cells are managed in the grid directory
- Grid cells are either
  - Directly addressed – each cell is one bucket = one block on disc
  - Grouped into convex, d-dimensional grid regions
Exact Point Search

• Compute grid cell coordinate
  – We keep scales for each dimension in memory
  – Looking up point coordinate in scales gives coordinates for each dimension
  – Map coordinate to block address on disk
    • Requires that grid directory on disk is organized as an array
    • Costly reorganization upon insertion and deletion – later

• Load grid directory
  – Look up block in grid directory (1st IO)
  – Find pointer to data bucket

• Access data bucket / block
  – 2nd IO
  – Search point in block
Range Query, Partial Match Query

- **Range query**
  - Compute grid cell coordinate for each end point
  - All grid directory entries in that range may contain points

- **Partial match query**
  - Compute partial grid cell coordinates
  - All grid directory entries with these coordinates may contain points
Inserting Points

• Search grid cell
  – If data bucket has space – no problem

• Otherwise
  – Without grid regions
    • Split space
      – Choose a dimension and an interval to split
      – Split all affected grid cells
    • Consider $n$ dimensions and $d_i$ intervals in dimension $i$
      – A split in dimension (last) increases grid directory by $d_1*d_2*...*d_{n-1}$ entries
      – Example: $d=3, d_i=4$
        » Grid directory has $4^3 = 64$ entries
        » Splitting one interval generates $4^2$ new entries
      – Directory blocks need to be reorganized to allow coordinate computation

• Problem – grid directory grows very fast
• Many empty cells (NULL pointer) or almost empty cells
• Choice of dimension and interval is very difficult and never perfect
  – Optimally, we would like to split as many very full blocks as possible
  – This is an optimization problem in itself
Example

- Imagine one block holds 3 pointers
  - Usually we have unevenly spaced intervals
- New point causes overflow
- Where should we split?
- Vertical split
  - Splits 2 (3,4)-point blocks
  - Leaves one 3-point block
- Horizontal split
  - Splits 2 (3,4)-point blocks
  - Leaves one 3-point block
- Need to consider $O(d_i^{n-1})$ regions
Inserting Points -2-

- With grid regions
  - Search grid region
  - Space in bucket of region?
    - No problem
  - Region coarser-grained than scales?
    - Split region into smaller regions (or cells)
    - Possible split dimensions/axes: interval borders not used for split yet
  - Region already at finest level
    - Choose split as without grid regions
    - All but the overflowed grid cell remain unchanged
      - Split is not performed; regions “raise” in granularity
    - Directory need to be extended and reorganized

- Grid regions help to prevent the “many almost empty blocks” problem
Grid File Example 1 (from Johannes Gehrke)

(N=6)

\[ \text{Diagram showing a grid file example with points labeled 1 to 6.} \]
Grid File Example 2

(N=6)

(A

B

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>
Grid File Example 3

(N=6)

A

1

13

8

15

B

6

2

9

C

5

10

11

4

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>7</th>
<th>8</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grid File Example 4

(N=6)

<table>
<thead>
<tr>
<th>A</th>
<th>D</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>8</th>
<th>13</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
Grid File Example 5

(N=6)

\[
\begin{array}{cccccc}
\text{y}_4 & A & H & D & F & B \\
\text{y}_3 & & & & & \\
\text{y}_2 & & & & & \\
\text{y}_1 & & & & & \\
\end{array}
\]

\[
\begin{array}{cccccc}
A & H & D & F & B \\
A & I & D & F & B \\
A & I & G & F & B \\
E & E & G & F & B \\
C & C & C & C & B \\
\end{array}
\]
Deleting Points

- Search point and delete
- If regions become “almost” empty, choose merges
  - A merge is the removal of a split
  - Must build larger convex regions
  - This can become very difficult
    - Potentially, more than two regions need to be merged to keep convexity condition
  - Example:
    Where can we merge regions??

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>H</th>
<th>D</th>
<th>F</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>I</td>
<td>D</td>
<td>F</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>G</td>
<td>F</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>F</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Nearest Neighbor Queries

- Search point
- Search points in same region and choose closest
  - If no point in same region, check surrounding buckets
  - Can we finish if point was found??
Nearest Neighbor Queries

- Search point
- Search points in same region and choose closest
  - If no point in same region, check surrounding buckets
  - Can we finish if point was found??
  - Usually not
    - Compute distance to all interval border (hyperplanes)
    - If point found is closer than all borders, we can finish
What’s in a Bucket?

- Buckets hold “the data”

- Choices
  - Complete tuples
    - Not compatible with other database structures (indexes, etc.)
    - Few records per data blocks
    - Frequent splits, fast growing directory
  - Only TIDs
    - Many records per data block, few splits, small directory
    - But queries need to check (load) all tuples referenced in a block to check real coordinates
    - Too expensive
  - TIDs and coordinates
    - Middle way between other choices
    - Medium number of records per block, moderate size of grid directory
    - No access to tuples necessary for checking coordinates
Conclusions

• Grid files always split at hyperplanes parallel to the dimension axes
  - This is not always optimal
  - Use other than rectangles as cell structure: circles, polygons, etc.
  - More complex forms might not disjointly fill the space any more
  - Allow overlaps - R trees

• Good: Good bucket fill degrees
  - Thanks to grid regions

• Bad: Grid directory grows very fast

• Each split decision finally becomes valid for all covering regions
  - Need not be realized at once, but restricts later choices
  - Bad adaptation to skewed data
  - The more dimensions, the worse
Content of this Lecture

• Introduction to multidimensional indexing
• Partitioned Hashing
• Grid files
• kd and kdb Trees
• R trees
kd-Tree

• Grid file disadvantages
  – All hyperregions of the n-dimensional space are eventually split at the same dimension/position
    • Although not all regions are actually performing the split
  – First cell that overflows determines split
  – This choice is global and never undone

• kd-trees
  – Multidimensional variation of binary search trees
  – Hierarchical splitting of space into regions
  – Regions in different subtrees may use different split positions
  – Better adaptation to clustering of data than grid files
  – kd-tree is mostly a main memory data structure
    • IO-optimization for layout of inner nodes later (kdb)
kd-Tree General Idea

- Binary, rooted tree
- Each inner node has two children
- Path is selected based on a pair (dimension / value)
- Dimensions need not be statically assigned to levels of the tree
  - Can be rotating, random, decided at time of block split, ...
  - Usually: rotating
- Data points are only stored in leaves
- Each leave stores points in a n-dimensional hypercube with m border planes (m ≤ n)
Example – the Brick wall

(x<3, y<1) (x<3, y≥1) (x≥3, y<7) (4,9)

(x<5, y<3) (x<5, y≥3) (x≥5, y<2) (x≥5, y≥2)
Local Adaptation
kd-Tree Search Operations

• Exact point search
  - ??
• Partial match query
  - ??
• Range query
  - ??
• Nearest Neighborhood
  - ??
kd-Tree Search Operations

• Exact point search
  – In each inner node, decide direction based on split condition
  – Search leaf for searched point

• Partial match query
  – If dimension of condition in inner node is part of the query – proceed as for exact match
  – Otherwise, follow all children in parallel
    • Leads to multiple search paths

• Range query
  – Follow all children matching the range conditions
    • Again: multiple search paths

• Nearest Neighborhood
  – Chose likely “close-enough” range and perform range query
  – If no success, iteratively broaden range
kd-Tree Insertion

- Inserting a point
  - Search data block of leaf
  - If space available - done
  - Otherwise, chose split dimension and position for this block
    - This is a local decision, but remains stable for the future
    - Find dimension and split that divides set of points into two sets
      - Consider current points and split in two sets of approximately equal size
      - Consider known distributions of values in different dimensions
      - Use alternation scheme for dimensions
      - Finding “optimal” split points is expensive for high dimensional data (point set needs to be sorted in each dimension) - use heuristics
    - Wrong decisions in early splits lead to tree degradation
      - CS students at HU: Don’t split at sex, religion, place of birth, …
  - But we don’t know which points will be inserted in future
    - Use knowledge on attribute value distributions
kd-Tree Deletions

• Deleting points
  – Search data block and delete point
  – If block becomes empty
    • Leave it – **bad points/space ratio**
    • Delete block and parent node
      – Changes height of tree – danger of **tree degradation**
    • Consider sibling in tree and reorganize
      – Touches more blocks

• Keeping kd-trees balanced is difficult
  – Usually, **some degradation is accepted**

• Improvements for kd-trees on secondary storage??
  – Option 2 is kdb tree – a balanced, IO-optimized kd tree
Fill Inner Blocks

• Option 1: **Multiway branching**
  - Split chosen dimension at \( r \) positions
    
    • \( r \): Number of pointer/value pairs fitting in block
    • When sibling nodes need to be merged,
      - Split points of children usually are incompatible
      - Reorganization of subtrees required
kdb trees

• Option 2: Store entire subtrees in one block
  - Inner nodes still have only two children
    • But those are (usually) stored in the same block
    • We need to “map” nodes to trees
  - kdb-tree: inner nodes store kd-trees

• Operations
  - Searching: As with kd trees
    • But on average better IO complexity
  - Insertion/Deletion
    • Complex schemes for keeping balance in tree (later)
Another View

• Inner nodes define (possibly open) bounding boxes on subtrees
• kdb tree is a hierarchical index structure
Example – Composite Index

• Consider 3 dimensions, \( n=1E7 \) points, block size 4096, \(|\text{point}|=9, \ |b\text{-ptr}|=10\)
  - We need \(~22000\) leaf blocks

• Composite B* index
  - Inner blocks store at least 100 pointers (max \(~220\))
  - We need 3 levels (2nd level has 10,000 pointers)
  - With uniform distribution, 1st level will mostly split on 1st dimension, 2nd level on 2nd dimension, 3rd level on 3rd dimension

• Box query in all three coordinates, 5% selectivity in each dimension
  - We read 5% of 2nd level blocks = 5 IO
  - For each, we read 5% of 3rd level blocks = 5*5=25 IO
  - For each, we read 5% of data blocks = 125 IO
  - Altogether: 155 IO
Example Partial Box Query

- Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  - We need to scan all 100 2nd level blocks
    - Each 2nd level block contains the 5% range
  - Next, we scan 5% of 3rd level blocks = 500 blocks
    - We follow 5% of pointers from 2nd level blocks
  - For each, we read 5% of data blocks = 2500 data blocks
  - Altogether: 3100 IO
- Note: 5% selectivity in 2 dims means 0.0025 selectivity altogether = 25000 points
  - Only 60 blocks if optimally packed
Example – kdb-tree

- **Balanced tree** will have ~14 levels
  - ~400 points in one block (assume optimal packaging)
  - We need to address $1E7/400 = 25,000\sim 2^{14}$ blocks
- Consider $128=2^7$ inner nodes in one block
  - Rough estimate; we need to store two b-ptr for each inner node, but most b-ptr are just offsets into the same block
- **kdb tree structure**
  - 1st level block holds 128 inner nodes = levels 1-7 of kd tree
  - There are 128 2nd level blocks holding levels 8-14 of kd tree
- **1st block evenly splits space in 128 regions**
- Box query in all three coordinates, 5% selectivity in each dimension
  - Overall selectivity is $(0.05)^3 = 0.000125\%$ of all points (1250 points)
  - Very likely, we need to look at only ~1 2nd level block
    - On 7 levels, 2 dim. will have been split into 4 sections, one dim. into 8
    - If query intersects with split planes in each dimension: worst case 8 IO
    - Example: 100 values in one dimension, split will be at 25,50,75, query region covers 5 consecutive values – only 30 of 95 such regions cross a split point
  - Very likely, we need to look at only 4 data blocks (holding together the 1250 points)
  - Altogether: $1+1+4 = 6$ IO (compared to 155 for composite index)
Example - Partial Box Query 2

• Box query on 2nd and 3rd dimensions only, asking for a 5% range in each dimension
  – In first block (7 levels), we will have ~2 splits in each dimension (2 times 2, 1 time 3)
    • Assume we miss the dimension with 3 splits
  – Hence, in ~4 of 7 splits we know where we need to go, in ~3 we need to follow both children
  – We need to check only $2^3 = 8$ second-level blocks
    • Again – number gets higher when query range crosses split plane
    • The less likely, the lower the selectivity is
  – Same argument holds in 2nd level blocks = 8*8 data blocks
    – Altogether: $1 + 8 + 64 = 73$ IO
      • We almost reach optimum with 60 blocks
      • Compare to 3100 for composite index

• Beware
  – We made many, many assumptions
  – Layout of subtrees to nodes rarely optimal
  – Performance can greatly vary due to n# of dimensions, distributions, order of insertions and deletions, selectivity, split and merge policies, …
Conclusion

• Kdb trees can be perfectly balanced
  – Similar method as for b* trees
  – When splitting a leaf, a new node must be inserted into parent
  – Overflow may walk up to root
  – When inner nodes are split, splits must be propagated downward
    • As regions need to stay convex

• Kdb trees have problem with fill degree
  – Many insertions/deletions lead to almost empty leaves
  – Index grows unnecessarily large
  – No guarantee for lowest fill degree as in b* tree
Content of this Lecture

- Introduction to multidimensional indexing
- Partitioned Hashing
- Grid files
- kd and kdb Trees
- R trees
R-Trees

- Can store geometric objects (with size) as well as points
- Each object is stored in exactly one region on each level
- Since sized objects may overlap, R tree regions may overlap
- Better adaptation to distribution of data objects
- Only those hyperregions containing data objects are represented
- Many variations (see literature)
General Idea

• R-trees store n-dimensional rectangles
  - For geometric objects, use minimal bounding box (MBB)
• Objects in a region of the n-dimensional space are stored in a block
  - The region borders is the MBB of all objects in contains
  - Regions may overlap – see below
• Regions are recursively contained in larger regions
  - Tree-like shape
  - Region borders in each level are MBB of all child regions
  - Regions are only as large as necessary
  - Regions of a level need not cover the space completely
• Regions in one level may overlap
  - Or not – variation of classical R tree
  - Without overlaps: much more complicated insertion/deletion, but better search complexity
• Finding all rectangles overlapping with a query rectangle
  - In each level, intersect with all regions
  - More than one region might have non-empty overlap
    • All must be considered
    • In general, no O(log(n)) complexity
Inserting an Object

- In each level, find regions that contains object
  - *Completely or partly*
  - *More than one region with complete overlap*
    - Chose one (smallest?) and descend
  - *None with complete, but several with partial overlap*
    - Chose one (largest overlap?) and descend
  - *No overlapping region at all*
    - Chose one (closest?) and descend
  - We insert object in *only one region*

- In leaf node with space available
  - Insert object and adapt MBB
  - *Recursively adapt MBBs up the tree*
  - This generates larger and larger overlaps - search degrades

- In leaf node with no space available
  - Split block in two regions
  - Compute MBBs
  - Can affect MBB of higher regions - *ascend recursively*
Other Operations

• Deleting an object
  – Equally complicated

• Balancing the R-tree
  – Much more complicated
Example (from Donald Kossmann)

Compute MBBs for non-rectangular objects
Example: Regions

- Objects are *hierarchically grouped* into regions
- Regions may overlap
- Objects are represented only once
Example: Searching
Example: Insertion, Search Phase

- Search regions whose MBB must be expanded the least
- Repeat on each level (do not adapt MBBs yet)

Overflow, split required

Note: Having chose b4 would avoid split – but how can we know?
Example: Insertion, Split Phase

Usually, several splits are possible.
Example: Insertion, Adaptation Phase

- MBBs of all parent nodes must be adapted
- Block split might induce node splits in higher levels of the tree
Block Splits

- Problem: How should we optimally split a overflow-node into two regions?
- Option 1: Avoid overlaps, cover large space
  - Compute partitioning such that there exists a separating hyperplane
  - Minimizes necessity to descend to different children during search
  - Generally requires larger regions – search in empty regions is detected later
- Option 2: Allow overlaps, minimize space coverage
  - Compute partitioning such that sum of volumes of MBBs is minimal
  - Overlaps increases changes to descend on multiple paths during search
  - But: Unsuccessful searches can stop early
Block Splits

- Whatever strategy we chose
  - Consider a block with $n$ objects
  - There are $2^n$ possibilities to partition this block into two
  - Most strategies require to check them all
  - Use heuristics instead of optimal solution

- R* tree
  - Chose as criterion combination of sum of covered spaces, space of intersection, and sum of girt
  - Use heuristic for concrete decision
  - Currently best strategy (still?)
Multidimensional Data Structures Wrap-Up

• We only scratched the surface
• Partitioned Hashing, Gridfile, kdb-Tree, R-Tree
• Other: X tree, hb tree, R+ tree, UB tree, …
  – Store objects more than once; other than rectangular shapes; map coordinates into integers; …
• Curse of dimensionality
  – Your intuition plays tricks on you
  – The more dimensions, the more difficult
    • Balancing the tree, finding MBBs, split decisions, etc.
  – All structures begin to degenerate somehow
    • Exploding size of directories, linear kdb-trees, all regions overlap, …
  – Often, linear scanning of objects is quicker
    • Or: Compute lower-dimensional, relationship-preserving approximations of objects and filter on those
Example

• Assumption: When deleting in object in R-tree, the new MBB will probably not be smaller, since most objects are far from the borders of the region
• Consider cubes and define border as within 10% of border
• In a 1-dimensional interval, 80% or points are not at the border
• In a 2-dimensional rectangle, 64%
• d=5: 32%; d=32: <0.001% - all points are at some border