Content of this Lecture

- Introduction
- 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**: \( \text{const}_1 \leq \text{Attribute} \leq \text{const}_2 \)  (B+ Tree)

• What about 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, …
  - **Similarity queries**: Most similar object, closest object, …
Example: Geometric Objects

- Geographic information systems (GIS) store rectangles
  \[
  \text{RECT} \ (X_1, \ Y_1, \ X_2, \ Y_2)
  \]

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

- Also other shapes: Lines, polygons, 3D, …
Example: 2D objects

- Objects are **points in a 2D space**
- Queries
  - Exact: All objects with coordinates \((X_1, Y_1)\)
  - Box: Find all points in a given rectangle
  - Partial: All points/rectangles with X (Y) coordinate between …

<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>
Option 1: Composite Indexes

- 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

```
CREATE INDEX
ON tab(x,y)
```

<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

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

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

- Partial match query on one attribute: Supported
- Partial match query on many attributes or box query
  - Compute TID lists for each attribute
  - Intersect

CREATE INDEX ON tab(x)
CREATE INDEX ON tab(y)
Example – Independent versus Composite Index

• Data
  - 3 dimensions of range 1,...,100
  - 1,000,000 points, randomly distributed
  - Index blocks holding 50 keys or records

• Range query: Points with \(40 \leq x \leq 50\), \(40 \leq y \leq 50\), \(40 \leq z \leq 50\)
  - Each of the three indexes has height 4
  - Using x-index, we generate TID-list \(|X| \sim 100,000\)
  - Using y-index, we generate TID-list \(|Y| \sim 100,000\)
  - Using z-index, we generate TID-list \(|Z| \sim 100,000\)
  - For each index, we have \(4 + 100,000/50 = 2004\) IO
  - Hopefully, we can keep the three lists in main memory
  - Intersection yields app. 1,000 points, together \(6012\) IO
Composite Index

Indexes on X

Indexes on Y

Indexes on Z
Using composite index (X,Y,Z)

- **Key length increases** – assume \( k = 30 \)
- Index is higher: height \( \sim 5 \)
  - Worst case – index blocks only 50% filled
- We descend in 5 IO to leaves, read 10 points (1 IO),
  ascend to Y-axis (2 IO – but cached), descend to leaves (2 IO),
  read 10 points (1 IO) ...
- We do this 10*10 times
- Altogether
  - \( k = 30 \Rightarrow \text{app. } 3 + 100 \times (2+1) \sim 305 \text{ IO} \)
  - \( k = 10 \Rightarrow \text{app. } 4 + 100 \times (3+1) \sim 404 \text{ IO} \)
- But: Much random IO
Conclusion

- We want composite indexes: Less IO
  - Benefit grows for low-selectivity queries
    - TID lists don’t fit into main memory – paging, more IO
    - If selectivity is low, scanning of relation might be faster than intersection single-attribute indexes
- For partial match queries, we would need to index all attribute combinations – not feasible
- Solution: Use multidimensional indexes
  - Support partial match queries without a pre-defined priority of dimensions
  - Ideally, we would have nearby points nearby on disk
    - In an ideal world, we would need only 1000/30~33 IO
  - Area of intensive research for decades
Multidimensional Indexes

• All dimensions are equally important
• Specialized MDIS for objects with or without extend
• Critical issues
  - Balancing: Worst case search complexity
  - Size: Amount of space required on disk versus # of objects stored
  - Locality: Neighbors in space are (hopefully) stored on nearby blocks
    • Necessary for range / partial match queries
    • Necessary for nearest neighbor queries
      - The nearest, all within distance k
Caveats

• Things get complicated if data is not uniformly distributed
  - Dependent attributes (age – weight, income, height, …)
  - Clustering of points

• Curse of dimensionality: MDIS degrade for many dims
  - Trees difficult to balance, bad space usage, excessive management cost, expensive insertions/deletions, …

• Alternative (partially): Bitmap indexes
  - Very small memory footprint, only for discrete attribute values, range queries become large disjunctions

• Commercially, MDIS are mostly used for …
  - Geometric objects (spatial extender)
  - Native implementation of multidimensional data model (DWH)
Geographic Information Systems
• Customer, logistic centre, supplier, company division, ...
Multimedia Databases

- Map object into feature vector
  - Here: Tumor images; FV derived from mathematical morphology
- Compute **nearest neighborhood queries in feature space**
  - Common approach: Filters away false positives as fast as possible
    - For instance by using FV at different levels of granularity
  - Often, a final check of temporary results is necessary
Content of this Lecture

- Introduction
- Partitioned Hashing
- Grid Files
  - kdb Trees
- R Trees
Partitioned Hashing

- Let $a_1, a_2, ..., a_d$ be the attributes to be indexed
- Define a hash function $h_i$ for each $a_i$ generating a bitstring
- **Definition**
  - Let $h_i(a_i)$ map each $a_i$ into an integer with $b_i$ bit
  - Let $b = \sum b_i$ (length of global hash key in bits)
  - The global hash function $h(v_1, v_2, \ldots, v_d) \rightarrow [0, \ldots, 2^b-1]$ is defined as $h(v_1, v_2, \ldots, v_d) = h_1(v_1) \oplus h_2(v_2) \oplus \ldots \oplus h_k(v_d)$
- We need $B = 2^b$ buckets
  - Static address space – dynamic structures later
Example

- Data: (3,6),(6,7),(1,1),(3,1),(5,6),(4,3),(5,0),(6,1),(0,4),(7,2)
- Let $h_1$, $h_2$ be $(b_1=b_2=1)$
  
  $h_i(a_i) = \begin{cases} 
  0 & \text{if } 0 \leq a_i \leq 3 \\
  1 & \text{otherwise}
  \end{cases}$

- There are four buckets with address 00, 01, 10, 11
Queries with Partitioned Hashing

• Exact point queries: Direct access to bucket

• Partial match queries
  - Only parts of the global hash key are determined
  - Use those as filter; scan all buckets passing the filter
  - Let $c$ be the number of unspecified bits
    - Then $2^c$ buckets must be searched
    - These are certainly not ordered on disk—random IO

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

- **Example**
  - Suppose \( d=3 \), each dim with range 1..1024 (10 bits)
  - Use three highest bits as hash keys in each dimension
    - **Order preserving**; 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 \( z<600 \)
    - \( h_y(200)=0011001000, h_y(300)=0100101100, h_z(600)=1001011000 \)
    - Scan buckets with
      - X-coordinate: ?
      - Y-coordinate: between 001 and 010 (001, 010)
      - Z-coordinate: less than 100 (000, 001, 010, 011)
    - We need to scan \( 8 (x) \times 2 (y) \times 4 (z) = 64 \) buckets

- **Vulnerable to not-uniformly distributed data**
  - Use Modulo instead – and lose order-preservation
Conclusions

• Neighboring points in space or not neighbors on disk
  - (Partial) range queries generate random IO
  - No support for nearest neighbor queries

• **Static address** space (as described here)
  - Problem if buckets overflow
  - Can be combined with extensible/linear hashing
  - Careful: Partitions of the hash function grow independently
  - Directory in extensible hashing can grow quite large
    - Must be buffered; more IO

• No adaptation to **clustered data** – overflow buckets or large directories
Content of this Lecture

- Introduction to multidimensional indexing
- Partitioned Hashing
  - Grid Files
  - kdb Trees
  - R Trees
Grid File

• Classical multidimensional index structure
    - Conceptually simple
    - Can be seen as extensible version of partitioned hashing
    - Good for uniformly distributed data, but for skewed data
    - Numerous variations, we only look at the base form

• Design goals
  - Index point objects
  - Support exact, partial match, and neighbor queries
  - **Guarantee “two IO” access to each point**
    - Under some assumptions
  - **Adapt dynamically** to the number of points
Principle

• Partition each dimension into disjoint intervals (scales)
• Intersection of all intervals defines grid cells
  - Convex d-dimensional hypercubes
• Grid cells are addressed from the grid directory (GD)
Principle

- Partition each dimension into disjoint intervals (scales)
- Intersection of all intervals defines grid cells
  - Convex d-dimensional hypercubes
- Grid cells are addressed from the grid directory (GD)
- Cells are grouped in regions; region = bucket = block
  - When buckets overflows - split region into cells
  - When cells overflow - new scales

- Buckets hold points + TID
Exact Point Search

• Assumption: **GD in main memory**
  - Size: $|S_1| \times |S_2| \times \ldots \times |S_d|$, when $S_i$ is the set of scales for dim $i$
• 1. Compute grid cell
  - Look-up point coordinate in set of scales gives GD coordinates
  - Cell in GD contain bucket address on disk
  - Bucket contains all data points in this grid cell (maybe more)
• 2. **Load block** and find point(s): 1\textsuperscript{st} IO
  - As usual, we do not count search inside the block
• 3. Access record following TID: 2\textsuperscript{nd} IO
Other Queries

- **Range query**
  - Compute all matching scales
  - Access all corresponding cells in GD
  - Load and search all buckets (*random IO*)

- **Partial match query**
  - Compute partial GD coordinates
  - All GD cells with these coordinates may contain points (*random IO*)

- **Neighborhood search**
  - No specific support
  - Compute all surrounding scale intersections
  - If nearest neighbor is searched, *iteratively increase distance*
Inserting Points

- Search grid cell; if bucket has space: Insert point
- Otherwise (overflow): **Split cells**
  - Assume there were no regions – each cell points to its own block
  - Choose a dimension and point in the interval
  - Split all affected grid cells (generates many new cells)
    - Consider \( n \) dimensions and \( S_i \) intervals in dimension \( i \)
    - Split in dim \( i \) affects increases \( d_1 \times \cdots \times d_{i-1} \times d_{i+1} \times \cdots \times d_n \) cells in GD
    - Example: \( d=3, S_i=4; |GD|=4^3=64; \) any split affects \( 4^2 \) cells
  - Problem – Many un-overflown blocks need to be split
  - Many empty cells (NULL pointer) or almost empty blocks
  - Choice of dimension and interval is very difficult and never perfect
    - Optimally, we would like to split as many very full blocks as possible
    - We also want to consider our future expectations
Example

- Imagine one block holds 3 points
  - [Usually scales are unevenly spaced]
- New point causes overflow

- 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
Inserting Points – With Regions

- Regions: Cells pointing to the same block
- Search cell
- Space in block? Yes – insert point
- Otherwise
  - If block is shared
    - Split region into smaller regions (or cells)
    - Possible split dimensions/axes: scales not used for split in this region
      - No local adaptation – decisions from the past have to be obeyed
  - otherwise
    - Choose split as without regions
    - Non-overflowing blocks are untouched – only pointer is “doubled”
      - Split is not performed; regions keep their granularity
- Helps to alleviate the “many almost empty blocks” problem
Assume $k=6$

Grid File Example 1 (from Johannes Gehrke)
Grid File Example 2

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

A  | B  |
---|----|
1  | 3  |
5  | 7  |
7  | 8  |
8  | 10 |
2  | 4  |
6  | 9  |
12 | 11 |
4  | 12 |
Grid File Example 3
Grid File Example 4
We first must perform this split: immediate new overflow, „almost empty block“ again.
Grid File Example 5

<table>
<thead>
<tr>
<th>y_4</th>
<th>y_3</th>
<th>y_2</th>
<th>y_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>H</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>G</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>x_4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deleting Points

• Search point and delete
• If block become "almost empty", try to merge
  - A merge is the removal of a split
  - Must build larger convex regions
    - Or range queries become inefficient
  - This can become very difficult
    - Potentially, more than two regions need to be merged to keep convexity condition
  - Eventually, also scales may be removed

- Example: Where can we merge?
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 scales
    - If point found is closer than all scales, we can finish
    - Otherwise, we need to search neighboring regions
    - Do it iteratively and always adapt radius to current closest points
What’s in a Bucket?

• **Complete tuples**
  - Not compatible with other database structures (indexes, etc.)
  - Few records per data blocks
  - Frequent splits, **fast growing GD**

• **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**

• **TIDs and coordinates**
  - Medium number of records per block, moderate growth of GD
  - No access to tuples necessary for checking coordinates
Observations

- 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 (see R trees)
- There is no guaranteed block fill degree – degeneration
- No local adaptation: GD grows very fast
  - Recall extensible hashing
- Each split finally becomes valid for all covering regions
  - Need not be realized immediately, but restricts later choices
  - Bad adaptation to skewed data