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
  \[ \text{RECT } (X_1, Y_1, X_2, Y_2) ; x_1 < x_2, y_1 < y_2 \]

- Typical GIS queries
  - **Box query**: All rectangles contained in query box \((a_1, b_1)-(a_2, b_2)\)
    
    \[
    \text{SELECT * FROM RECT}
    \text{WHERE } a_1 \leq x_1 \text{ and } b_1 \leq y_1 \text{ and }
    a_2 \geq x_2 \text{ and } b_2 \geq y_2
    \]
  
  - Results in a range query
  - **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 Points

- Objects are points in a 2D space
- Queries
  - Exact: Find all points with coordinates (X1, Y1)
  - Box: Find all points in a given rectangle
  - Partial: Find all points with X (Y) coordinate between ...
Option 1: Composite Index

CREATE INDEX ON tab(x,y)

- Exact queries: Efficiently supported
- Box queries: Efficiently supported
- Partial match query
  - All points with X coordinate between ...: Efficiently supported
  - All points with Y coordinate between ...: Not efficiently supported
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 `tab(Y, X)` – all permutations
    - Combinatorial explosion for more than two attributes
  - Use independent indexes on each attribute
Option 2: Independent Indexes

- Exact query: **Not really efficient**
  - Compute TID lists for each attribute
  - Intersect
- Box query: **Not really efficient** (compute ranges, intersect)
- Partial match query on one attribute: Efficiently supported

```
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 leaves holding k=50 keys or records

- **Assume three independent indexes**
- **Range query**: Points with $40 \leq x \leq 50$, $40 \leq y \leq 50$, $40 \leq z \leq 50$
  - Each of the three B+-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
Intuition

Source: T. Grust, 2010
Composite Index

- Index on X
- Indexes on Y
- Indexes on Z
- Indexes on Y
- Indexes on Z
- Indexes on Y
- Indexes on Z
- Indexes on Y
- Indexes on Z
Using composite index \((X,Y,Z)\)

- **Key length increases** – assume \(k=30\) (or 10 / more dims)
- Index is higher: Height \(\sim 5\) (6)
  - 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

  - \(k=30\) => app. \(3+100\times(2+1)\) \(\sim 303\) IO
    - Compared to 6012 for independent indexes!
  - \(k=10\) => app. \(4+100\times(3+1)\) \(\sim 404\) IO
Conclusion

• **We want composite indexes:** Less IO
  – Benefit grows for highly selective queries
  – But: If selectivity is low, scanning of relation might be faster than any index (sequential versus random IO)

• For partial match queries, we would need to index all attribute combinations – not feasible

• Solution: Use **multidimensional index structures** (MDIS)
Multidimensional Indexes

- Specialized IS for MD-objects with or without extend
  - Points versus shapes
  - Should have no priority or preferred dimensions
  - Should adapt to uneven and changing data distribution
  - Should have low worst case complexity (balanced structures)
  - Should not use too much space
  - Locality: Neighbors in space are stored nearby on disk (memory)
    - In an ideal world, we would need only $1000/30 \approx 33$ IO
    - Necessary for efficient range queries
    - Desirable for nearest neighbor queries; not in this lecture

- Area of intensive research for decades
Caveats

• In commercial DBMS, high dim data is supported for
  – Geometric objects: GIS extensions, spatial extender
  – Multimedia data (images, songs, ...)

• Things get tricky if data is not uniformly distributed
  – Dependent / correlated attributes (age – weight, income, height)
  – Clustered values (e.g. population density)
  – Special distributions (normal, Zipf, ...)
  – Skew – deviation from assumed distribution

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

• Alternative: Scans, bitmap indices
Geographic Information Systems
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 a bitstring of length $b_i$
  - Let $b = \sum b_i$ (length of global hash key in bits)
  - The global hash function $h(v_1, v_2, ..., v_d) \rightarrow [0, ..., 2^b-1]$ is defined as $h(v_1, v_2, ..., v_d) = h_1(v_1) \oplus h_2(v_2) \oplus ... \oplus h_k(v_d)$
- 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 ($b_1=b_2=1$, $b=2$)
  \[ h_i(v_i) = \begin{cases} 
  0 & \text{if } 0 \leq v_i \leq 3 \\
  1 & \text{otherwise} 
\end{cases} \]

- Four buckets with addresses 00, 01, 10, 11

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>a_2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(1,1)</td>
<td>(3,1)</td>
<td>(3,6)</td>
</tr>
<tr>
<td>1</td>
<td>(4,3)</td>
<td>(5,0)</td>
<td>(6,1)</td>
</tr>
</tbody>
</table>

| (7,2) |
Queries with Partitioned Hashing

• Exact point queries: Direct access to bucket
  – All points in bucket are candidates; check identity to query

• 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 efficiently supported, if hash function doesn’t preserve order
    • Not order preserving: modulo; order preserving: division
Order Preserving Hash Function (OPH)

- **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 $z < 600$
    - $h_y(200)=001\overline{1001000}$, $h_y(300)=010\overline{0101100}$, $h_z(600)=100\overline{1011000}$
    - Scan buckets with
      - X-coordinate: ?
      - Y-coordinate: between 001 and 010 (001, 010)
      - Z-coordinate: less than 100... (000, 001, 010, 011,100)
    - We need to scan $8 \times 2 \times 5 = 80$ buckets

- **Vulnerable to not-uniformly distributed data**
  - Few buckets are extremely full, others empty

Without OPH:
Enumerate all values in DB and compute hashkeys
Partitioned Hashing: Conclusions

- No balancing, no adaptation to **skew**
  - Long overflow buckets or large directories
- **Size**: Static size of hash table, no adaptation
  - Problem if buckets overflow
  - Can be combined with extensible/linear hashing
- **Locality**: Neighboring points in space not nearby in index
  - Usually, hash functions are not order preserving to achieve more uniform spread
  - Bad support for (partial) range queries or nearest neighbor queries
Content of this Lecture

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

• Classical multidimensional index structure
  – Can be seen as extensible version of partitioned hashing
  – Good for uniformly distributed data, **bad for skewed data**
  – Numerous variations, we only look at the basic method

• Design goals
  – Aims to support exact, partial match, and neighbor queries
  – **Guarantee “two IO” access to each point**
    • Under certain assumptions
  – **Adapt dynamically** to the number of points
Principle

- Partition each dimension into disjoint intervals (scales)
  - EXCESS: Uniform scales; less adaptive, no scale management
- Intersection of all intervals defines grid cells
  - d-dimensional hypercubes
- Grid cells are addressed from the grid directory (GD)
  - A simple multidimensional array
Principle

- Partition each dimension into disjoint intervals (scales)
- Intersection of all intervals defines grid cells
- Grid cells are addressed from the grid directory (GD)
- Cells are grouped in regions; region = bucket = block
  - When multi-cell region overflows – split into cells
  - When single-cell region overflows – new scale, change GD
- Buckets hold values + TID
Exact Point Search

- **Assumption:** GD in main memory
  - Size: $|S_1|*|S_2|*...*|S_d|$, when $S_i$ is the set of scales for dimension $I$
  - Too large for really high dimensional data ($d>10$)

1. Compute grid cell
   - Look-up coordinates in scales to obtain GD coordinates
   - Cell in GD contains region/bucket address on disk
   - Bucket contains all data points in this grid cell (maybe more)

2. Load bucket and find point(s): 1st IO
   - As usual, we do not look at how to search inside a bucket

3. Access record following TID: 2nd 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)*
Nearest Neighbor Queries

- Find bucket containing query point
- Search points in **this region** and choose closest
  - Can we finish with the closest point in this region?
Nearest Neighbor Queries

• Find bucket containing query point
• Search points in this region and choose closest
  – Can we finish with the closest point in this region?
  – Usually not
    • Check distances to all borders
    • If point found is closer than any border, we are done
    • Otherwise, we need to search neighboring regions
    • Do it iteratively and always adapt radius to current closest point
  – Very fast if neighbor is in same region
    • I.e.: dense buckets and query point not at a border
Inserting Points

- Search grid cell; if bucket has space: Insert point
- Otherwise (overflow): **Split**
  - Assume we have to split a single-cell region
  - Choose a dimension and **new scale** within region interval
  - Split **all affected GD cells** – cuts through all dimensions
    - Consider \( n \) dimensions and \( S_i \) scales in dimension \( i \)
    - Split in dim \( i \) affects \( d_1 \times \ldots \times d_{i-1} \times d_{i+1} \times \ldots \times d_n \) cells in GD
    - Example: \( d=3, S_i=4; |GD|=4^3=64; \) any split affects \( 4^2 \) cells
  - Split overflowed bucket along new scale (**new region**)
  - Do not split other (un-overflown) buckets containing the new scale
    - Only **copy pointers** within GD
  - Choice of dimension and interval is difficult
    - Optimally, we would like to “split” many rather full blocks
    - We also want to consider our **future expectation**
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
- Note: Real splits will happen only in the future
Choosing a Split

- We wish
  - W1: Split points evenly in overflow bucket
  - W2: Future-Split points evenly other affected buckets
  - W3: Split future points within bucket range evenly
  - W4: Future-Split future points within other affected buckets

- Wishes typically are contradicting
- W1: Sort points in every direction and chose median
- W2 is expensive: Load all affected blocks in every dim.
- W3, W4: Require guessing the future
  - W1 and W2 assume that future distribution is same as past distribution

- Alternative: Round-robin in dimensions and chose median
Inserting Points in Multi-Cell Regions

- **Overflow in a multi-cell region**
  - A bucket to which multiple GD entries point

- **Action: Split region into smaller regions (or cells) along existing, not yet realized scales**
  - GRID file only considers *existing scales not yet used for split* in this region
    - No local adaptation – decisions from the past have to be obeyed
    - GD structure is left unchanged; only cell entries change

- **Which scale to use (there may be more than one)?**
  - This is a *local decision*
  - Chose splits that best distribute the bucket that is split
Grid File Example 1 [J. Gehrke]

Assume $k=6$
Grid File Example 2
Grid File Example 3

```

A
  1  14
  13  7
  8  15
B
  6
  2  9
C
  5
  3  10

A  B
C  B

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

A

1

13

16

8

14

D

7

15

B

6

2

9

C

3

10

11

4

A | D | B

| C | C | B

A | 1 | 8 | 13 | 16

| B | 2 | 4 | 6 | 9 | 11 | 12

| C | 3 | 5 | 10 |

| D | 7 | 14 | 15 |
One Future

We now must perform this split; creates one almost empty and one full bucket; next split will happen soon
Grid File Example 5

A B C D E F G H I

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y1</th>
<th>y2</th>
<th>y3</th>
<th>y4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>A</td>
<td>I</td>
<td>G</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>
Deleting Points

• Search point and delete
• If bucket becomes "almost empty", try to merge with other buckets
  – A merge is the removal of a split – chose scale to "unmake"
  – Should build larger convex regions
  – This can become difficult
    • Potentially, more than two regions need to be merged to keep convexity
  – Eventually, also scales may be removed
    • Shrinkage of GD
  – Example: Where can we merge?

<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>
Convex Regions

- Non-convex regions: Range and neighborhood queries have to scan increasingly many buckets
Some Observations

- Grid files always split at hyperplanes parallel to the dimension axes
  - This is not always optimal
  - Use other bounding shapes: 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
- Choosing a new scale is a local decision with global consequences
  - No local adaptation: GD grows very fast
  - Need not be realized immediately, but restricts later choices in other regions
  - Bad adaptation to skewed data