Multidimensional Spatial Data Structures

Nick Kuilema
nicnak@cs.umd.edu

Survey Paper by Jeffery Scott Vitter
I/O Crisis!

Time for rotation ≈ Time for seek.

Amortize search time by large block transfer so that
Time for rotation ≈ Time for seek ≈ Time to transfer data.

Parallel disks.
Multidimensional Data Spaces

• Want to provide efficient searching
• Insertion and deletion
• Needed for GIS range search
Desired Complexity

Same at using a B-tree for 1-D range search

1. Get a combined search and answer cost for queries of $O(\log_B N + z)$ I/Os,

2. Use only a linear amount (namely, $O(n)$ blocks) of disk storage space, and

Try for Linear Space

- Cross Tree
  - Upper levels - Weight balanced B-Tree
  - Lower levels - Quad Trees
- O-Tree
Quadtree
R-Trees

- Internal nodes have degree $\Theta(B)$
- Leaves store $\Theta(B)$ items
- Each node has a bounding box
- Points may be in more than one child node
R-trees

🌟 Structure for storing $d$-dimensional rectangles.
🌟 Structured like B-tree:
  - Data in leaves.
  - Fan-out $B$.
  - Rebalancing basically like in B-trees.
🌟 Internal node holds minimal bounding rectangle of each subtree.
Querying R-trees

☆ Query with point \( q \):
  - Visits all nodes with minimal bounding rectangle containing \( q \).

☆ Minimal bounding rectangles allowed to overlap:
  - Small overlap or perimeter desirable.
  - Several insert/split heuristics (\( R^+ \)-trees, \( R^* \)-trees, Hilbert trees, ...) have been proposed, surveyed in [G89, GG98].
R*-Tree

• Seems to give best performance
• Heuristic for Insertion
  • Recursively insert into bounding box that expands the least
  • If tie, add to subtree with smallest bounding box
• When a node is full remove a percentage of elements and re-insert
  • If node is still full, split it
• Improves packing and query times
Constructing ("Bulk Loading") an R-tree

- Using repeated insertion takes $O(N \log_B n)$ I/Os.

- Bottom-up algorithms [RL85, KF93, DKLPY94, LLE96, vdBSW97]
  - Rectangles are sorted (using space-filling curve)
    $\implies O(n \log_m n)$-I/O algorithm.
  - Can only handle construction—not e.g. “bulk updates.”
  - Questionable query performance, esp. in high dimensions.

- Buffer technique immediately applies:
  - Conceptually simple (algorithm unchanged).
  - Modular design (all R-tree insert heuristics can be used).
  - Handles all “bulk” operations.
## TIGER/Line Data

- TIGER/Line data from U.S. Census Bureau
  (standard benchmark data for spatial databases)

<table>
<thead>
<tr>
<th>State</th>
<th>Category</th>
<th>Size</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rhode Island (RI)</td>
<td>Roads</td>
<td>4.3 MB</td>
<td>68,278</td>
</tr>
<tr>
<td></td>
<td>Hydrography</td>
<td>0.4 MB</td>
<td>7,013</td>
</tr>
<tr>
<td>Connecticut (CT)</td>
<td>Roads</td>
<td>12.0 MB</td>
<td>188,643</td>
</tr>
<tr>
<td></td>
<td>Hydrography</td>
<td>1.8 MB</td>
<td>28,776</td>
</tr>
<tr>
<td>New Jersey (NJ)</td>
<td>Roads</td>
<td>26.5 MB</td>
<td>414,443</td>
</tr>
<tr>
<td></td>
<td>Hydrography</td>
<td>3.2 MB</td>
<td>50,854</td>
</tr>
<tr>
<td>New York (NY)</td>
<td>Roads</td>
<td>55.7 MB</td>
<td>870,413</td>
</tr>
<tr>
<td></td>
<td>Hydrography</td>
<td>10.0 MB</td>
<td>156,568</td>
</tr>
<tr>
<td>All</td>
<td>Roads</td>
<td>98.5 MB</td>
<td>1541,777</td>
</tr>
<tr>
<td></td>
<td>Hydrography</td>
<td>15.4 MB</td>
<td>243,211</td>
</tr>
</tbody>
</table>
Experimental Results: R-tree

- Buffers on all nodes for simplicity (buffer size $\Theta(B)$)

![Histograms showing I/Os for building R-tree and querying with hydro data](histograms.png)

- Naive repeated insertion: [Filled bar]

- Buffer: Size of buffer
  - $\frac{B^2}{4}$
  - $\frac{B^2}{2}$
  - $2B^2$
## Experimental Results: I/O Costs for R-trees

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Update Method</th>
<th>Update with 25% of the data Building</th>
<th>Querying</th>
<th>Packing</th>
</tr>
</thead>
<tbody>
<tr>
<td>RI</td>
<td>naive</td>
<td>259,263</td>
<td>6,670</td>
<td>64%</td>
</tr>
<tr>
<td></td>
<td>Hilbert</td>
<td>15,865</td>
<td>7,262</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td>13,484</td>
<td>5,485</td>
<td>90%</td>
</tr>
<tr>
<td>CT</td>
<td>naive</td>
<td>805,749</td>
<td>40,910</td>
<td>66%</td>
</tr>
<tr>
<td></td>
<td>Hilbert</td>
<td>51,086</td>
<td>40,593</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td>42,774</td>
<td>37,798</td>
<td>90%</td>
</tr>
<tr>
<td>NJ</td>
<td>naive</td>
<td>1,777,570</td>
<td>70,830</td>
<td>66%</td>
</tr>
<tr>
<td></td>
<td>Hilbert</td>
<td>120,034</td>
<td>69,798</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td>101,017</td>
<td>65,898</td>
<td>91%</td>
</tr>
<tr>
<td>NY</td>
<td>naive</td>
<td>3,736,601</td>
<td>224,039</td>
<td>66%</td>
</tr>
<tr>
<td></td>
<td>Hilbert</td>
<td>246,466</td>
<td>230,990</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>buffer</td>
<td>206,921</td>
<td>227,559</td>
<td>90%</td>
</tr>
</tbody>
</table>
Space Filling Curves

• Impose a total ordering over a multi-dimensional space
• Hilbert curve
• Z-order
• Sierpiński curve
• Peano curve
2-D Hilbert Curve
3-D Hilbert Curve
HPC application Moldyn

- Molecular dynamics simulation
- Computes interactions between each pair of particles
- Hundreds of thousands of particles
- Millions of interactions
Moldyn results

First touch re-ordering is reordering data set as it is visited

<table>
<thead>
<tr>
<th>Data Reordering</th>
<th>Computation Reordering</th>
<th>L1 Cache Misses</th>
<th>L2 Cache Misses</th>
<th>TLB Misses</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCM</td>
<td>None</td>
<td>0.96441</td>
<td>0.81847</td>
<td>0.49658</td>
<td>0.86650</td>
</tr>
<tr>
<td>First Touch</td>
<td>None</td>
<td>0.87487</td>
<td>0.76548</td>
<td>0.31928</td>
<td>0.79069</td>
</tr>
<tr>
<td>Hilbert</td>
<td>None</td>
<td>0.87978</td>
<td>0.78074</td>
<td>0.26397</td>
<td>0.80731</td>
</tr>
<tr>
<td>None</td>
<td>Hilbert</td>
<td>0.45053</td>
<td>0.12157</td>
<td>0.74006</td>
<td>0.37778</td>
</tr>
<tr>
<td>None</td>
<td>Blocking</td>
<td>0.30376</td>
<td>0.23557</td>
<td>0.19278</td>
<td>0.61910</td>
</tr>
<tr>
<td>First Touch</td>
<td>Hilbert</td>
<td>0.33735</td>
<td>0.14314</td>
<td>0.00806</td>
<td>0.38773</td>
</tr>
<tr>
<td>Hilbert</td>
<td>Hilbert</td>
<td>0.25816</td>
<td>0.10139</td>
<td>0.00624</td>
<td>0.26550</td>
</tr>
</tbody>
</table>
For example, indexing constraints in constraint query languages:
External Range Searching Results

★ **Corner (Interval tree):** $O(n)$ space, $O(\log_B n + z)$ I/Os query, $O(\log_B n)$ I/Os update [AV96]

★ **2-sided:** $O(n \log \log B)$ space, $O(\log_B n + z)$ I/Os query, $O(\log_B n)$ amortized updates [RS94]

★ **3-sided:** $O(n)$ space, $O(\log_B n + z + IL^*(B))$ I/Os query, $O(\log_B n + \frac{1}{B}(\log_B n)^2)$ I/Os amortized updates [SR95]

★ **4-sided:** $O\left(n(\log N)/\log \log_B n\right)$ space, $O(\log_B n + z + IL^*(B))$ I/Os query [SR95]

★ **3-d range queries:** $O\left((\log \log \log_B n) \log_B n + z\right)$ I/Os query [VV96]

★ **Halfspace queries:** $O(n \log n)$ space, $O(\log_B n + z)$ I/Os query [AAEFV98]

★ **Lower bounds:** [SR95] Cannot achieve simultaneously $O\left(n(\log n)/\log \log_B n\right)$ space, $O\left((\log_B n)^c + z\right)$ I/Os query.
Acknowledgements

Jeffery Scott Vitter, Duke. Paper and slides at
http://www.cs.duke.edu/~jsv/Papers/catalog/node38.html

John Mellor-Crummey, David Whalley, Ken Kennedy