External Memory Geometric Data Structures

Lars Arge
Duke University

June 27, 2002

Summer School on Massive Datasets
External Memory Geometric Data Structures

• Many massive dataset applications involve geometric data (or data that can be interpreted geometrically)
  – Points, lines, polygons
• Data need to be stored in data structures on external storage media such that on-line queries can be answered I/O-efficiently
• Data often need to be maintained during dynamic updates

• Examples:
  – Phone: Wireless tracking
  – Consumer: Buying patterns (supermarket checkout)
  – Geography: NASA satellites generate 1.2 TB per day
Example: LIDAR terrain data

- Massive (irregular) point sets (1-10m resolution)
- Appalachian Mountains (between 50GB and 5TB)
- Need to be queried and updated efficiently

Example: Jockey’s ridge (NC cost)
External memory data structures

Model

- **Model** as previously
  - $N$ : Elements in structure
  - $B$ : Elements per block
  - $M$ : Elements in main memory
  - $T$ : Output size in searching problems

- **Focus** on
  - Worst-case structures
  - Dynamic structures
  - Fundamental structures
  - Fundamental design techniques
Outline

• **Today:** Dimension one
  – External search trees: B-trees
  – Techniques/tools
    * Persistent B-trees (search in the past)
    * Buffer trees (efficient construction)

• **Tomorrow:** “Dimension 1.5”
  – Handling intervals/segments (interval stabbing/point location)
  – Techniques/tools: Logarithmic method, weight-balanced B-trees, global rebuilding

• **Saturday:** Dimension two
  – Two-dimensional range searching
External Search Trees

- Binary search tree:
  - Standard method for search among \( N \) elements
  - We assume elements in leaves

\[
O(\log_2 N)
\]

- Search traces at least one root-leaf path
- If nodes stored arbitrarily on disk
  \( \Rightarrow \) Search in \( O(\log_2 N) \) I/Os
  \( \Rightarrow \) Rangesearch in \( O(\log_2 N + T) \) I/Os
External memory data structures

**External Search Trees**

\[ O(\log_2 B) \]

\[ \Theta(B) \]

- BFS blocking:
  - Block height \( O(\log_2 N) / O(\log_2 B) = O(\log_B N) \)
  - Output elements blocked

\[ \downarrow \]

Rangesearch in \( O(\log_B N + \frac{T}{B}) \) I/Os

- **Optimal**: \( O(\frac{N}{B}) \) space and \( O(\log_B N + \frac{T}{B}) \) query
External Search Trees

• Maintaining BFS blocking during updates?
  – Balance normally maintained in search trees using rotations

• Seems very difficult to maintain BFS blocking during rotation
  – Also need to make sure output (leaves) is blocked!
B-trees

- BFS-blocking naturally corresponds to tree with fan-out $\Theta(B)$

- B-trees balanced by allowing node degree to vary
  - Rebalancing performed by splitting and merging nodes
**(a,b)-tree**

- $T$ is an $(a,b)$-tree ($a \geq 2$ and $b \geq 2a - 1$)
  - All leaves on the same level (contain between $a$ and $b$ elements)
  - Except for the root, all nodes have degree between $a$ and $b$
  - Root has degree between 2 and $b$

- $(a,b)$-tree uses linear space and has height $O(\log_a N)$

\[ \downarrow \]

Choosing $a,b = \Theta(B)$ each node/leaf stored in one disk block

\[ \downarrow \]

$O(N/B)$ space and $O(\log_B N + T/B)$ query
(a,b)-Tree Insert

- Insert:

Search and insert element in leaf $v$
DO $v$ has $b+1$ elements

Split $v$:
make nodes $v'$ and $v''$ with
$\left\lfloor \frac{b+1}{2} \right\rfloor \leq b$ and $\left\lceil \frac{b+1}{2} \right\rceil \geq a$ elements
insert element (ref) in $parent(v)$
(make new root if necessary)
$v = parent(v)$

- Insert touch $O(\log_a N)$ nodes
(a,b)-Tree Insert
(a,b)-Tree Delete

- **Delete:**

  Search and delete element from leaf $v$
  
  DO $v$ has $a$-1 children
  
  **Fuse** $v$ with sibling $v'$:
  
  move children of $v'$ to $v$
  
  delete element (ref) from $\text{parent}(v)$
  
  (delete root if necessary)
  
  If $v$ has $>b$ (and $\leq a+b-1$) children split $v$
  
  $v = \text{parent}(v)$

- **Delete touch** $O(\log_a N)$ nodes
(a,b)-Tree Delete
(a,b)-Tree

- (a,b)-tree properties:
  - If $b = 2a - 1$ one update can cause many rebalancing operations
  - If $b \geq 2a$ update only cause $O(1)$ rebalancing operations amortized
  - If $b > 2a$ $O\left(\frac{1}{b/2 - a}\right) = O\left(\frac{1}{a}\right)$ rebalancing operations amortized
    * Both somewhat hard to show
  - If $b = 4a$ easy to show that update causes $O\left(\frac{1}{a} \log_a N\right)$ rebalance operations amortized
    * After split during insert a leaf contains $\approx 4a/2 = 2a$ elements
    * After fuse (and possible split) during delete a leaf contains between $\approx 2a$ and $\approx 5/2 a$ elements
(a,b)-Tree

- (a,b)-tree with leaf parameters \( a_l, b_l \) (\( b=4a \) and \( b_l=4a_l \))
  - Height \( O(\log_a \frac{N}{a_l}) \)
  - \( O\left(\frac{1}{a_l}\right) \) amortized leaf rebalance operations
  - \( O\left(\frac{1}{a\cdot a_l}\log_a N\right) \) amortized internal node rebalance operations

- B-trees: (a,b)-trees with \( a, b = \Theta(B) \)
  - B-trees with elements in the leaves sometimes called B\(^+\)-tree

- Fan-out \( k \) B-tree:
  - \((k/4,k)\)-trees with leaf parameter \( \Theta(B) \) and elements in leaves

- Fan-out \( \Theta(B^{\frac{1}{c}}) \) B-tree with \( c \geq 1 \)
  - \( O(N/B) \) space
  - \( O(\log_{B^{\frac{1}{c}}} N + \frac{T}{B}) = O(\log_B N + \frac{T}{B}) \) query
  - \( O(\log_B N) \) update
Persistent B-tree

- In some applications we are interested in being able to access previous versions of data structure
  - Databases
  - Geometric data structures (later)
- Partial persistence:
  - Update current version (getting new version)
  - Query all versions

- We would like to have partial persistent B-tree with
  - $O(N/B)$ space – $N$ is number of updates performed
  - $O(\log_B N)$ update
  - $O(\log_B N + T/B)$ query in any version
Persistent B-tree

- East way to make B-tree partial persistent
  - Copy structure at each operation
  - Maintain “version-access” structure (B-tree)

- Good $O(\log_B N + T/B)$ query in any version, but
  - $O(N/B)$ I/O update
  - $O(N^2/B)$ space
Persistent B-tree

• Idea:
  – Elements augmented with "existence interval"
  – Augmented elements stored in one structure
  – Elements "alive" at "time" $t$ (version $t$) form B-tree

  – Version access structure (B-tree) to access B-tree root at time $t$
Persistent B-tree

- Directed **acyclic graph** with elements in leaves (sinks)
  - Routing elements in internal nodes
- Each element (routing element) and node has **existence interval**
- Nodes **alive** at time $t$ make up $(B/4, B)$-tree on alive elements
- B-tree on all roots (version access structure)

\[ \downarrow \]

Answer query at version $t$ in $O(\log_B N + \frac{T}{B})$ I/Os as in normal B-tree

- **Additional invariant:**
  - New node (only) contains between $\frac{3}{8} B$ and $\frac{7}{8} B$ live elements

\[ \downarrow \]

$O(N/B)$ blocks
**Persistent B-tree Insert**

- Search for relevant leaf $l$ and insert new element
- If $l$ contains $x > B$ elements: **Block overflow**
  - Version split:
    - Mark $l$ dead and create new node $v$ with $x$ alive element
  - If $x > \frac{7}{8} B$: **Strong overflow**
  - If $x < \frac{3}{8} B$: **Strong underflow**
  - If $\frac{3}{8} B \leq x \leq \frac{7}{8} B$ then recursively update $parent(l)$:
    - **Delete** reference to $l$ and **insert** reference to $v$
Persistent B-tree Insert

- **Strong overflow** \((x > \frac{7}{8} B)\)
  - Split \(v\) into \(v'\) and \(v''\) with \(\frac{x}{2}\) elements each \((\frac{3}{8} B < \frac{x}{2} \leq \frac{1}{2} B)\)
  - Recursively update \(parent(l)\):
    - Delete reference to \(l\) and insert reference to \(v'\) and \(v''\)

- **Strong underflow** \((x < \frac{3}{8} B)\)
  - Merge \(x\) elements with \(y\) live elements obtained by version split on sibling \((x + y \geq \frac{1}{2} B)\)
  - If \(x + y \geq \frac{7}{8} B\) then (strong overflow) perform split
  - Recursively update \(parent(l)\):
    - Delete two references insert one or two references
Persistent B-tree Delete

- Search for relevant leaf $l$ and mark element dead
- If $l$ contains $x < \frac{1}{4}B$ alive elements: Block underflow
  - Version split:
    Mark $l$ dead and create new node $v$ with $x$ alive element
  - Strong underflow ($x < \frac{3}{8}B$):
    Merge (version split) and possibly split (strong overflow)
  - Recursively update $parent(l)$:
    Delete two references insert one or two references
Persistent B-tree

Insert

Block overflow

Version split

Strong overflow

Split

don -1,+2

don -1,+1

Delete

Block underflow

Version split

Strong underflow

Merge

Strong overflow

don -2,+2

don -2,+1

0,0
Persistent B-tree Analysis

• **Update:** $O(\log_B N)$
  - Search and “rebalance” on one root-leaf path

• **Space:** $O(N/B)$
  - At least $\frac{1}{8} B$ updates in leaf in *existence interval*
  - When leaf $l$ die
    * At most two other nodes are created
    * At most one block over/underflow one level up (in $\text{parent}(l)$)

  \[ \begin{align*}
  &\text{During } N \text{ updates we create:} \\
  &\quad \ast \quad O\left(\frac{N}{B}\right) \text{ leaves} \\
  &\quad \ast \quad O\left(\frac{N}{B^i}\right) \text{ nodes } i \text{ levels up} \\
  &\Rightarrow \text{Space: } \bigoplus_{i} O\left(\frac{N}{B^i}\right) = O\left(\frac{N}{B}\right)
  \end{align*} \]
Summary: B-trees

- **Problem**: Maintaining $N$ elements dynamically

- **Fan-out $\Theta(B^{1/c})$ B-tree ($c \geq 1$)**
  - Degree balanced tree with each node/leaf in $O(1)$ blocks
  - $O(N/B)$ space
  - $O(\log_B N + T/B)$ I/O query
  - $O(\log_B N)$ I/O update

- **Space and query optimal in comparison model**

- **Persistent B-tree**
  - Update current version
  - Query all previous versions
Other B-tree Variants

• **Weight-balanced B-trees**
  – Weight instead of degree constraint
  – Nodes high in the tree do not split very often
  – Used when secondary structures are used

  *More later!*

• **Level-balanced B-trees**
  – Global instead of local balancing strategy
  – Whole subtrees rebuilt when too many nodes on a level
  – Used when parent pointers and divide/merge operations needed

• **String B-trees**
  – Used to maintain and search (variable length) strings

  *More later (Paolo)*
B-tree Construction

• In internal memory we can sort $N$ elements in $O(N \log N)$ time using a balanced search tree:
  – Insert all elements one-by-one (construct tree)
  – Output in sorted order using in-order traversal

• Same algorithm using B-tree use $O(N \log_B N)$ I/Os
  – A factor of $O(B \frac{\log M}{\log B})$ non-optimal

• We could of course build B-tree bottom-up in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os
  – But what about persistent B-tree?
  – In general we would like to have dynamic data structure to use in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ algorithms $\Rightarrow O(\frac{1}{B} \log_{M/B} \frac{N}{B})$ I/O operations
Buffer-tree Technique

- **Main idea**: Logically group nodes together and add buffers
  - Insertions done in a “lazy” way – elements inserted in buffers.
  - When a buffer runs full elements are pushed one level down.
  - Buffer-emptying in $O(M/B)$ I/Os
    - $\Rightarrow$ every *block* touched constant number of times on each level
    - $\Rightarrow$ inserting $N$ elements ($N/B$ blocks) costs $O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os.
**Basic Buffer-tree**

- **Definition:**
  - Fan-out $\frac{M}{B}$ B-tree — $(\frac{1}{4} \cdot \frac{M}{B}, \frac{M}{B})$-tree with size $B$ leaves
  - Size $M$ buffer in each internal node

- **Updates:**
  - Add time-stamp to insert/delete element
  - Collect $B$ elements in memory before inserting in root buffer
  - Perform **buffer-emptying** when buffer runs full
Basic Buffer-tree

- Note:
  - Buffer can be larger than $M$ during recursive buffer-emptying
    * Elements distributed in sorted order
      $\Rightarrow$ at most $M$ elements in buffer unsorted
  - Rebalancing needed when “leaf-node” buffer emptied
    * Leaf-node buffer-emptying only performed after all full internal node buffers are emptied
Basic Buffer-tree

- Internal node buffer-empty:
  - Load first $M$ (unsorted) elements into memory and sort them
  - Merge elements in memory with rest of (already sorted) elements
  - Scan through sorted list while
    * Removing “matching” insert/deletes
    * Distribute elements to child buffers
  - Recursively empty full child buffers

- Emptying buffer of size $X$ takes $O(X/B+M/B)=O(X/B)$ I/Os
Basic Buffer-tree

- **Buffer-empty** of leaf node with $K$ elements in leaves

  - Sort buffer as previously
  - Merge buffer elements with elements in leaves
  - Remove “matching” insert/deletes obtaining $K'$ elements
  - If $K' < K$ then
    * Add $K-K'$ “dummy” elements and insert in “dummy” leaves
  
  Otherwise
  * Place $K$ elements in leaves
  * Repeatedly insert block of elements in leaves and rebalance

- Delete dummy leaves and rebalance when all full buffers emptied
Basic Buffer-tree

• Invariant:
  Buffers of nodes on path from root to emptied leaf-node are empty
  \[ \downarrow \]

  • Insert rebalancing (splits)
    performed as in normal B-tree
  
  \[ \overset{v}{\longrightarrow} \overset{v'}{\longrightarrow} \overset{v''}{\longrightarrow} \]

  • Delete rebalancing: \(v'\) buffer emptied before fuse of \(v\)
    – Necessary buffer emptyings performed before next dummy-block delete
    – Invariant maintained
  
  \[ \overset{v}{\longrightarrow} \overset{v'}{\longrightarrow} \overset{v}{\longrightarrow} \]
Basic Buffer-tree

- Analysis:
  - Not counting rebalancing, a buffer-emptying of node with $X \geq M$ elements (full) takes $O(X/B)$ I/Os
    $\Rightarrow$ total full node emptying cost $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os
  - Delete rebalancing buffer-emptying (non-full) takes $O(M/B)$ I/Os
    $\Rightarrow$ cost of one split/fuse $O(M/B)$ I/Os
  - During $N$ updates
    * $O(N/B)$ leaf split/fuse
    * $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ internal node split/fuse

$\downarrow$

Total cost of $N$ operations: $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os
Basic Buffer-tree

• **Emptying all buffers** after $N$ insertions:
  Perform buffer-emptying on all nodes in BFS-order
  \[ \Rightarrow \text{resulting full-buffer emptyings cost } O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) \text{ I/Os} \]
  empty \( O\left(\frac{N}{M/B}\right) \) non-full buffers using \( O(M/B) \) \(\Rightarrow\) \( O(N/B) \) I/Os

\[
\begin{array}{c}
\frac{1}{4} & M & B & \cdots & M & \frac{N}{B} \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\vdots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\end{array}
\]

\[\downarrow\]

• $N$ elements can be sorted using buffer tree in $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ I/Os
Buffer-tree Technique

- **Insert** and **deletes** on buffer-tree takes $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os amortized
  - Alternative rebalancing algorithms possible (e.g. top-down)
- One-dim. **rangesearch** operations can also be supported in $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B} + \frac{T}{B}\right)$ I/Os amortized
  - Search elements handle lazily like updates
  - All elements in relevant sub-trees reported during buffer-emptying
  - Buffer-emptying in $O\left(\frac{X}{B} + T'/B\right)$,
    where $T'$ is reported elements

- Buffer-tree can e.g. be use in standard plane-sweep algorithms for orthogonal line segment intersection (alternative to distribution sweeping)
Buffered Priority Queue

- Basic buffer tree can be used in external priority queue
- To delete minimal element:
  - Empty all buffers on leftmost path
  - Delete $\frac{1}{4}M$ elements in leftmost leaf and keep in memory
  - Deletion of next $M$ minimal elements free
  - Inserted elements checked against minimal elements in memory

- $O\left(\frac{M}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os every $O(M)$ delete $\Rightarrow O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ amortized
Other External Priority Queues

- External priority queue has been used in the development of many I/O-efficient graph algorithms

- Buffer technique can be used on other priority queue structure
  - Heap
  - Tournament tree

- Priority queue supporting update often used in graph algorithms
  - $O\left(\frac{1}{B} \log_2 \frac{N}{B}\right)$ on tournament tree
  - Major open problem to do it in $O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os

- Worst case efficient priority queue has also been developed
  - $B$ operations require $O\left(\log_{M/B} \frac{N}{B}\right)$ I/Os
Other Buffer-tree Technique Results

- Attaching $\Theta(B)$ size buffers to normal B-tree can also be used to improve update bound
- Buffered segment tree
  - Has been used in batched range searching and rectangle intersection algorithm
- Can normally be modified to work in D-disk model using D-disk merging and distribution
- Has been used on String B-tree to obtain I/O-efficient string sorting algorithms
- Can be used to construct (bulk load) many data structures, e.g:
  - R-trees
  - Persistent B-trees
Summary

• Fan-out $\Theta(B^{1/c})$ B-tree ($c \geq 1$)
  – Degree balanced tree with each node/leaf in $O(1)$ blocks
  – $O(N/B)$ space
  – $O(\log B N + T_B)$ I/O query
  – $O(\log B N)$ I/O update

• Persistent B-tree
  – Update current version, query all previous versions
  – B-tree bounds with $N$ number of operations performed

• Buffer tree technique
  – Lazy update/queries using buffers attached to each node
  – $O(\frac{1}{B} \log M/B \frac{N}{B})$ amortized bounds
  – E.g. used to construct structures in $O(\frac{N}{B} \log M/B \frac{N}{B})$ I/Os
Tomorrow

- “Dimension 1.5” problems: Interval stabbing and point location

- Use of tools/techniques discussed today as well as
  - Logarithmic method
  - Weight-balanced B-trees
  - Global rebuilding