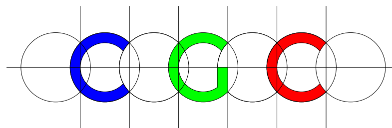
Computing on MASSIVE Data

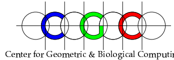
Jeff Vitter

Duke University

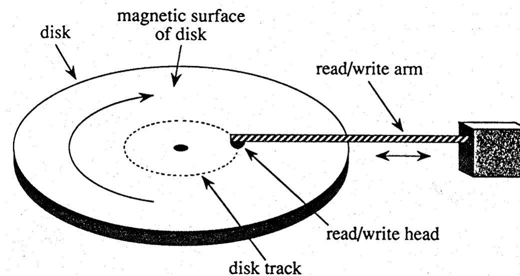


http://www.cs.duke.edu/CGBC/

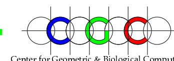
CPS 230



Magnetic Disk Drives as Secondary Memory

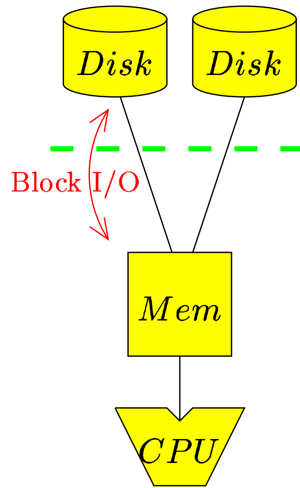


- ☆ I/O Crisis! Disk access is 1,000,000 times slower!
- ☆ Time for rotation ≈ Time for seek.
- ☆ Amortize search time by transferring large blocks so that Time for rotation ≈ Time for seek ≈ Time to transfer data.
- ☆ Solution 1: Exploit locality and take advantage of block transfer. *Focus of this talk!*
- ☆ Solution 2: Parallel disks. (Not our focus today.)



Parallel Disk Model

[Aggarwal & Vitter 88], [Vitter & Shriver 90, 94], ...



N = problem data size.

M = size of internal memory.

B = size of disk block.

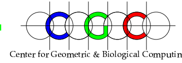
D = number of independent disks.

P = number of CPUs.

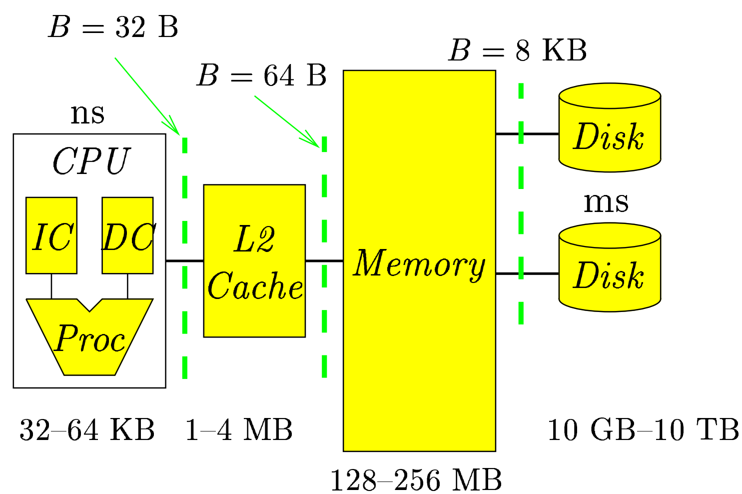
Z = problem output size.

Notational convenience (in units of blocks):

$n = \frac{N}{B}$, $m = \frac{M}{B}$, $z = \frac{Z}{B}$.



A "Real" Machine



Outline

1. Fundamental I/O Bounds.
   - ☆ Merge sort.
   - ☆ Distribution sort.
   - ☆ Searching.
   - ☆ Lower Bounds.
2. Techniques for solving batched geometric problems.
   - ☆ Distribution sweeping.
   - ☆ Red-blue orthogonal rectangle intersection.
   - ☆ Water flow routing and accumulation in terrains.
   - ☆ Empirical results (via TPIE programming environment).
3. Online data structures.
   - ☆ B-trees, buffer trees, R-trees, etc.
   - ☆ Range searching.

Fundamental I/O Bounds (with $D = 1$ disk)

☆ Batched problems [AV88], [VS90], [VS94]:

- Scanning (touch problem): $\Theta\left(\frac{N}{B}\right) = \Theta(n)$
- Sorting:
  $$\Theta\left(\frac{N}{B}\frac{\log \frac{N}{B}}{\log \frac{M}{B}}\right) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \Theta\left(n\log_m n\right)$$
- Permuting: $\Theta\left(\min\left\{N,\ n\log_m n\right\}\right)$

☆ For other problems [CGGTVV95], [AKL95], ...

- Graph problems $\asymp$ Permutation
- Computational Geometry $\asymp$ Sorting

☆ Online problems:

- Searching and Querying: $\Theta\left(\log_B N + \frac{Z}{B}\right) = \Theta(\log_B N + z)$

I/O Lower Bound for Permuting

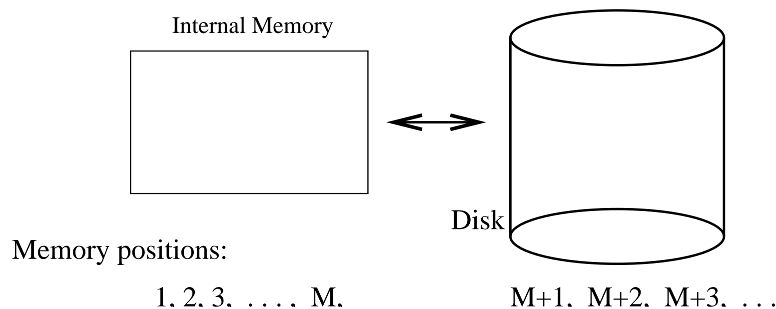
Permuting problem: Given N distinct items from $\{1, 2, \ldots, N\}$, rearrange the N items into sorted order.

- ☆ We will show the lower bound that permuting requires $\Omega(\min\{N, \ n \log_m n\})$ I/Os.
- ☆ Typically the min term is $n \log_m n$.
- ☆ Permuting is a special case of sorting.
- ☆ I/O lower bound also applies to sorting. *It is based only upon routing considerations, since the order is already known.*
- ☆ For the pathological case when $N < n \log_m n$, we can show that sorting requires $\Omega(n \log_m n)$ I/Os in comparison model.
- ☆ In the RAM model, permutation takes only $O(N)$ time. But in I/O model, it (and most interesting problems) require sorting complexity (except for pathological case)!

I/O Lower Bound for Permuting

Goal: See how many I/O steps T are needed so that any of the $N!$ permutations of the N items can be realized.

We say that a permutation is realizable if it appears in extended memory in the required order.



Tactic: Determine how much the tth I/O step can increase the number of possible realizable permutations.

I/O Lower Bound for Permuting

Assumption: the N items to permute are indivisible.

\# realizable permutations after tth I/O

$$= \begin{cases} \binom{M}{B} \times (\#\text{ realizable permutations after } (t-1)\text{st I/O}) \\ \qquad\qquad\qquad \text{if block was previously accessed} \\ \\ B! \times \binom{M}{B} \times (\#\text{ realizable permutations after } (t-1)\text{st I/O}) \\ \qquad\qquad\qquad \text{if this is first access to block} \end{cases}$$

There are N/B blocks initially unaccessed.

$$\#\text{ choices for block accessed in } t\text{th I/O} = \left(\frac{N}{B} + t\right) \leq N^2.$$

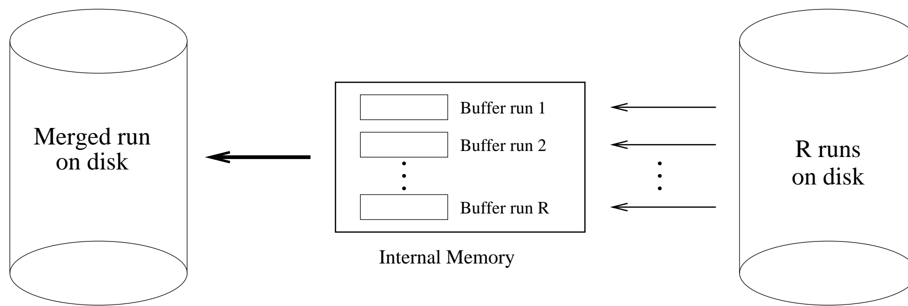
Number T of required I/Os for Permuting

$$(B!)^{N/B} \left(\binom{M}{B} N^2 \right)^T \geq N!$$

Taking logs and applying Stirling's approximation:

$$\begin{aligned}
\frac{N}{B} \log B! + T \left(\log \binom{M}{B} + \log N \right) &= \Omega(\log N!) \\
\frac{N}{B} (B \log B) + T \left(B \log \frac{M}{B} + \log N \right) &= N \log N \\
T \left(B \log \frac{M}{B} + \log N \right) &= N \log N - N \log B \\
&= N \log \frac{N}{B} \\
T &= \Omega \left(\min \left\{ N,\ \frac{N}{B} \frac{\log(N/B)}{\log(M/B)} \right\} \right) \\
&= \Omega \left(\min \{ N,\ n \log_m n \} \right)
\end{aligned}$$

Merge Sort

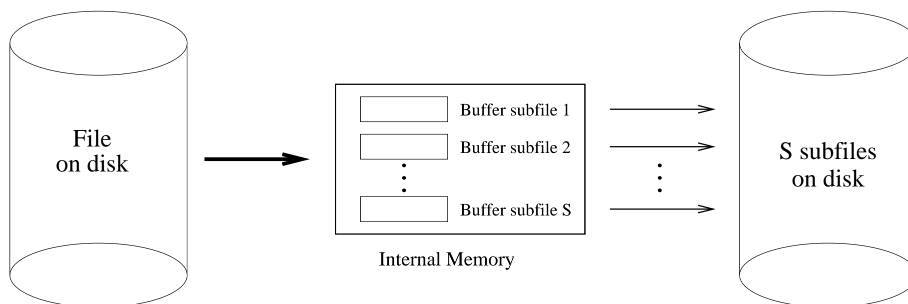


- ☆ Form n/m initially sorted runs, each consisting of m blocks (i.e., one memoryload).
- ☆ Reserve $R = \Theta(m)$ buffers, each of size B, in internal memory. Repeatedly merge together R runs at a time

  $\Longrightarrow$ # passes $= \log_R \frac{n}{m} = \log_m n - 1$
- ☆ Each pass uses $\Theta(n)$ I/Os

  $\Longrightarrow$ # I/Os $= O(n \log_m n)$.

Distribution (Bucket) Sort



- ☆ Select $S = \Theta(m)$ or $\Theta(\sqrt{m})$ partitioning elements that divide the file evenly into buckets. (See [AV88, §5].)
- ☆ Stream the file through internal memory, forming S subfiles.
- ☆ Independently sort the S subfiles recursively.

$\Longrightarrow$ number of levels of recursion is $\log_S n = \log_m n$.
each level of recursion uses $O(n)$ I/Os

$\Longrightarrow$ # I/Os $= O(n \log_m n)$.

Batched Problems in Geometry

[GTVV93], [AVV95], [APRSV98a], [APRSV98b], [CFMMR98]

- ☆ Orthogonal rectangle intersection.
- ☆ Red-blue line segment intersection.
- ☆ General line segment intersection.
- ☆ All nearest neighbors.
- ☆ 2-D and 3-D convex hulls.
- ☆ Batched range queries.
- ☆ Trapezoid decomposition
- ☆ Batched planar point location.
- ☆ Triangulation.

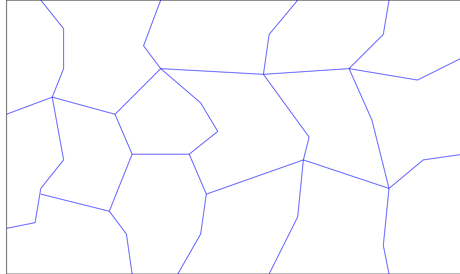
Use of virtual memory $\Longrightarrow$ $\Omega\big(N \log_B N + Z\big)$ I/Os. *Bad !!!*

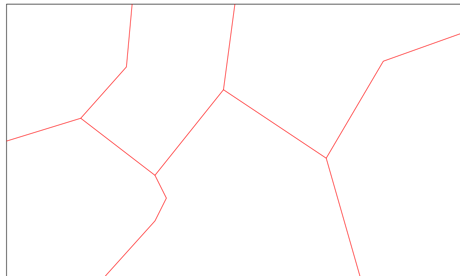
We can improve this to $O\big(n \log_m n + z\big)$ I/Os using

- ☆ Distribution sweep.
- ☆ Batched filtering.
- ☆ Random incremental construction.
- ☆ Parallel simulation.

Map Overlay/Spatial Join

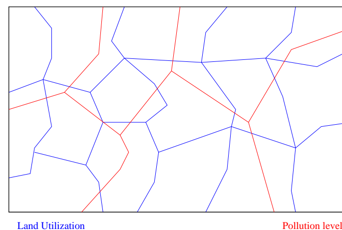
- ☆ Basis for some geographic information systems (GIS).
- ☆ Example: Find all farmland having a level of pollution over a certain threshold.


Land Utilization


Pollution level

☆ Red-blue line segment intersection important subproblem.

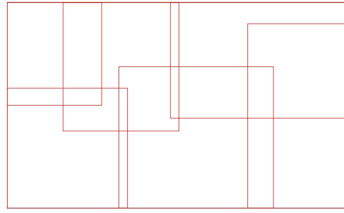
Map Overlay/Spatial Join



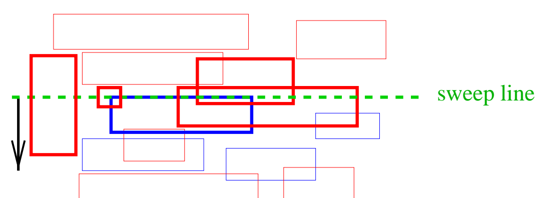
☆ In database literature often solved in two steps:

- Filter step: Compute minimal bounding rectangles for each region and compute intersections between rectangles from different maps (red-blue rectangle intersection).
- Refinement step: Validate intersections.

☆ We consider filter step: We focus on the case where input is "unordered" (not indexed). Occurs e.g. when input is intermediate result.



Red-Blue Rectangle Intersection

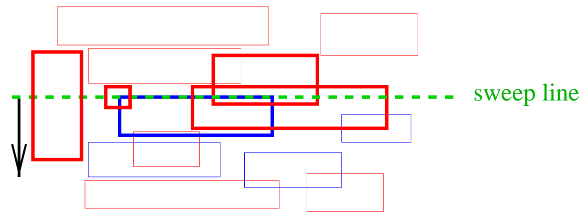


☆ Sweep plane while maintaining two active lists of red and blue rectangles intersecting vertical sweep line [BW80]:

- When top of blue rectangle is reached:
  (i) Insert blue rectangle in blue active list.
  (ii) Find intersections with rectangles in red active list.
- When bottom of blue rectangle is reached:
  (i) Remove rectangle from blue active list.

☆ Red rectangles are handled similarly.

Red-Blue Rectangle Intersection



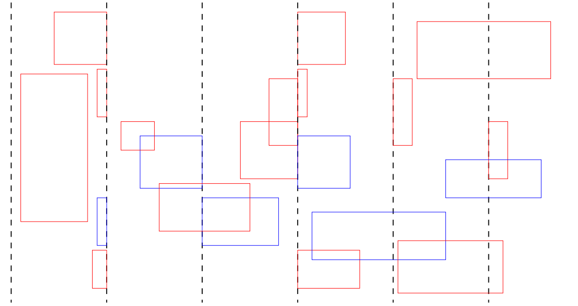
☆ Algorithm performs badly ($> N$ I/Os) if size of active lists $> M$.

☆ Solved in optimal $O(n \log_m n + z)$ I/Os using general method for solving Batched Dynamic Problems.

☆ Sequence of operations $a_1, a_2, \ldots, a_N$ known beforehand. (a_i is `Insert`, `Delete` or `Query`.)

☆ Key point: Updates and queries are batched!

Sketch of External Solution [APRSV98]:



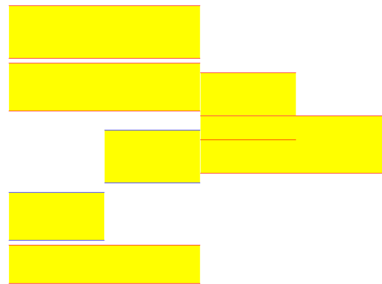
1. Divide plane into $\sqrt{m}$ slabs, each with $O(N/\sqrt{m})$ endpoints.
2. Find Z' intersections involving the part of a rectangle completely spanning slabs.
3. Recursively solve problem in each slab.

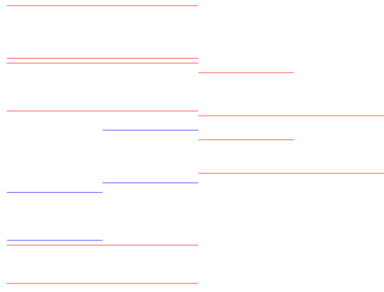
☆ $O(\log_{\sqrt{m}} n) = O(\log_m n)$ levels of recursion.

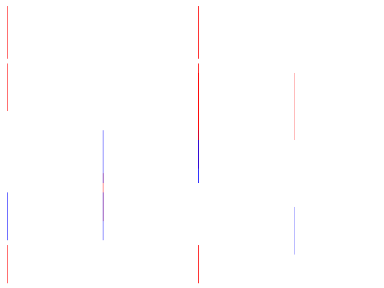
☆ Performing Step 2 in $O\left(n + \frac{Z'}{B}\right)$ I/Os

$\Longrightarrow \quad O(n \log_m n + z)$ I/Os total.

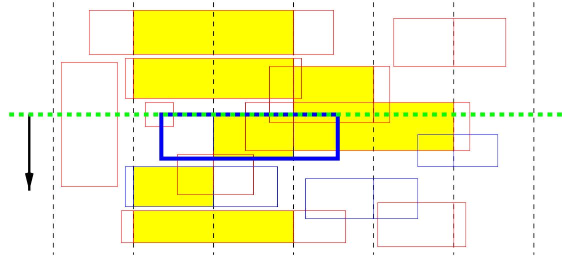
Center for Geometric & Biological Computing





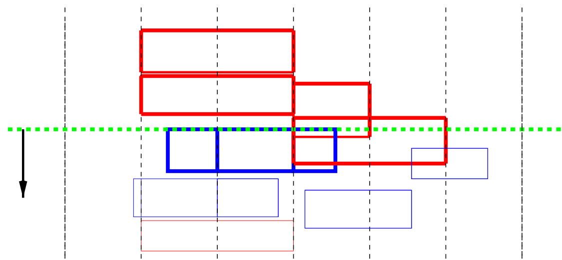


Key Idea



- ☆ Use $\sqrt{m}$ slabs
- ☆ $\Longrightarrow$ $O(m)$ multislabs (continuous ranges of slabs)
- ☆ $\Longrightarrow$ Can keep first B rectangles per multislab in internal memory.
- ☆ Perform top down sweep:
  - Maintaining active list for each multislab.

Sketch of Sweep



- ☆ Intersections between red centerpieces and tops of blue rects.:
  - At red rectangle: Insert into relevant multislab list.
  - At blue rectangle: Scan through all relevant multislab lists of red rectangles.
    (i) Report intersection with "non-expired" red rectangles.
    (ii) Remove "expired" red rectangles ("lazy" deletion).
- ☆ Other cases handled similarly—in one sweep!

Higher Dimensions

- ☆ Technique can be used recursively in dimension $d > 2$ by decreasing number of slabs to $m^{1/2(d-1)}$.
- ☆ I/O performance using technique:
  - d-dim. batched range searching:
    $O(n \log_m^{d-1} n + t)$ I/Os, $O(n)$ space.
  - d-dim. rectangle intersection:
    $O(n \log_m^{d-1} n + t)$ I/Os, $O(n)$ space.
  - Batched semidynamic planar point location:
    $O((n + k) \log_m^2 (n + k))$ I/Os, $O(n + k)$ space.

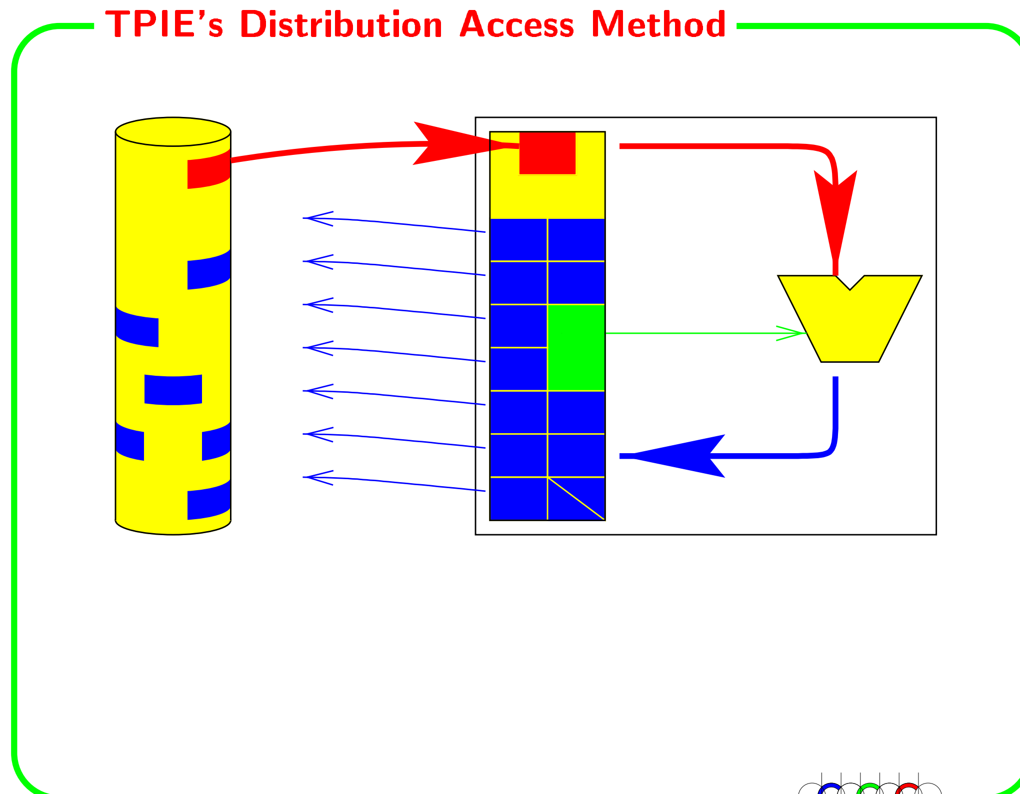
TPIE, http://www.cs.duke.edu/TPIE/

- ☺ Many problems can be solved using small number of paradigms.
- ☹ OS often provides inadequate support for I/O and internal memory management.

☆ TPIE originally designed by former student Darren Vengroff:

- Make implementation easy (and portable). I/O-efficient (and portable) programs.
- Framework oriented: Implements a number of high-level paradigms on streams (C++)
  —Scanning, merging, distribution, sorting, permuting, ...
- Access-Oriented: For index structures.

TPIE's Distribution Access Method

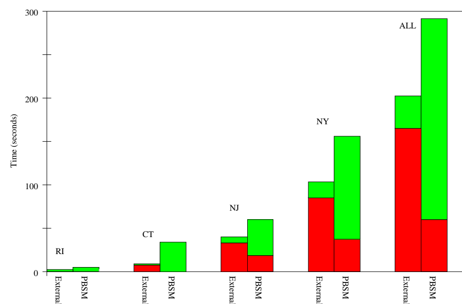


TIGER/Line Data

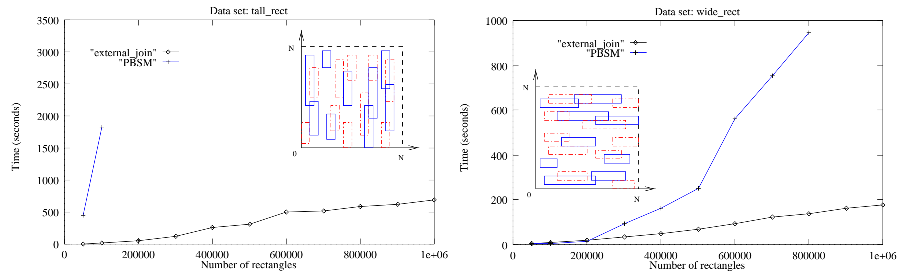
☆ TIGER/Line data from U.S. Census Bureau
(standard benchmark data for spatial databases)

State	Category	Size	Objects
Rhode Island (RI)	Roads	4.3 MB	68,278
	Hydrography	0.4 MB	7,013
Connecticut (CT)	Roads	12.0 MB	188,643
	Hydrography	1.8 MB	28,776
New Jersey (NJ)	Roads	26.5 MB	414,443
	Hydrography	3.2 MB	50,854
New York (NY)	Roads	55.7 MB	870,413
	Hydrography	10.0 MB	156,568
All	Roads	98.5 MB	1541,777
	Hydrography	15.4 MB	243,211

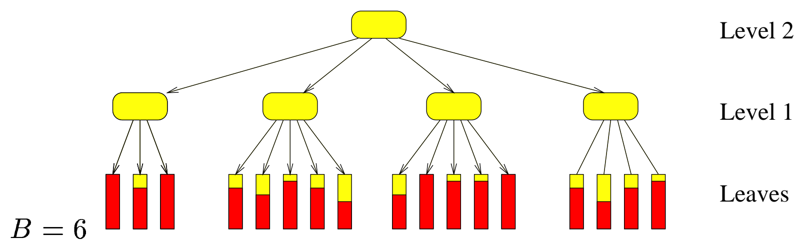
Performance Comparison with PBSM [DP96]



Sun SparcStation 20 (Solaris 2.5) , 32MB memory (TPIE 12MB)



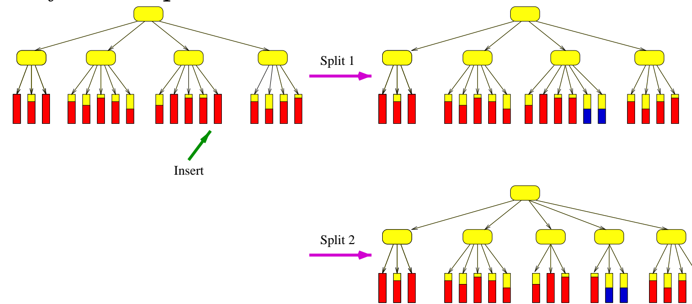
B-Trees [Bayer-McCreight72]



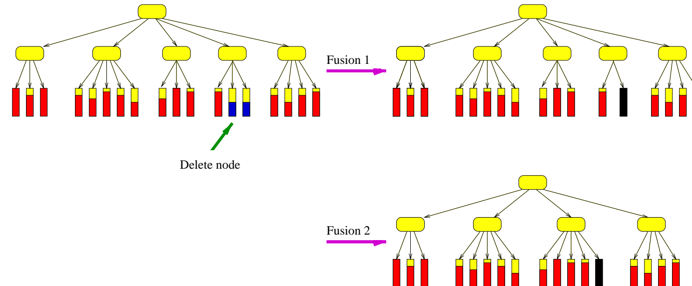
- ☆ Each node is stored in one disk block.
- ☆ Leaf nodes store all the data items.
- ☆ Internal nodes store keys and pointers to guide the searching.
- ☆ $\frac{B}{2} \leq$ node degree $\leq B - 1$, except for the root.
- ☆ One I/O per level
  $\Longrightarrow \quad O(\log_B N + z)$ I/Os for a 1-D range query.

B-Trees—Updates in $O(\log_B N)$ I/Os

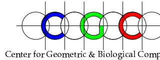
Insertion may cause splits:



Deletions may cause nodes to combine:



Sharing also possible (good packing).



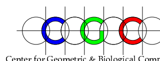
Batched Dynamic Processing

Motivation: Sometimes we may want to handle inserts, deletes, and queries in batches. In RAM algorithms, we can use an optimal dynamic data structure and get optimal results.
But NOT in external memory!!!

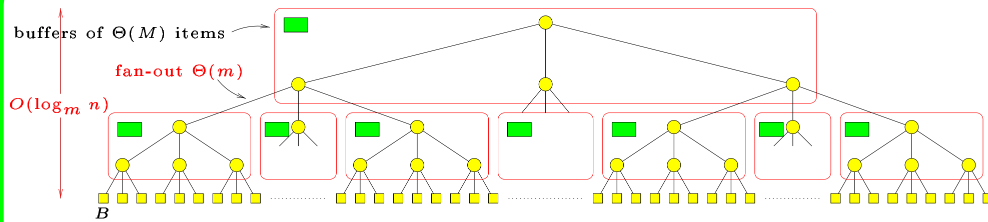
Example: Sorting by inserting and deleting into a priority queue.

- ☆ Insert items one at a time into priority queue.
- ☆ Repeat: Find item in priority queue with minimum key. Delete it from priority queue (*delete_min* operation) and output it.
- ☆ N inserts into priority queue
  $\Longrightarrow$ $O(N \log_B N)$ I/Os. *Bad !!!*
- ☆ We want $O(n \log_m n)$ I/Os. *Good !!!*

Another example: Sweep-line algorithm for finding rectangle intersections.



The Buffer Tree [Arge95]



Main idea: Combine degree-B nodes to form supernodes of degree $m = M/B$. Each supernode gets a buffer of size M.

☆ Insertions are done "lazily": Items are inserted into buffers. When a buffer runs full, its items are pushed one level down.

☆ Buffer-emptying of M items takes $O(m)$ I/Os

$\Longrightarrow$ $O(m/M) = O(1/B)$ I/Os per item per level

$\Longrightarrow$ N items inserted in $O\left(\frac{N}{B}\log_m n\right) = O(n \log_m n)$ I/Os.

Online Search

B-trees are an optimal data structure for the dictionary problem and one-dimensional range queries:

☆ $O(\log_B N + z)$ I/Os per query.
☆ $O(\log_B N)$ I/Os per update.
☆ $O(n)$ disk blocks of space.

We want similar performance for harder problems:

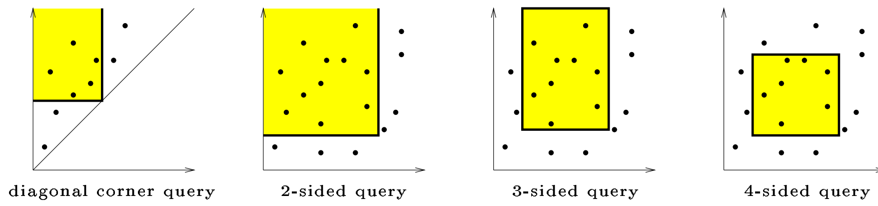
☆ Proximity queries, nearest neighbor, clustering
☆ Point location, ray shooting
☆ Range searching in 2-D, 3-D, orthogonal, halfspace

Two very useful paradigms:

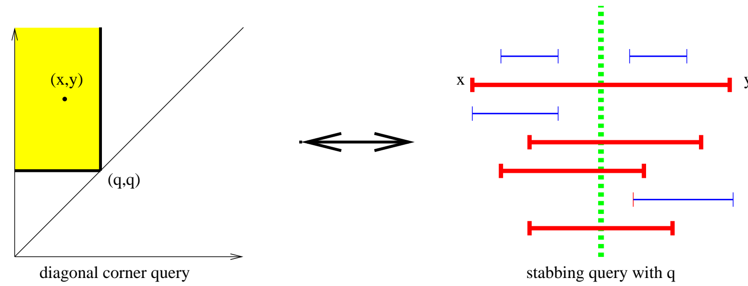
1. Filtering [Chazelle86]: You can afford to do an extra I/O during search if you can charge it to $\Theta(B)$ query outputs.
2. Bootstrapping [AV96, ASV99]
   ☆ Use an external version of an efficient internal-memory data structure as the global search structure.
   ☆ Within the structure, use efficient static data structures on smaller instances (of size $O(B^2)$) of the problem.

Online Range Searching

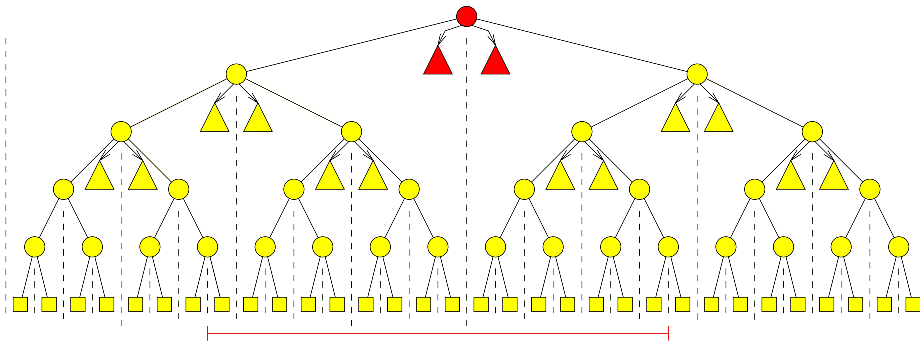
Applications: geographic information systems (GIS), spatial databases, indexing for new data models (constraint, temporal, object-oriented).



For example, indexing constraints in constraint query languages:



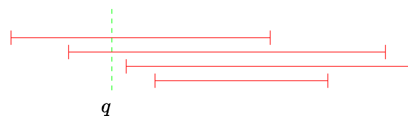
Internal Interval Tree (fixed endpoint set)



☆ An interval X_v is associated with each node v.

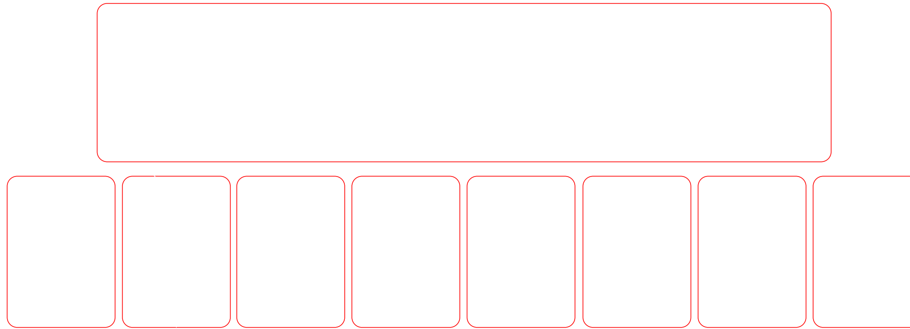
☆ A segment is stored in the highest node v where it contains the midpoint of X_v.

☆ Segments in node v are stored in two secondary search trees, sorted according to left and right endpoint, respectively $\Longrightarrow$ *$O(N)$ space* and *$O(\log_2 N)$-time update* bounds.



☆ Query with q on left side of midpoint of X_{root}:

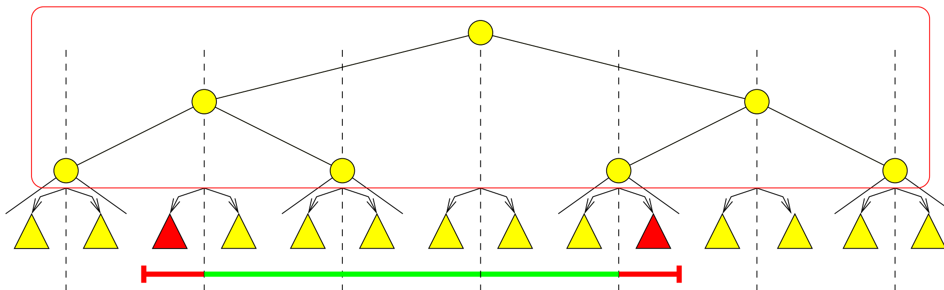
- Search from left-to-right in secondary search tree sorted according to left endpoint.
- Recurse in left son $\Longrightarrow$ *$O(\log_2 N + Z)$-time query* bound.



☆ Number of *stabbed* segments in a secondary structure may be small (or zero).

- We can be forced to use an I/O in each of $O(\log_2 N)$ nodes without reporting anything.

External Interval Tree



☆ Decrease fan-out to $\sqrt{B}$. Height remains $O(\log_B N)$.

☆ $\sqrt{B}$ "slabs" define $O(\sqrt{B}^2) = O(B)$ "multislabs".

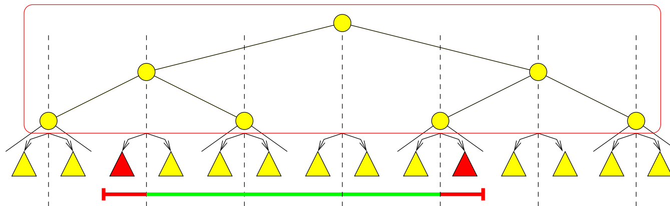
☆ Secondary list structure (B-tree) for each of the $2 \cdot \sqrt{B}$ slabs and $O(B)$ multislabs.

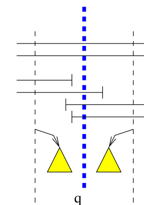
☆ Interval stored in two slab lists and (maybe) one multislab list $\Longrightarrow$ *$O(n)$ space and $O(\log_B N)$ I/Os per update.*

External Interval Tree

☆ Need weight-balanced B-trees to remove fixed-endpoint assumption. Rebalancing node v requires $O\big(\mathit{weight}(v)\big)$ I/Os.

Query in External Interval Tree





Search for q

☆ Search down the structure and in every of the $O(\log_B N)$ nodes:

- "Search" in two slab lists: $2 \cdot O(z + 1)$ I/Os.
- Report all segments in relevant multislab lists

— $O(z+B)$ *!!!*

— $O(z)$.

—The segments for any multislab containing less than $B/2$ segments are collectively stored in the node's "corner structure" (of total size $O(B^2)$). We avoid looking at those multislabs by instead querying the corner structure — $O(z+1)$.

$\Longrightarrow$ *Optimal query bound of* $O(\log_B N + z)$ *I/Os.*

Corner Structure of $O(B^2)$ points

Corner Structure: A set of $k \le B^2$ segments can be represented using $O(k/B)$ blocks such that a stabbing query can be answered in $O(z+1)$ I/Os [KRVV93].

- ☆ The corner structure can be made dynamic with $O(1)$ amortized update bound: Keep a special update block and rebuild the structure after B updates have been performed [KRVV93].
- ☆ Amortization can be removed by rebuilding the corner structure lazily.

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]. *Optimal !*
- ☆ *2-sided and 3-sided:* $O(n)$ space, $O(\log_B N + z)$ I/Os query, $O(\log_B N)$ I/Os update [ASV99]. *Optimal !*
- ☆ *4-sided:* $O\big(n(\log n)/\log\log_B N\big)$ space, $O(\log_B N + z)$ I/Os query, $O\big((\log_B N)(\log n)/\log\log_B N\big)$ I/Os update [ASV99]. *Optimal, except update!*
- ☆ Lower bounds: Cannot simultaneously achieve $O\big(n(\log n)/\log\log_B N\big)$ space & $O\big((\log_B n)^c + z\big)$ I/Os query [SR95].
- ☆ Three-dimensional range queries: $O\big(n(\log n)^{k+1}/(\log(\log_B N))^k\big)$ space for $(3+k)$-sided 3-D queries, $O(\log_B N + z)$ I/Os query [VV96, Vit99].
- ☆ Halfspace queries: $O(n)$ space, $O(\log_B n + z)$ I/Os query [AAEFV99]. *Static: optimal !*
- ☆ Open problems: Higher dimensions, typical-case analysis, restricted sets of queries, non-orthogonal searching, point location, etc.

Conclusions and Open Problems

☆ *Répertoire of useful paradigms (distribution, merging, distribution sweeping, persistence, parallel simulation, B-trees, external interval tree, external priority search tree) for important problems.*

- Worst-case optimality requires overhead.
- Simpler versions are practical!
- Building blocks for external data structures (e.g., interval tree allows brute force rebuilding)

☆ *Lots of open problems in the design and analysis of external memory algorithms and data structures!*

- Lower bounds without indivisibility assumption.
- New models: hierarchical memory, caching, MEMS, optical storage, ...
- TPIE, see http://www.cs.duke.edu/TPIE/ partitioning elements, large fanouts. (Don't use square root trick.)

Conclusions and Open Problems

- Fundamental graph problems (e.g. topological sorting, BFS, DFS, connectivity).
- Online dynamic data structures (e.g. dynamic point location, range search in higher dimensions, clustering, similarity search).
- GIS applications (e.g. practical red-blue line segment intersection, spatial join, terrain processing).
- String processing, molecular databases.
- Typical-case behavior of popular data structures (e.g., R-trees).
- . . .