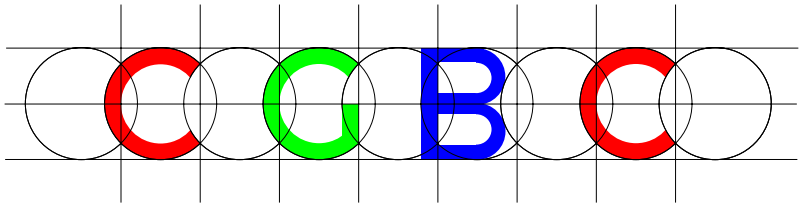
Batched Dynamic Geometric Problems

Jeff Vitter

Duke University

Center for Geometric and Biological Computing
and Department of Computer Science



Center for Geometric & Biological Computing
http://www.cs.duke.edu/CGBC/

July 2002

Outline

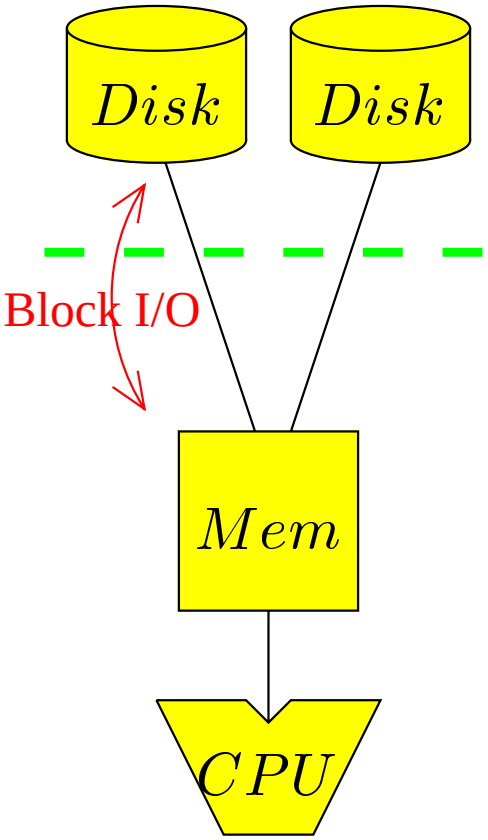
- ☆ Fundamental Techniques for batched problems.
  - Merge sort, distribution sort.

$\Longrightarrow$ Techniques for solving batched geometric problems.

  - Distribution sweeping, batched filtering, randomized incremental construction, parallel simulation.
  - Red-blue orthogonal rectangle intersection, convex hull, range search, nearest neighbors.
  - Empirical results (via TPIE programming environment).
- ☆ Fundamental lower bounds.
  - Sorting, permuting, FFT, matrix transposition, bundle sort.
  - Dynamic memory allocation
  - Hierarchical memory.
- ☆ Parallel disks.
  - Load balancing among disks is key issue.
  - Duality: reading (prefetching) $\longleftrightarrow$ writing, merging $\longleftrightarrow$ distribution

Review of 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.

Q = number of queries.

Z = problem output size.

Notational convenience (in units of blocks):

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

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

Batched Problems in Geometry

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

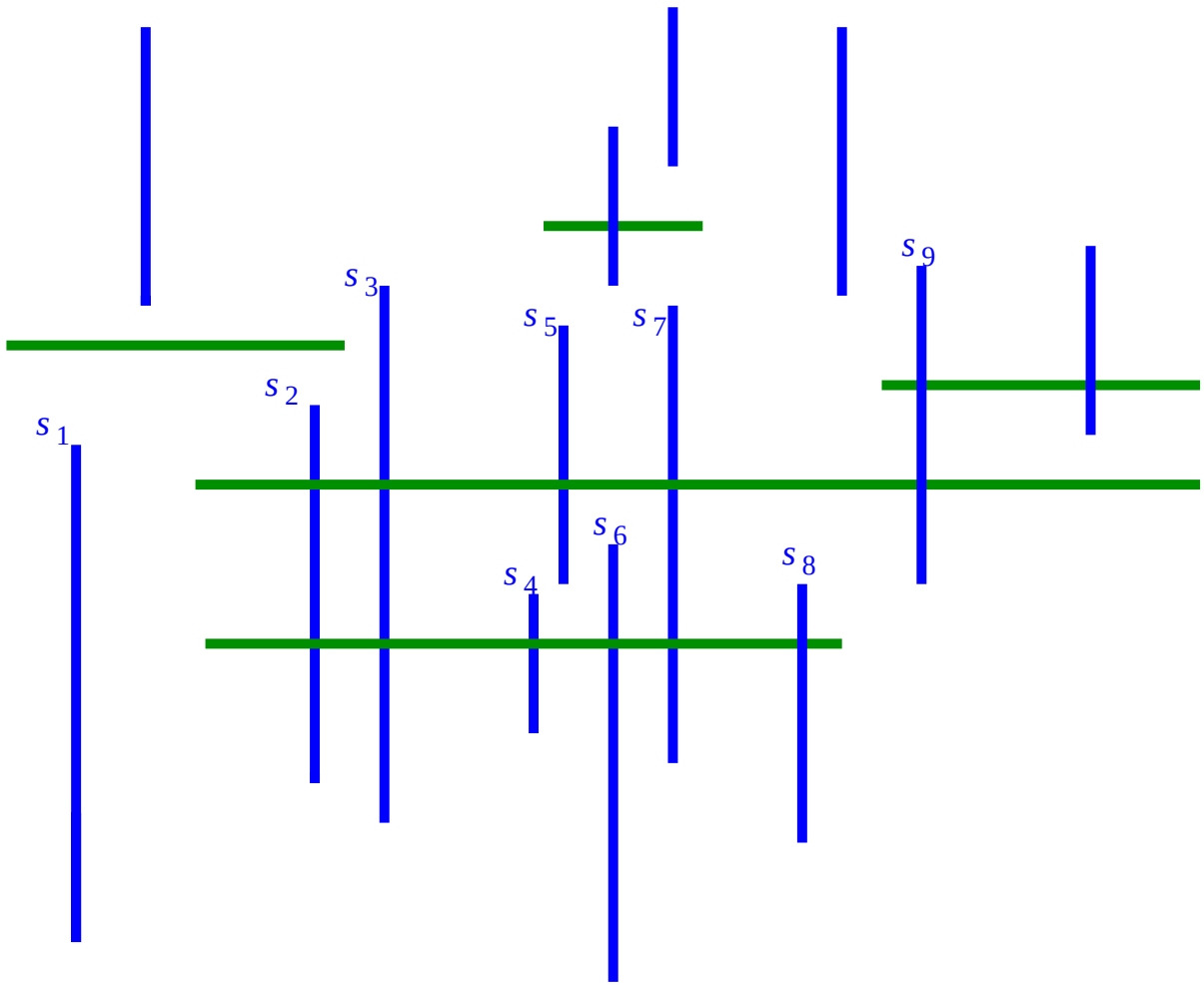
- ☆ Orthogonal rectangle intersection.
- ☆ Red-blue line segment intersection.
- ☆ General line segment intersection.
- ☆ All nearest neighbors.
- ☆ 2-D and 3-D convex hulls.
- ☆ Batched range queries.
- ☆ Trapezoid decomposition
- ☆ Batched planar point location.
- ☆ Triangulation.

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

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

- ☆ Distribution sweep.
- ☆ Persistent B-trees and batched filtering.
- ☆ Random incremental construction.
- ☆ Parallel simulation.

Orthogonal Line Segment Intersection

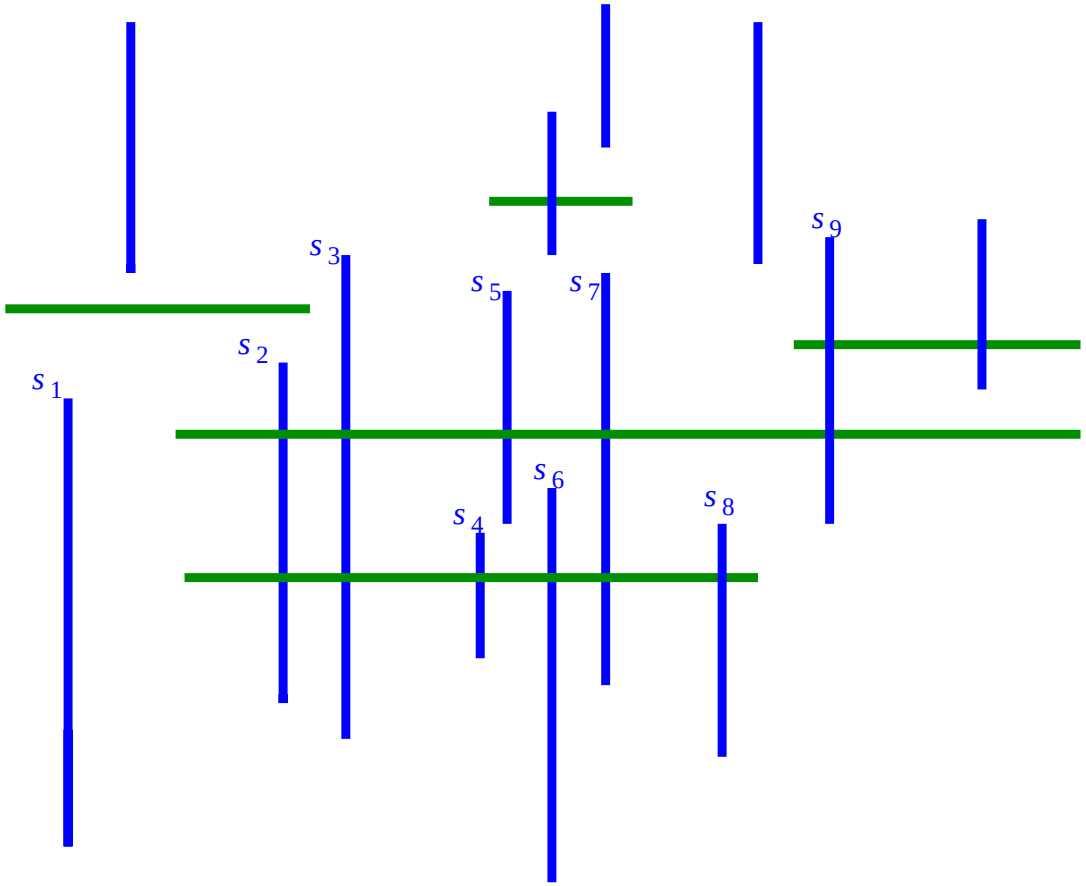


Problem: Find all intersections of vertical segments with horizontal segments.

Internal Memory Approach

- ☆ Presort the endpoints in y-order.
- ☆ Sweep the plane from top to bottom with a horizontal line.
- ☆ When reaching a vertical segment, store its x value in a balanced tree. When leaving a vertical segment, delete its x value from the tree.
- ☆ At any given time, the balanced tree stores the vertical segments hit by the sweep line.
- ☆ When reaching a horizontal segment, do a 1-d range query in the tree to find intersections with vertical segments. Time is $O(\lg N + Z')$, where Z' is number of intersections reported.
- ☆ Total running time is $O(N \lg N + Z)$.

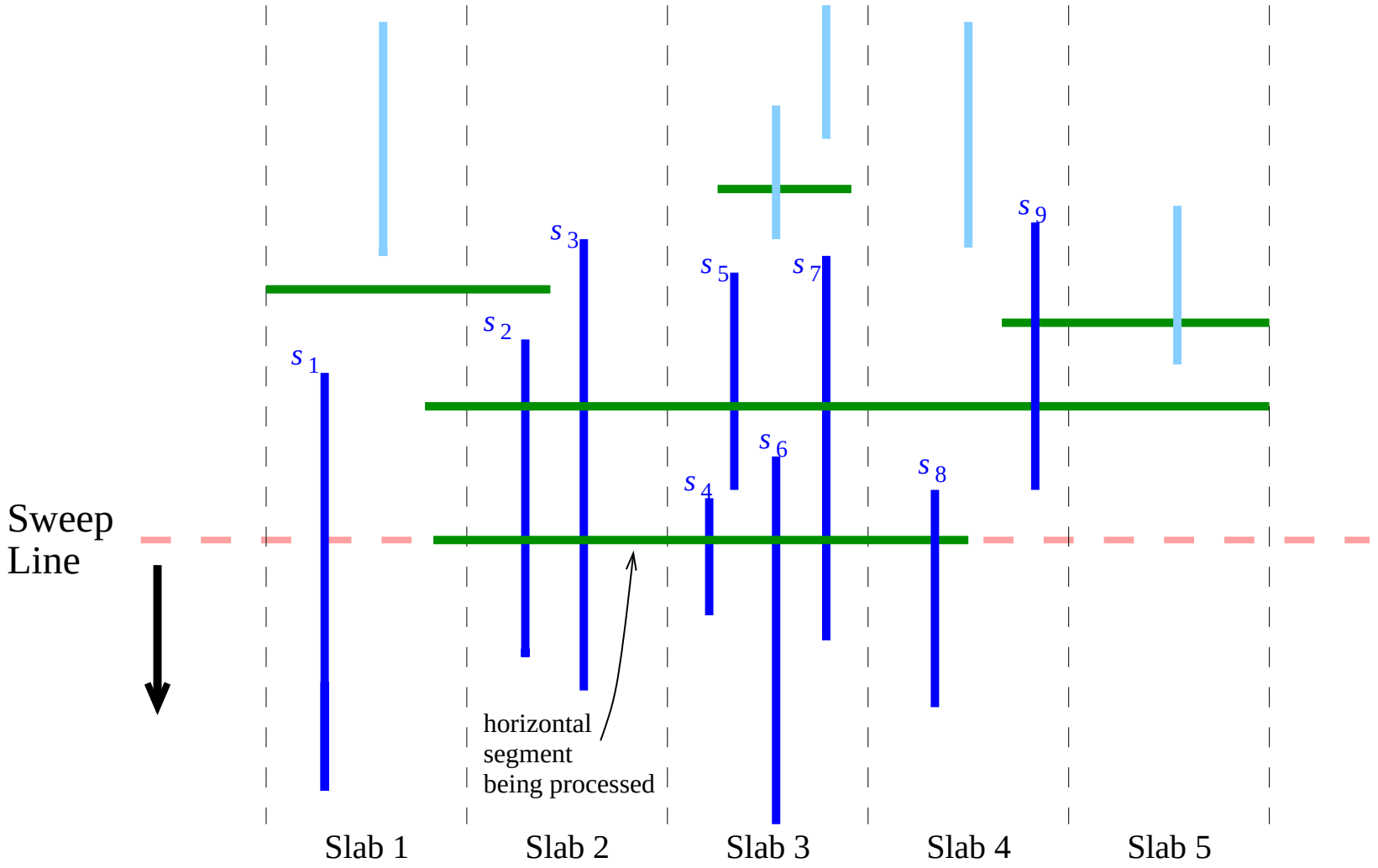
External Solution?



- ☆ Internal plane-sweep solution runs in $O(N \log N + Z)$ time.
- ☆ Using B-tree gives an $O(N \log_B n + z)$ I/O solution.
- ☆ We want an $O(n \log_m n + z)$ I/O solution that takes advantage of batching!

Distribution Sweeping

[Goodrich, Tsay, Vengroff & Vitter 93]



Distribution Sweeping

- ☆ Presort endpoints by x and y coordinates.
- ☆ Divide the x-range into $\Theta(m)$ *slabs*, so that each slab contains the same number of x values of vertical segments.
- ☆ Sweep all slabs simultaneously from top to bottom, keeping the vertical segments of a slab in a stack.
- ☆ For each slab spanned by a horizontal segment, output all "living" vertical segments in the slab's stack and delete all "dead" vertical segments from stack.
- ☆ For the left and right "endpieces" of a horizontal segment, that stick out into a slab but don't completely span it, handle those intersections recursively for each slab.

Implementing a Stack

Various stack operations:

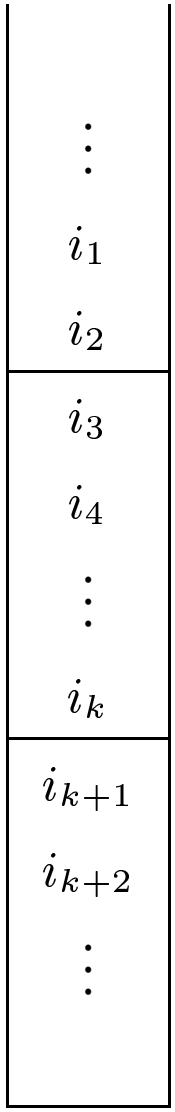
1. Push element onto top
2. Read top entry,
3. Pop entry from top.

Variants: We can read the top k entries from the stack by iterating operation 3 k times and then operation 1 k times.

Keep current block and one other in internal memory (using LRU).

It takes $O(B)$ pushes or pops to require one I/O.

$\Longrightarrow$ # I/Os per operation $= O\left(\frac{1}{B}\right)$ amortized.



Analysis of External Distribution Sweeping

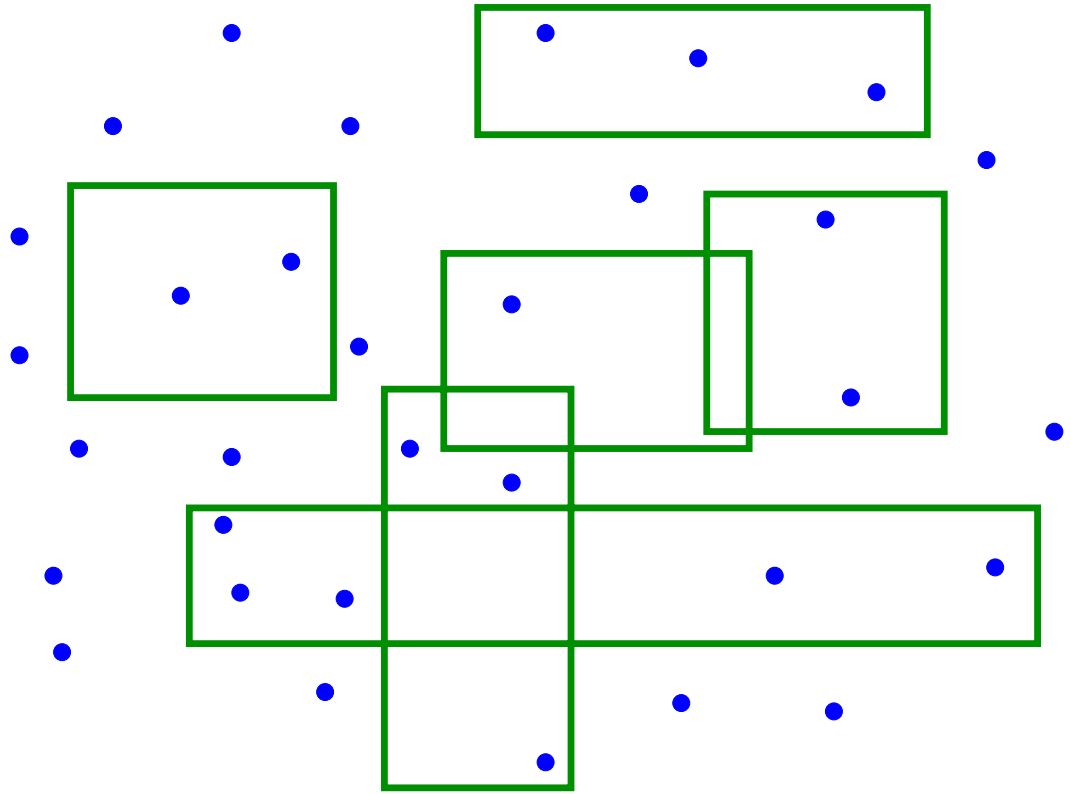
- ⋆ Each of the $\Theta(m)$ stacks can use $O(1)$ blocks in internal memory.
- ⋆ Therefore, each push, pop, or read uses $\left(\frac{1}{B}\right)$ I/Os amortized.
- ⋆ In each pass, the $O(N)$ vertical segments are inserted into the stack in $O(n)$ I/Os.
- ⋆ For each of the $O(N)$ horizontal segments, we report intersections in the slabs it completely spans. If the total number of intersections reported in this pass is Z', the number of I/Os is $O(n)$ plus the cost of Z' stack push, pop, or read operations, which is $O(n + Z'/B)$.

Analysis of External Distribution Sweeping

- ☆ We recurse on each of the $\Theta(m)$ slabs to handle the left endpieces and right endpieces of the horizontal segments.
- ☆ Note that the total number of endpieces at every level of recursion is at most $2\times$ # horizontal segments.
  It doesn't double at each level.
- ☆ Number levels of recursion is $O(\log_m n)$.
- ☆ Final result: $O(n \log_m n + z)$ I/Os.

Class Quiz

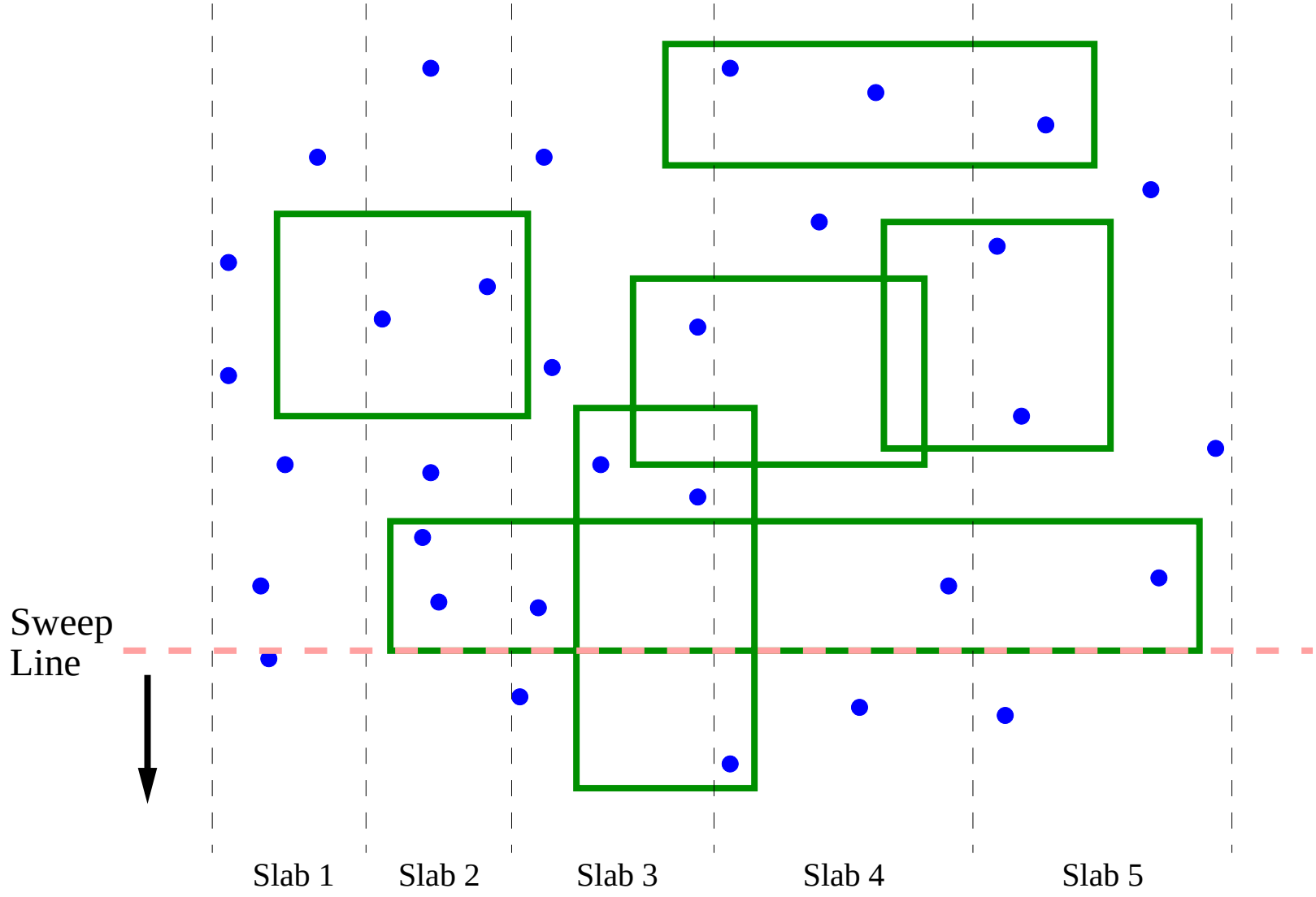
What about batched range searching?



We want to be able to do Q range queries on N points in $O((n+q)\log_m n + z)$ I/Os.

Ideas???

Distribution Sweeping to the Rescue



Distribution Sweeping

- ☆ Presort points on x and y coordinates.
- ☆ Presort the bottom horizontal sides of the query rectangles by their y coordinate.
- ☆ Sweep all slabs simultaneously from top to bottom, keeping the points of each slab in a stack.
- ☆ For each slab spanned by a bottom horizontal side, traverse its stack.
- ☆ Recursively handle the left endpiece and the right endpiece.

Analysis of Distribution Sweeping

- ☆ Each sweep uses $O\left(n + q + \frac{Z'}{B}\right)$ I/Os.
- ☆ In each pass, the points are inserted into the stacks in $O(n)$ I/Os.
- ☆ For each query rectangle, we report the points that are both inside the rectangle and inside the slab spanned by the rectangle. If the total number of points reported in this pass is Z', the number of I/Os is $O(q + Z'/B)$.
- ☆ We recurse in each of the $\Theta(m)$ slabs to handle the left endpieces and right endpieces of the query rectangles.
- ☆ The total number of endpieces at every level of recursion is at most $2Q$.
- ☆ Recursion levels: $O(\log_m n)$.
- ☆ Final result: $O((n + q)\log_m n + z)$ I/Os.

Output-Sensitive Convex Hulls

- ☆ Goal: Compute the convex hull in $T(N, H) = O(n \log_m \lceil h \rceil + n)$ I/Os, where $H = hB$ is the size of the convex hull.
- ☆ Motivation: H is often $\ll N$.
- ☆ Follow internal memory approach of [Kirkpatrick-Seidel].
- ☆ We no longer have time to sort by x coordinate for a distribution sweep.
- ☆ We can avoid the need to presort by x coordinate and can instead do the partitioning into slabs using the partitioning method described earlier.
- ☆ Cost is $O(n)$ I/Os to do the partitioning.
- ☆ But the number of slabs needs to be smaller: $O(\sqrt{m})$. But that's OK: # levels of recursion is still $O(\log_m h)$.

Output-Sensitive Convex Hulls

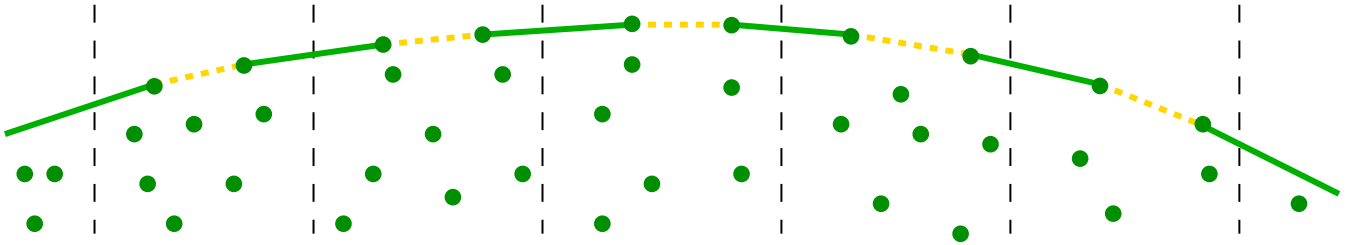
Main Ideas:

1. Apply Partitioning Lemma. Each of the $S = \sqrt{m}$ slabs has between $\frac{3N}{4S}$ and $\frac{5N}{4S}$ points.
2. Find hull edges crossing dividers in $O(n)$ I/Os (à la [Goodrich]).
3. Recurse only when needed.

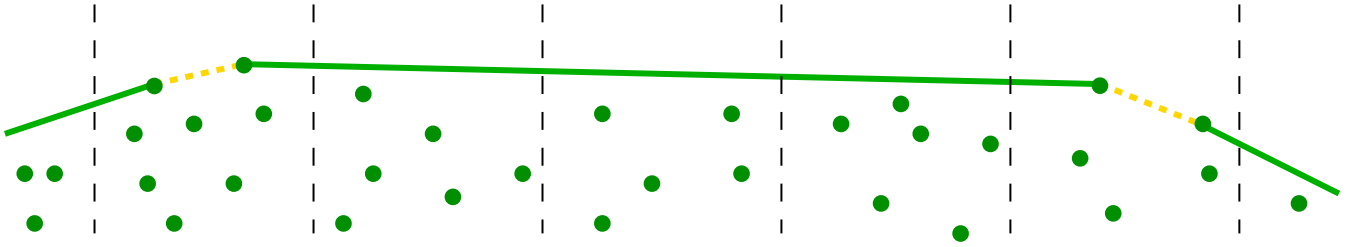
Result: $O(n \log_m \lceil h \rceil + n)$ I/Os.

Analysis: Assuming Step 2 requires $O(n)$ I/Os, each recursive call

- ☆ either finds more than $\sqrt{m}/2$ edges



- ☆ or it eliminates $N/2$ points.



Proof that $T(N, H) \leq cn \log_m \lceil h \rceil + n$

Divide-and-conquer gives $T(N, H) = \sum_i T(N_i, H_i) + n$.

By convexity, the worst case is when each $H_i = \frac{N_i}{N} H_i$,
which is between $\frac{3}{4}\frac{H}{\sqrt{m}}$ and $\frac{5}{4}\frac{H}{\sqrt{m}}$.

Case 1: $\left\lceil \frac{5}{4}\frac{H}{B\sqrt{m}} \right\rceil \leq \frac{2H}{B\sqrt{M}}$. By D-and-C and induction hypothesis,

$$\begin{aligned}
T(N, H) &\leq \sum_i \left(\frac{cn}{\sqrt{m}} \log_m \left\lceil \frac{2H}{B\sqrt{m}} \right\rceil + n_i \right) + n \\
&\leq cn \log_m \frac{2H}{B\sqrt{m}} + \sqrt{m} + n + n \\
&\leq cn \log_m \frac{H}{B} + cn \log_m 2 - \frac{cn}{2} + 2n + \sqrt{m} \\
&\leq cn \log_m h + n,
\end{aligned}$$

assuming $m > 4$ and c is large enough s.t.
$cn \log_m 2 - cn/2 + 2n + \sqrt{m} \leq n$.

Proof that $T(N, H) = O(n \log_m \lceil h \rceil + n)$

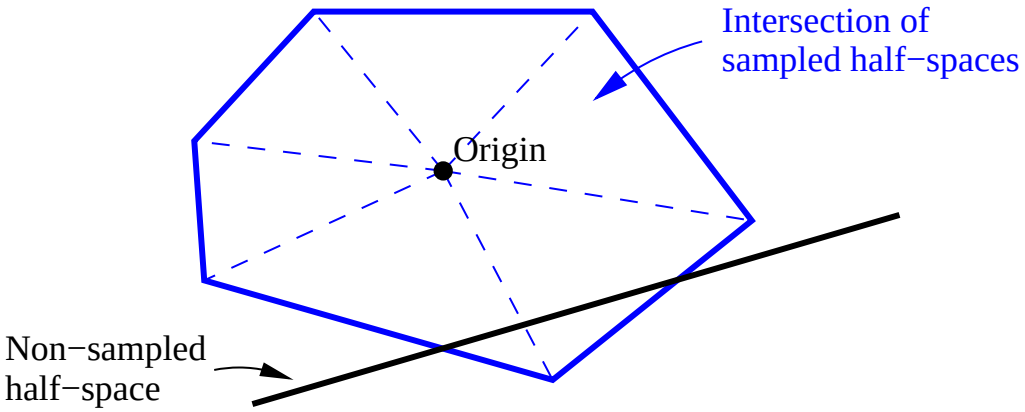
Case 2: $\frac{5}{4} \frac{H}{B\sqrt{m}} \leq 1$.

By divide-and-conquer and induction hypothesis,

$$\begin{aligned} T(N, H) &\leq \sum_i \left(\frac{cn}{\sqrt{m}}(0) + n_i \right) + n \\ &= 2n. \end{aligned}$$

3-d Convex Hull [Goodrich-Tsay-Vengroff-Vitter]

- ☆ Plane sweep and disribution sweep don't seem applicable.
- ☆ Instead we use externalization of randomized construction of [Reif-Sen] to compute 3-d convex hulls .
- ☆ Idea: Use random sampling in the dual problem (intersecting half-spaces containing origin).
- ☆ Take $O(\log_m n)$ samples of $S = N^\epsilon$ half-spaces and recursively compute intersection of each sample.
- ☆ For each sample, construct (triangulated) "cones" formed from origin to faces and find cones hit by the N input half-spaces.



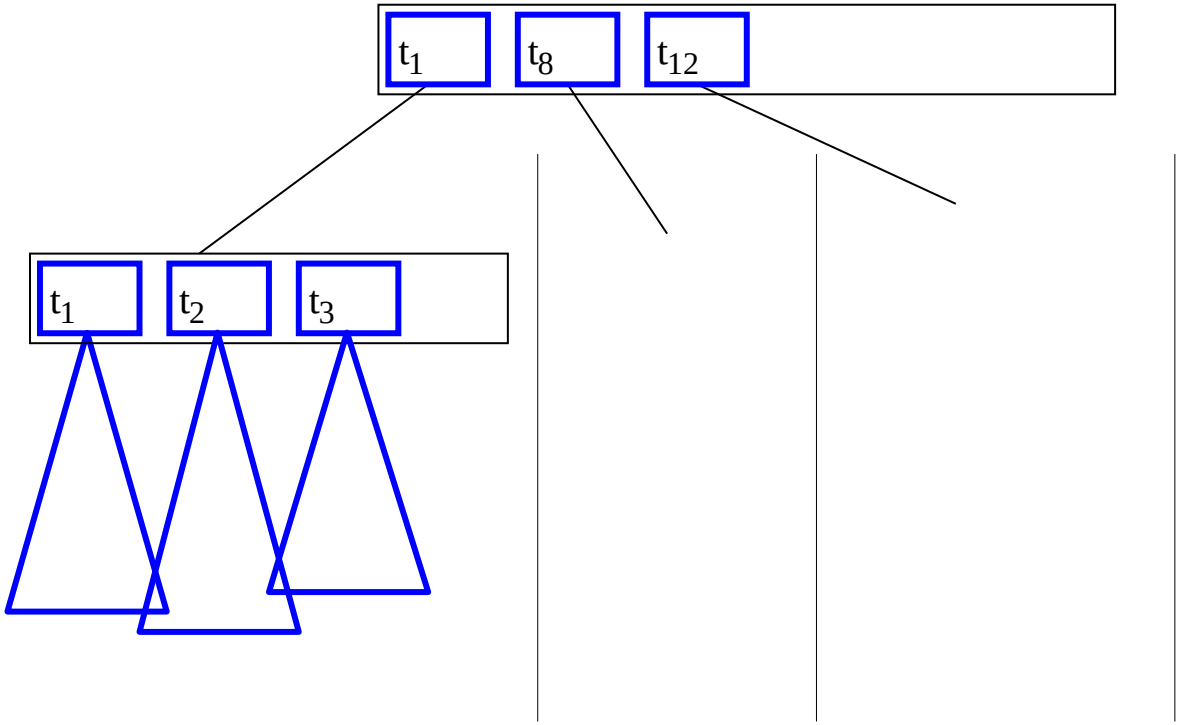
3-d Convex Hull

- ☆ Eliminate redundant half-spaces.
- ☆ Poll to find a sample that gives a well-balanced partition.
- ☆ With high probability, there will be a sample such that the subproblem sizes add up to $O(N)$ and the largest is at most $\log N$ times the smallest.
- ☆ Polling uses random sampling to find the good sample in $O(n)$ I/Os.
- ☆ Recurse in each cone.

Batched Persistent B-trees

- ☆ Problem: given $\sigma_1, \sigma_2, \ldots, \sigma_N$, where $\sigma_i =$ insert(x) or delete(x), construct a data structure that allows a "B-tree search" in the past.
- ☆ We will apply distribution sweeping to construct a structure with $\sqrt{m}$-way branching.
- ☆ We achieve $O(n \log_m n)$.
- ☆ Online method takes $O(N \log_m n)$.

Batched Persistent B-trees

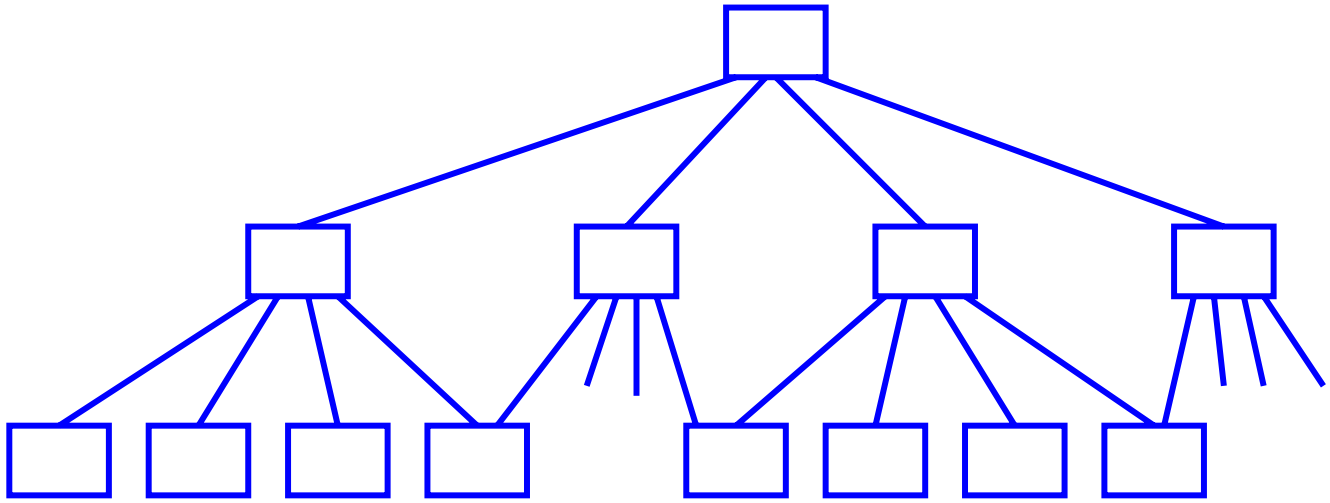


- ☆ Online property doesn't hold for batched persistent B-trees.
- ☆ Online Property: For any time t, a root to leaf search or range search w.r.t. time t traverses only blocks that are $\geq$ half-full.
- ☆ Important for output-sensitivity in time-stamped 1-d range search (3-sided range search).

Batched Persistent B-trees

- ☆ Online property not important for applications like batched planar point location.
- ☆ Applications:
  - K simultaneous point location queries.
  - K ray-shooting queries in CSG model.
  - K range queries.
  - Graph drawing.

Persistent B-trees and Batch Filtering



- ☆ Outdegree $\leq m$
- ☆ Search a layered planar dag in

$$O(n + (q+1)\text{height}) \text{ I/Os},$$

where $Q = qB$ is the number of queries.

Persistent B-trees and Batch Filtering

- ☆ Start by sending all queries to the root node.
- ☆ Proceed level by level, sending all Q queries to level i before sending any to level $i + 1$.
- ☆ To do this I/O-efficiently, maintain a FIFO queue of queries that flow through the edges between current level and next level.
  - If less than B queries traverse an edge, store edges in queue.
  - Otherwise, store a pointer to a linked list of blocks.
- ☆ The queue for the next level is produced from the current one I/O-efficiently.

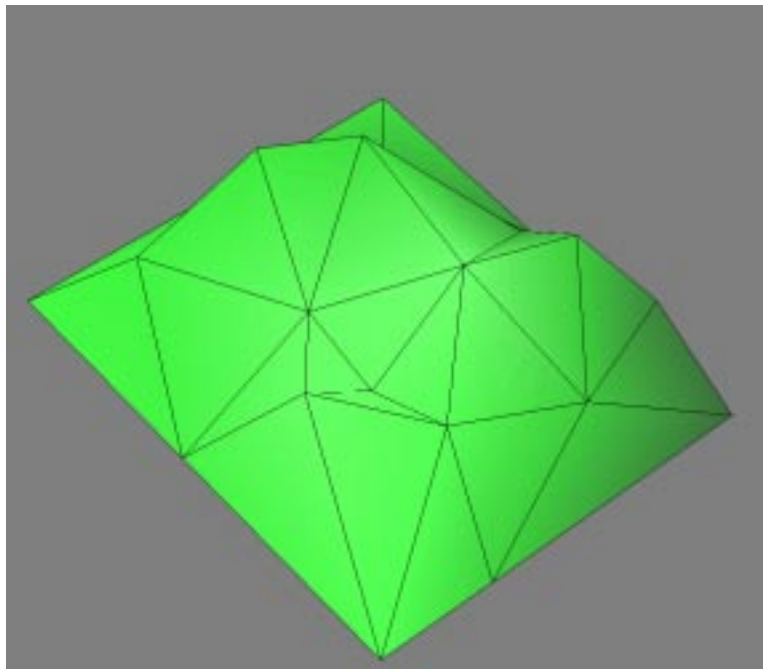
Map Overlay / Spatial Join

Spatial Data:

- Maps
- Terrains
- CAD models
- VLSI models

Traditionally, spatial data is stored in layers.

Overlaying layers (map overlay) is a fundamental operation in geographical information systems (GIS).

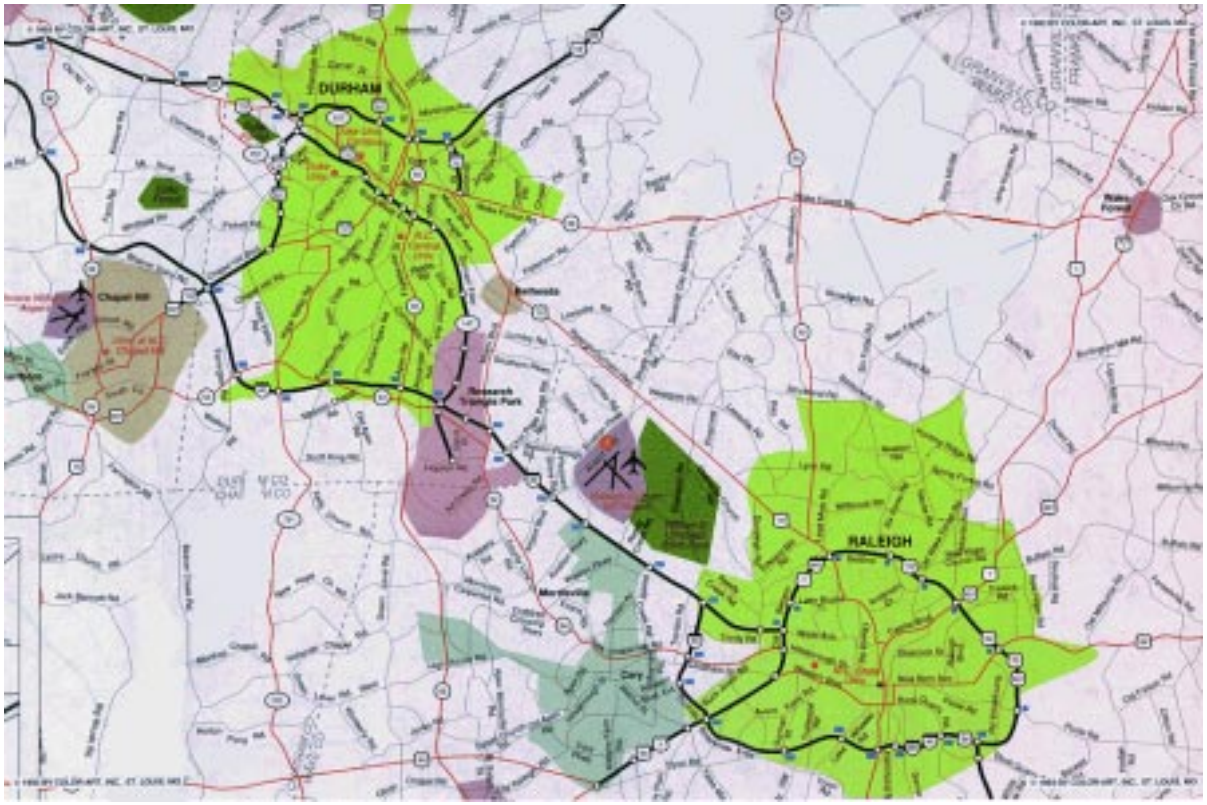


Geographical Information Systems

A typical GIS might store the following layers:

- Roads
- Rivers and lakes
- Railroads

Example: roads in Triangle Area.

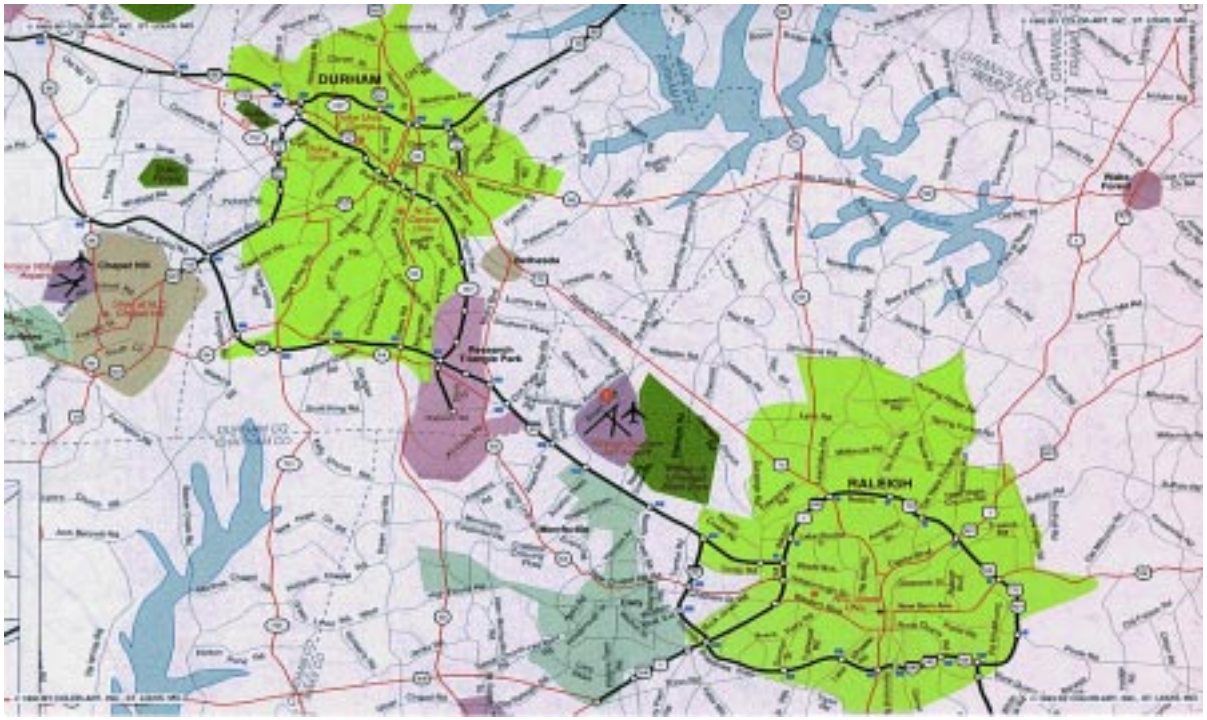


Geographical Information Systems

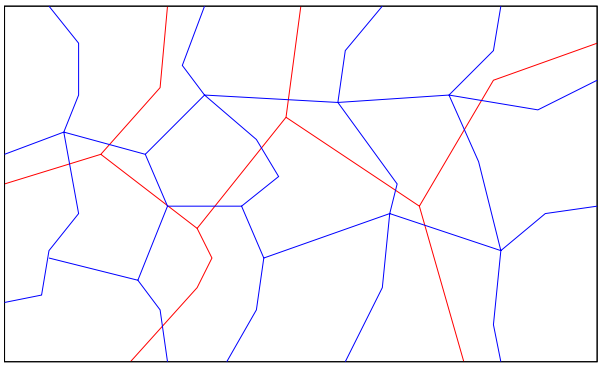
Query: “Find all bridges in Triangle Area”

Requires map overlay (the roads map with the rivers/lakes map),

a type of spatial join.



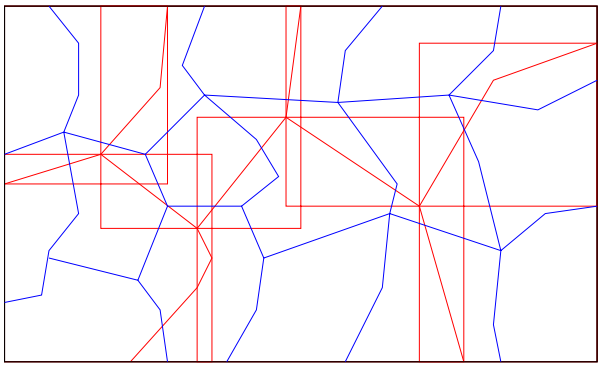
Spatial Join



Land Utilization Pollution level

- ☆ 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: intersecting the two sets of rectangles.
- ☆ Issues:
  - # I/Os,
  - Indexed vs. non-indexed structures for storing the rectangles.
  - Skewed data

Spatial Join



Land Utilization Pollution level

- ☆ 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: intersecting the two sets of rectangles.
- ☆ Issues:
  - # I/Os,
  - Indexed vs. non-indexed structures for storing the rectangles.
  - Skewed data

Case I: No Indexes

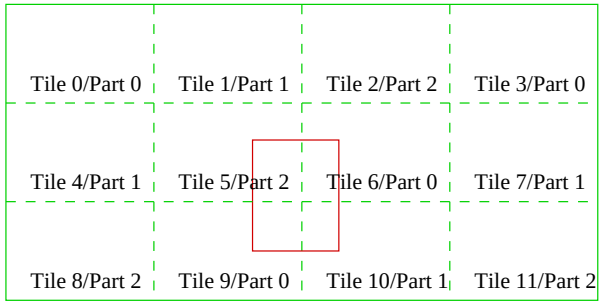
Previous Algorithm: PBSM [PD96]

Partitions data into tiles

Drawbacks:

Reports duplicate intersections

A tile may not fit in memory



New Improved Algorithm:

SSSJ [APRSV98]

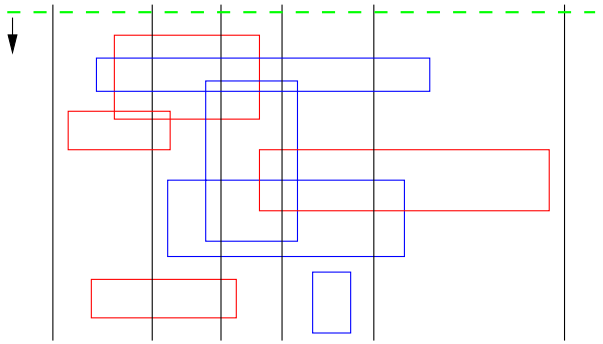
Sort on x coordinate, then sweep.

Advantages:

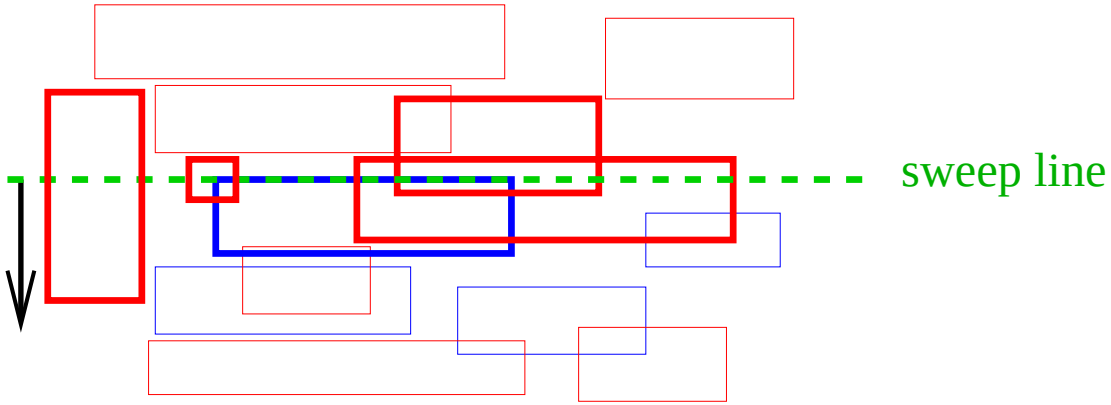
No duplicate intersections

Optimal I/O performance

Robust to skewed data

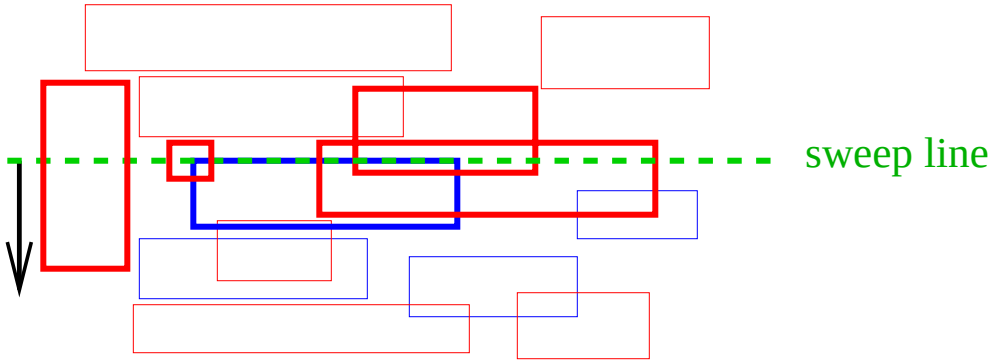


Red-Blue Rectangle Intersection



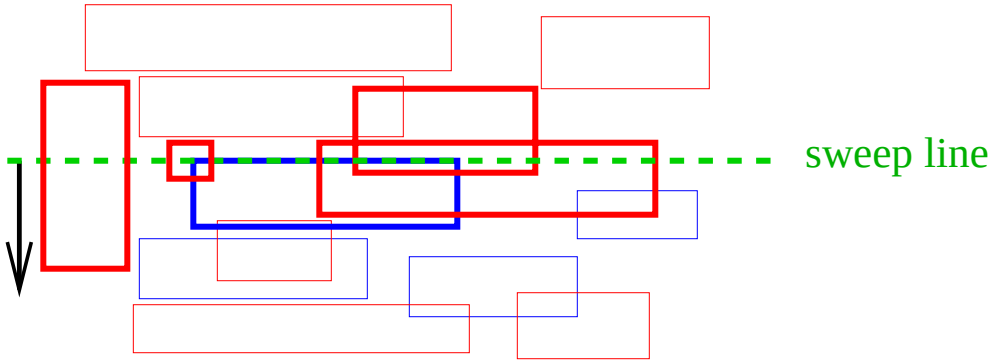
- ☆ Sweep plane while maintaining two active lists of red and blue rectangles intersecting vertical sweep line [BW80]:
  - When top of blue rectangle is reached:
    (i) Insert blue rectangle in blue active list.
    (ii) Find intersections with rectangles in red active list.
  - When bottom of blue rectangle is reached:
    (i) Remove rectangle from blue active list.
- ☆ Red rectangles are handled similarly.

Red-Blue Rectangle Intersection



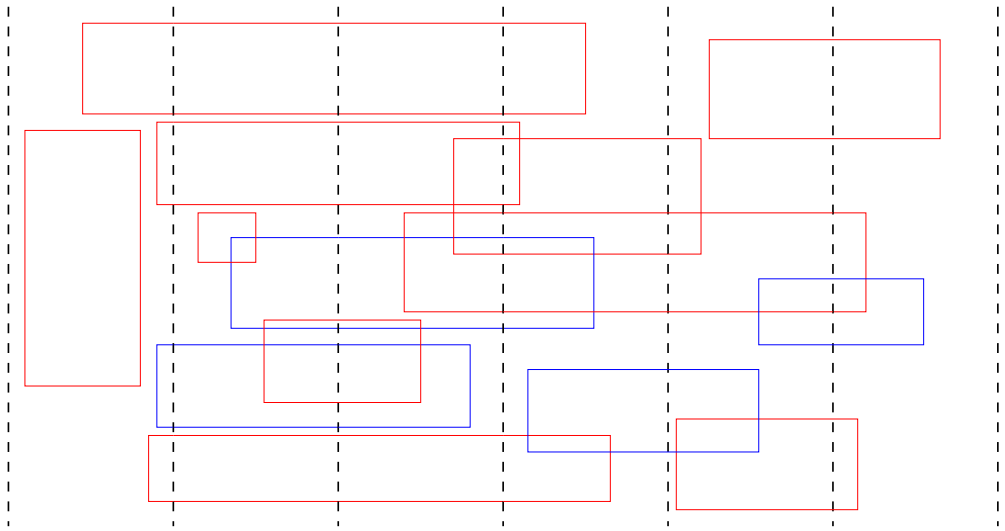
- ☆ Algorithm performs badly ($> N$ I/Os) if size of active lists $> M$.

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]:



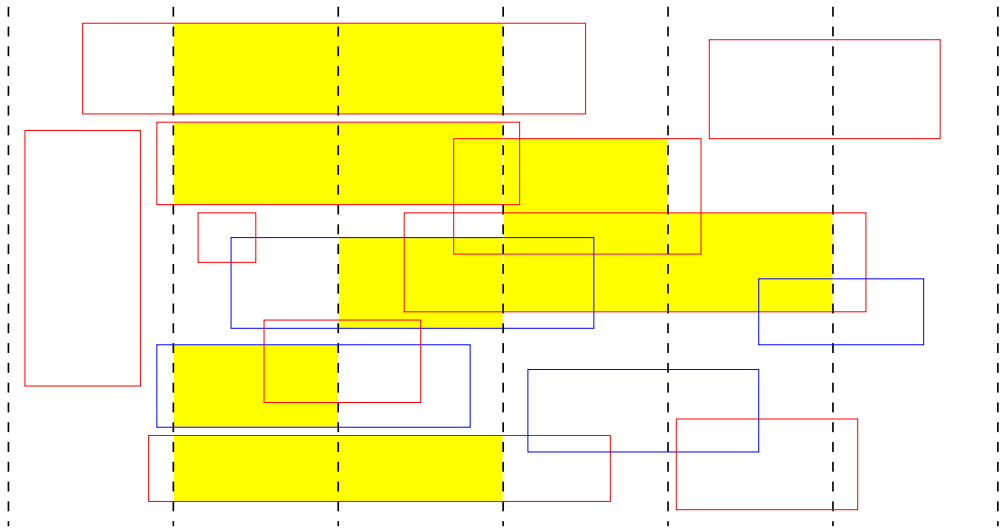
1. Divide plane into $\sqrt{m}$ slabs, each with $O(N/\sqrt{m})$ endpoints.
2. Break rectangles into three pieces:
   left endpiece, centerpiece, and right endpiece.
3. Find Z' intersections involving at least one centerpiece.
4. Recursively solve problem in each slab for endpieces.

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

★ Performing Step 3 in $O\left(n + \frac{Z'}{B}\right)$ I/Os

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

Sketch of External Solution [APRSV98]:



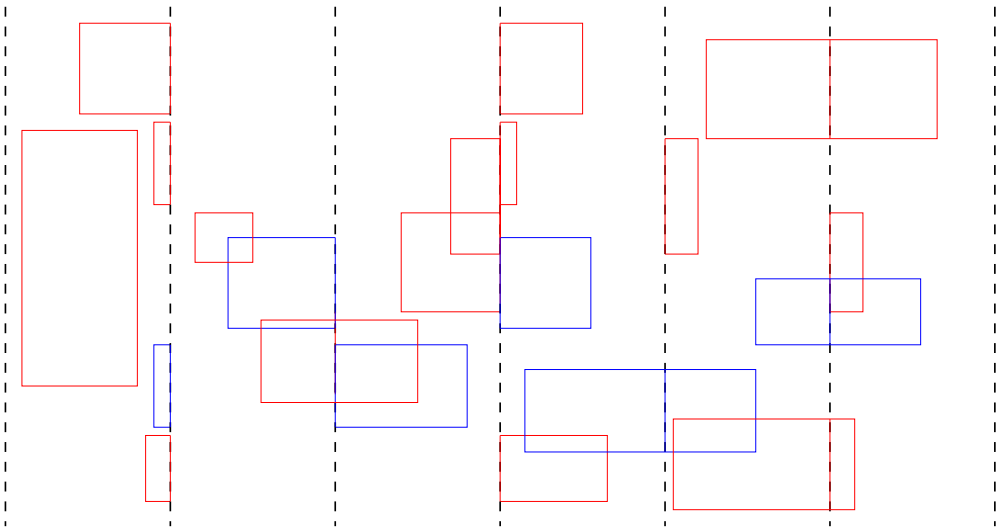
1. Divide plane into $\sqrt{m}$ slabs, each with $O(N/\sqrt{m})$ endpoints.
2. Break rectangles into three pieces:
   left endpiece, centerpiece, and right endpiece.
3. Find Z' intersections involving at least one centerpiece.
4. Recursively solve problem in each slab for endpieces.

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

★ Performing Step 3 in $O\left(n + \frac{Z'}{B}\right)$ I/Os

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

Sketch of External Solution [APRSV98]:



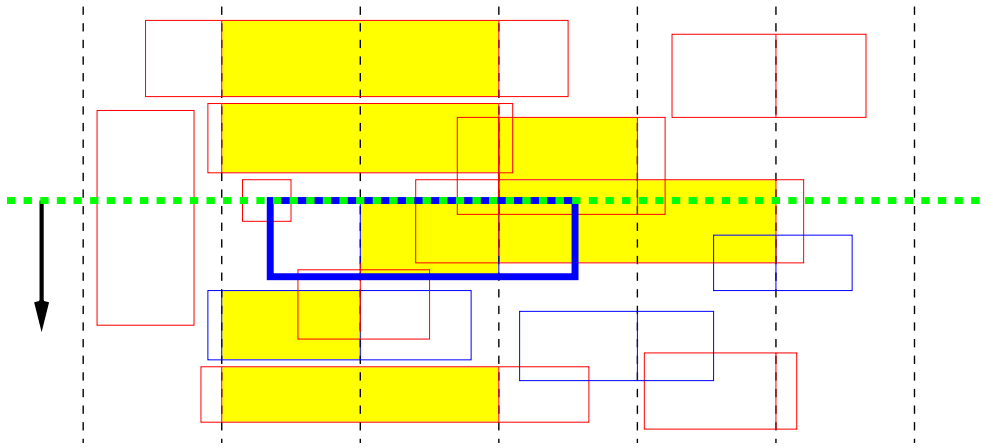
1. Divide plane into $\sqrt{m}$ slabs, each with $O(N/\sqrt{m})$ endpoints.
2. Break rectangles into three pieces:
   left endpiece, centerpiece, and right endpiece.
3. Find Z' intersections involving at least one centerpiece.
4. Recursively solve problem in each slab for endpieces.

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

★ Performing Step 3 in $O\left(n + \frac{Z'}{B}\right)$ I/Os

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

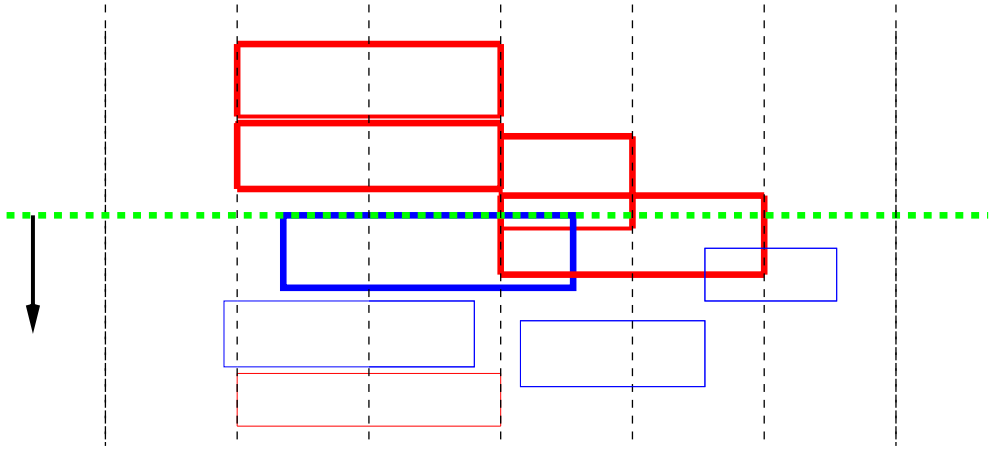
Key Idea



Consider intersections of red centerpieces and tops of blue rects.:

- ☆ Use $\sqrt{m}$ slabs
- ☆ $\Longrightarrow$ $O(m)$ multislabs (continuous ranges of slabs)
- ☆ Store each red centerpiece in a multislab, implemented as a stack.
- ☆ Stack effectively keeps the first B rectangles of each 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 (stack).
  - 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). (Combine block with neighbor if $< B/2$ living rectangles.)
- ☆ Other cases handled similarly—in one sweep!

Analysis of I/O Performance in each Pass

Intersections of red centerpieces and tops of blue rects.

- ☆ Centerpieces of red rectangles are scanned in $O(n)$ I/Os.
- ☆ For each top of a blue rectangle, we report intersections with non-expired red centerpieces in all relevant multislab lists.
- ☆ Since the first block of each multlislab list (stack) is in internal memory, if a multislab list has k centerpieces, # I/Os $= \left\lfloor \frac{k}{B} \right\rfloor \leq \frac{k}{B}$.
- ☆ Each centerpiece is deleted in lazy manner at most once.
- ☆ Sum of $\frac{k}{B}$ over all reportings is thus at most $\frac{Z' + N'}{B}$, where N' is number of red centerpieces in the current pass.
- ☆ Over the $\log_m n$ passes, summing $O\left(n + \frac{Z' + N'}{B}\right)$ gives a total of $O(n \log_m n + z)$ I/Os.

Avoiding redundant reportings of intersections

- ☆ Example: A given blue rectangle could intersect the centerpiece of a red rectangle, and the blue rectangle's endpiece could intersect the red rectangle's endpiece.
- ☆ Two intersections would be reported at different levels of recursion.
- ☆ *How to fix this without sorting all intersections?*
  (Technically, sorting would require $O(z \log_m z)$ I/Os, which is too much theoretically, and inefficient in practice.)

Avoiding redundant reportings of intersections

- ☆ Example: A given blue rectangle could intersect the centerpiece of a red rectangle, and the blue rectangle's endpiece could intersect the red rectangle's endpiece.
- ☆ Two intersections would be reported at different levels of recursion.
- ☆ *How to fix this without sorting all intersections?*
  (Technically, sorting would require $O(z \log_m z)$ I/Os, which is too much theoretically, and inefficient in practice.)
- ☆ Solution: Avoid redundant reportings of intersections by adopting a convention as to when to report an intersection.
- ☆ For example, each intersection could be reported only at the first available opportunity. At each potential reporting time, the two rectangles must be examined to determine if the intersection has already been reported.
- ☆ Charge each non-reporting to the actual intersection. Each intersection is non-reported at most $O(1)$ times.

Higher Dimensions

- ☆ Technique can be used recursively in dimension $d > 2$ by decreasing number of slab boundaries to $m^{1/2(d-1)}$ in each of the $d-1$ dimensions orthogonal to sweep.
- ☆ For $d = 3$, consider a checkerboard of slabs, $m^{1/4} \times m^{1/4}$.
- ☆ There are at most $m^{1/2} \times m^{1/2} = m$ multislabs.
- ☆ Rectangles are partitioned in x dimension and then a sweep is done in the z dimension simultaneously for all x-slabs to solve the y, z-dimension subproblems. *COMPLICATED!*
- ☆ 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, http://www.cs.duke.edu/TPIE/


Many problems can be solved using small number of paradigms.

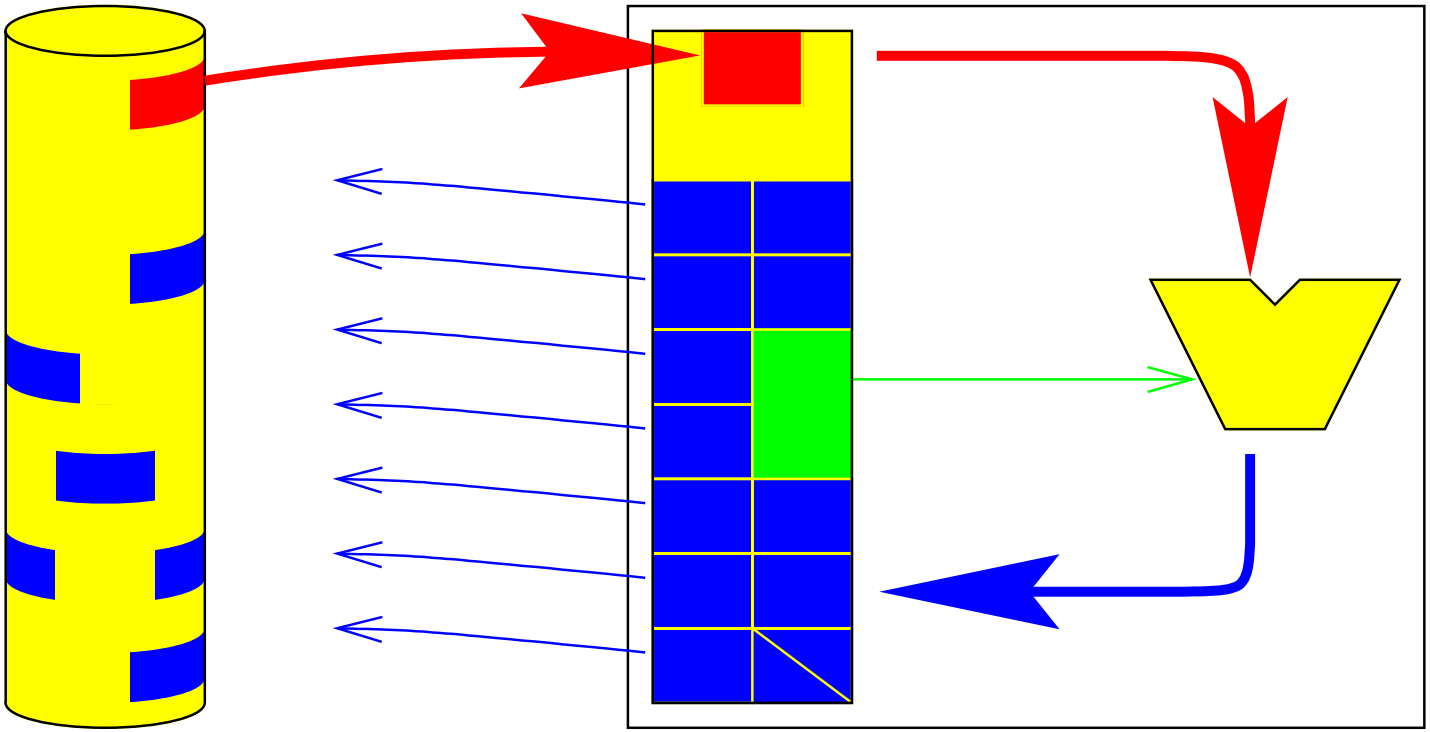

OS often provides inadequate support for I/O and internal memory management.

☆ 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

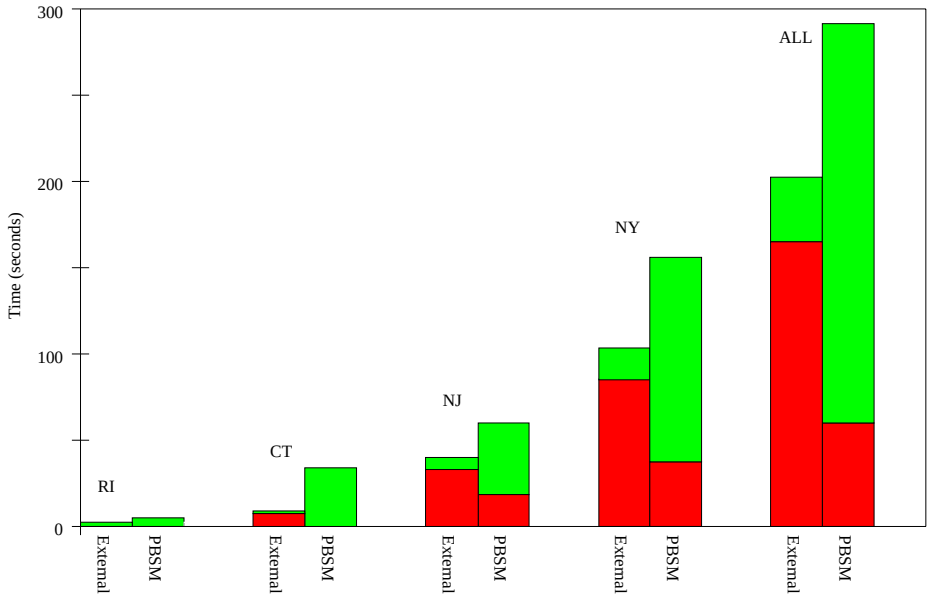


TIGER/Line Data

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

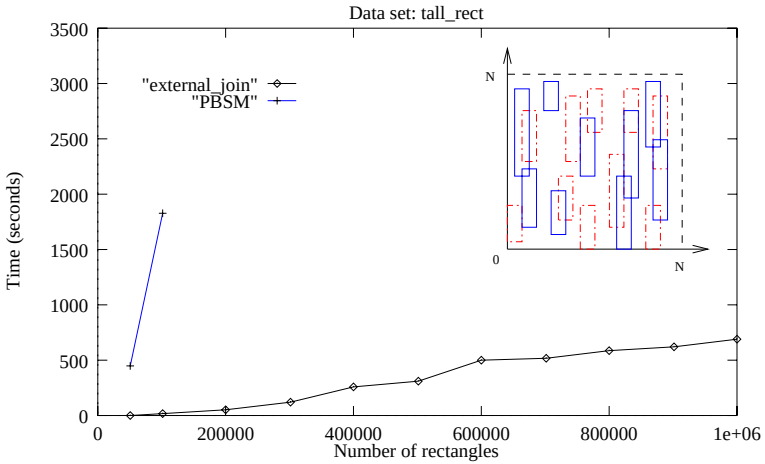
State	Category	Size	Objects
Rhode Island (RI)	Roads	4.3 MB	68,278
	Hydrography	0.4 MB	7,013
Connecticut (CT)	Roads	12.0 MB	188,643
	Hydrography	1.8 MB	28,776
New Jersey (NJ)	Roads	26.5 MB	414,443
	Hydrography	3.2 MB	50,854
New York (NY)	Roads	55.7 MB	870,413
	Hydrography	10.0 MB	156,568
All	Roads	98.5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

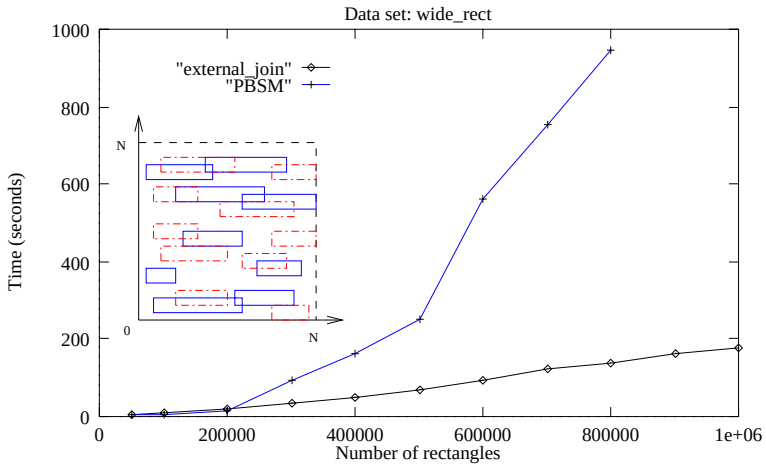
Performance Comparison with PBSM [DP96]



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

Performance Comparison with PBSM [DP96]





Case II: Indexes Exist

Previous Algorithm: ST [BKS93]

Carefully synchronized depth-first traversal.

Our Algorithm: PQ [APRSVV00]

Related Results

- ☆ External segment tree used in conjunction with *batched filtering* [GTVV93] and *external fractional cascading* to solve large number of problems with GIS applications [AVV95]:
  - Red-blue line segment intersection in $O(n \log_m n + t)$ I/Os.
- ☆ Persistent B-trees [GTVV93] to solve batched point location in $O(n \log_m n + t)$ I/Os.
- ☆ Random incremental construction [CFMMR98] to get optimal $O\big((n+q)\log_m n + z\big)$ I/Os for general line segment intersection.

Parallel Simulation Paradigm [CGGTVV95]

- ☆ Let A be an N-processor PRAM algorithm such that
  - A reduces a problem of size N to one of size αN in constant time.
  - Parallel running time of A is $\Theta(\log N)$.
- ☆ For each PRAM statement, sort the N operands so that they are contiguous.
- ☆ Simulate N operations via a linear pass through the data.
- ☆ I/O Complexity for $D = 1$:

$$\begin{aligned} T(N) &= O(sort(N)) + T(\alpha N) \\ &= O(sort(N)). \end{aligned}$$

- ☆ Gives optimal EM algorithms for list ranking, Euler tours, expression tree evaluation, connected components of sparse graph.
- ☆ Sometimes the sorting can be done in $O(N)$ I/Os because of constraints and assumptions [DDH97, SK97].
- ☆ Some problems like topological sorting, BFS, DFS are hard.

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
- ☆ Lots of open problems in the design and analysis of external memory algorithms and data structures. *Stay tuned!*
  - TPIE, see http://www.cs.duke.edu/TPIE/
  - Handling many disks, large merge orders, many partition elements, large fanouts. (Don't use square root trick.)
  - GIS applications (e.g. practical red-blue line segment intersection, nearest neighbor, spatial join, terrain processing).
  - Image processing (indexing images, analyzing images).
  - Fundamental graph problems (e.g. topological sorting, BFS, DFS, connectivity).

Conclusions and Open Problems

- Online dynamic data structures
  (e.g. dynamic point location, range search in higher dimensions, clustering, similarity search).
- String processing, molecular databases.
- Typical-case behavior of popular data structures (e.g., R-trees).
- . . .