External Memory Algorithms: Dealing with MASSIVE Data

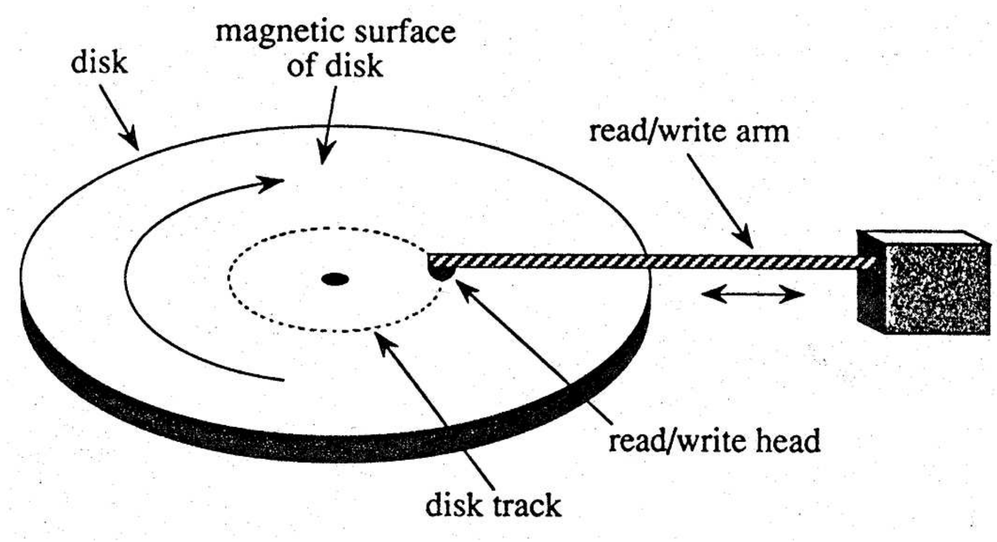
Jeff Vitter

Duke University

Center for Geometric Computing

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

Magnetic Disk Drives as Secondary Memory



☆ I/O Crisis!

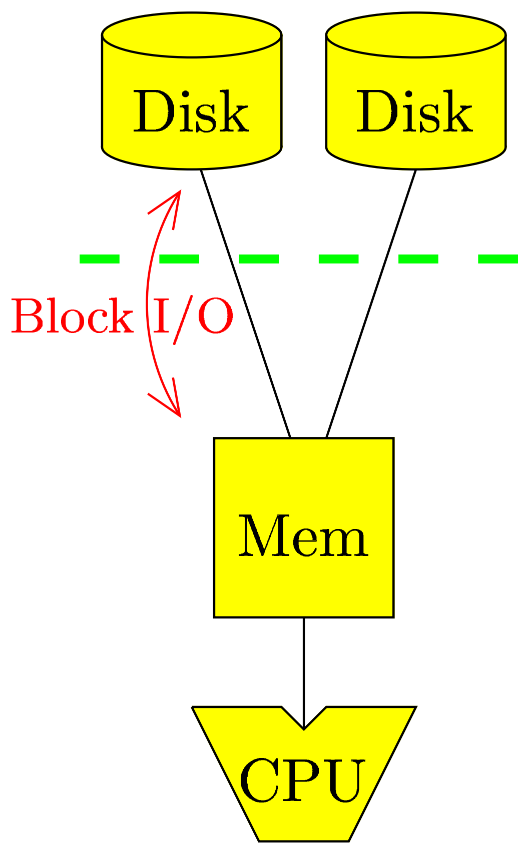
☆ Time for rotation $\approx$ Time for seek.

☆ Amortize search time by large block transfer so that
Time for rotation $\approx$ Time for seek $\approx$ Time to transfer data.

☆ Parallel disks.

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.

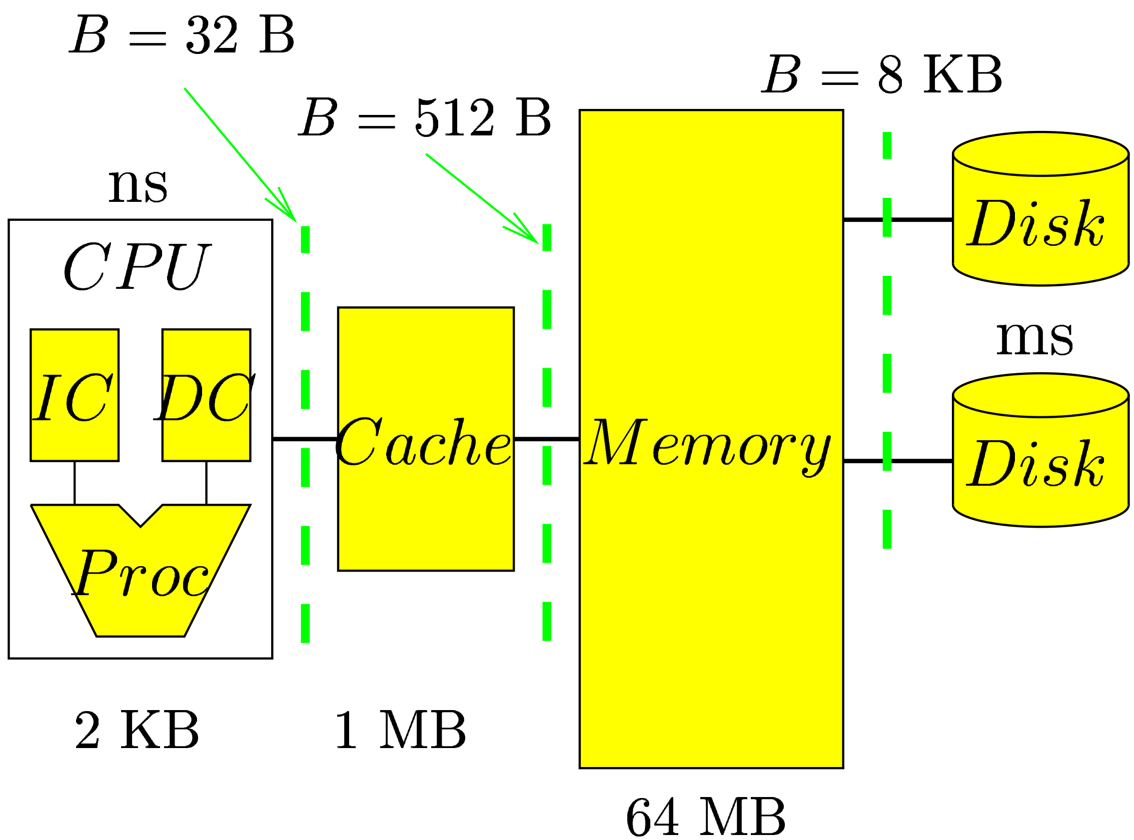
Q = number of queries.

Z = output size.

Notational convenience (in units of blocks):

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

A "Real" Machine



Fundamental Bounds

☆ Batched problems [AV 88], [VS 90,94]:

- Scanning (touch problem): $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$
- Sorting:

$$\Theta\left(\frac{N}{DB}\frac{\log\frac{N}{B}}{\log\frac{M}{B}}\right) = \Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right) = \Theta\left(\frac{n}{D}\log_m n\right)$$

- Permuting: $\Theta\left(\min\left\{\frac{N}{D}, \frac{n}{D}\log_m n\right\}\right)$

☆ Online problems:

- Searching: $\Theta(\log_B n)$

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

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

Disk Striping: $D = 5$ disks, block size $B = 2$

☆ Layout for data:

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$
stripe 0	0 1	2 3	4 5	6 7	8 9
stripe 1	10 11	12 13	14 15	16 17	18 19
stripe 2	20 21	22 23	24 25	26 27	28 29
stripe 3	30 31	32 33	34 35	36 37	38 39

☆ Disk striping involves using the D disks in lock step

$$\Longrightarrow \quad B \leftarrow DB, \; D \leftarrow 1.$$

☆ I/O bound $\Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right)$ increases to $\Theta\left(\frac{N}{DB}\log_{M/DB}\frac{N}{DB}\right)$

☆ Ratio of I/O bounds $\approx \frac{\log m}{\log \frac{m}{D}}$ when $D \approx m$.

☆ To get an optimal sorting algorithm, *use disks independently!*

Outline

1. Sorting with multiple disks.
   - ☆ Merge sort.
   - ☆ Distribution sort.
2. Techniques for solving batched dynamic problems.
   - ☆ Parallel simulation.
   - ☆ Distribution sweeping (via TPIE programming environment).
   - ☆ Red-blue orthogonal rectangle intersection.
   - ☆ Empirical results.
3. Online data structures.
   - ☆ B-trees, buffer trees, R-trees, etc.
   - ☆ Experiments on bulk-loading R-trees.
   - ☆ Range searching.
4. String processing, SB-trees.
5. Dynamic memory allocation and lower bounds.

Merge Sort with D disks

☆ Form initial sorted runs of length m blocks (one memoryload).

☆ Merge together $R = \Theta(m)$ runs at a time

$$\Longrightarrow \quad \#\ \text{passes} = \log_R \frac{n}{m} = \log_m n - 1$$

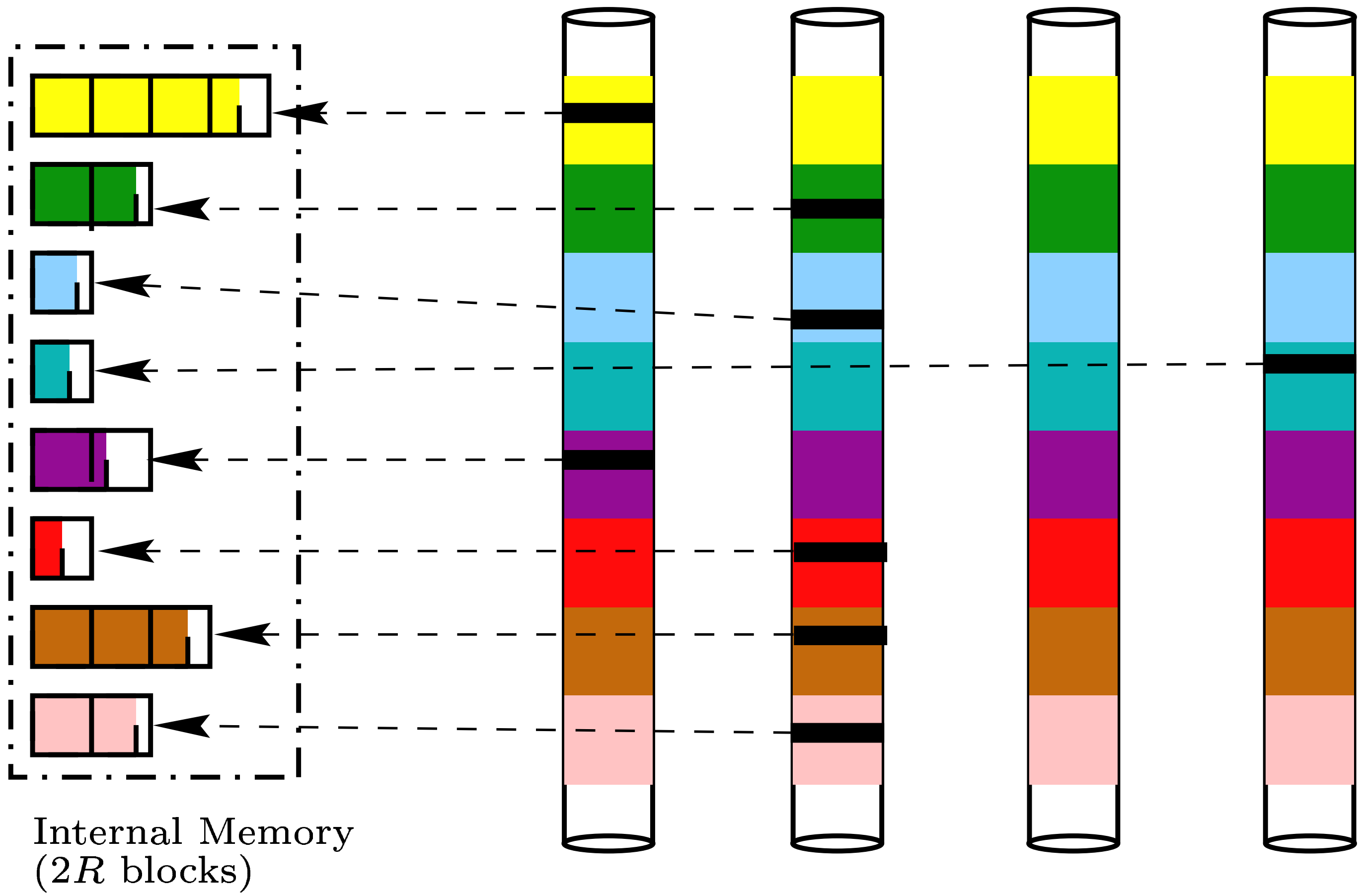
☆ If each pass uses $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ I/Os

$$\Longrightarrow \quad \#\ \text{I/Os} = O\left(\frac{n}{D} \log_m n\right).$$

☆ Three methods:

- Greed Sort [Nodine & Vitter 91]
- Share Sort [Aggarwal & Plaxton 94]
- Simple Randomized Merging [BGV96]

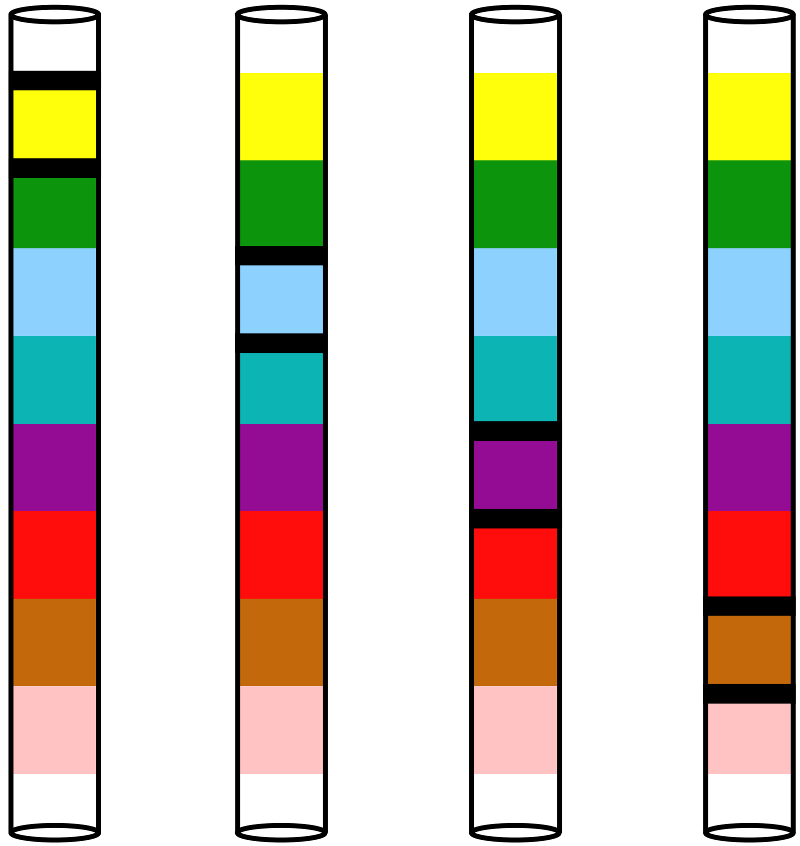
Why Mergesort with D disks is hard



☆ Merge $R \approx \frac{M}{2B} = \frac{m}{2}$ runs (formed in previous pass).

☆ In as few I/Os as possible, read the "next" R blocks. R/D I/Os?

$R = 8$ runs on $D = 4$ disks.



- ☆ Each run is striped, but the starting disk of the runs are staggered.
- ☆ Good balance initially:
  only R/D parallel reads needed to load leading blocks into memory.
- ☆ Balance can quickly deteriorate.

Gilbreath Principle

☆ Can achieve perfect balance for merging two runs, $R = 2$:

Run 1:	A	B	C	D	
	E	F	G	H	
	I	J	K	L	
	…				
Run 2:	D	C	B	A	(striped in reverse order)
	H	G	F	E	
	L	K	J	I	
			…		

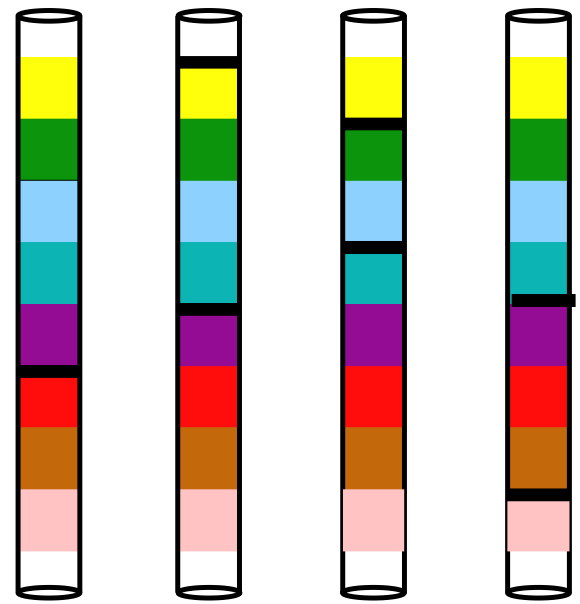
☆ Reduces necessary buffer space by half.

☆ Cannot be generalized to $R > 2$.

Greed Sort [NV91]

Overall structure of each merge pass:

1. Do approximate merge independently on each disk.
2. Interleave the "sorted" runs.
3. Use Columnsort to convert the approximately sorted output run into a totally ordered output run.

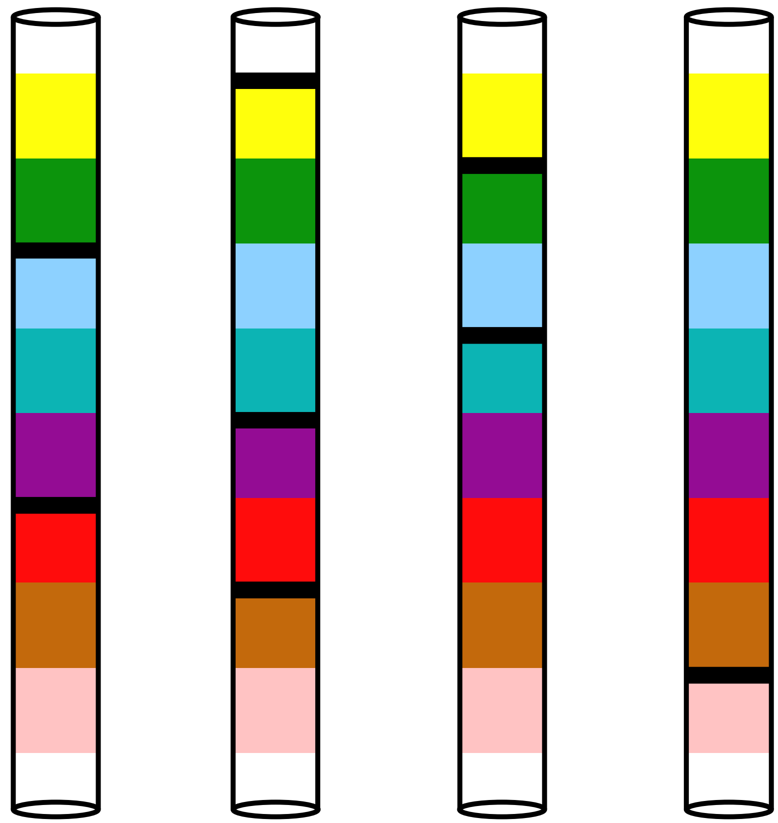


Merge procedure for each disk:

☆ Read the two blocks with smallest and smallest maximum items

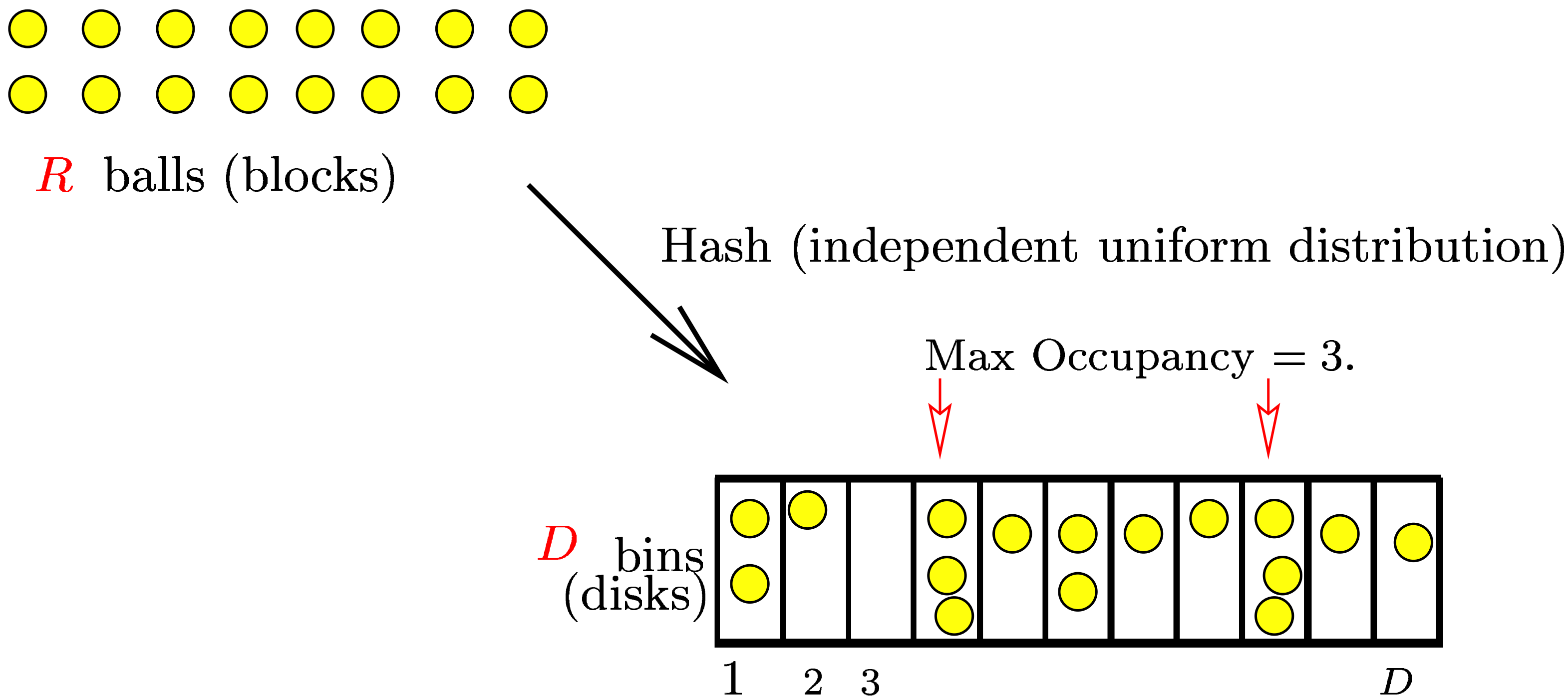
☆ Output the smallest B items of the $2B$ items.

Simple Randomized Mergesort [BGV96, Knuth98]



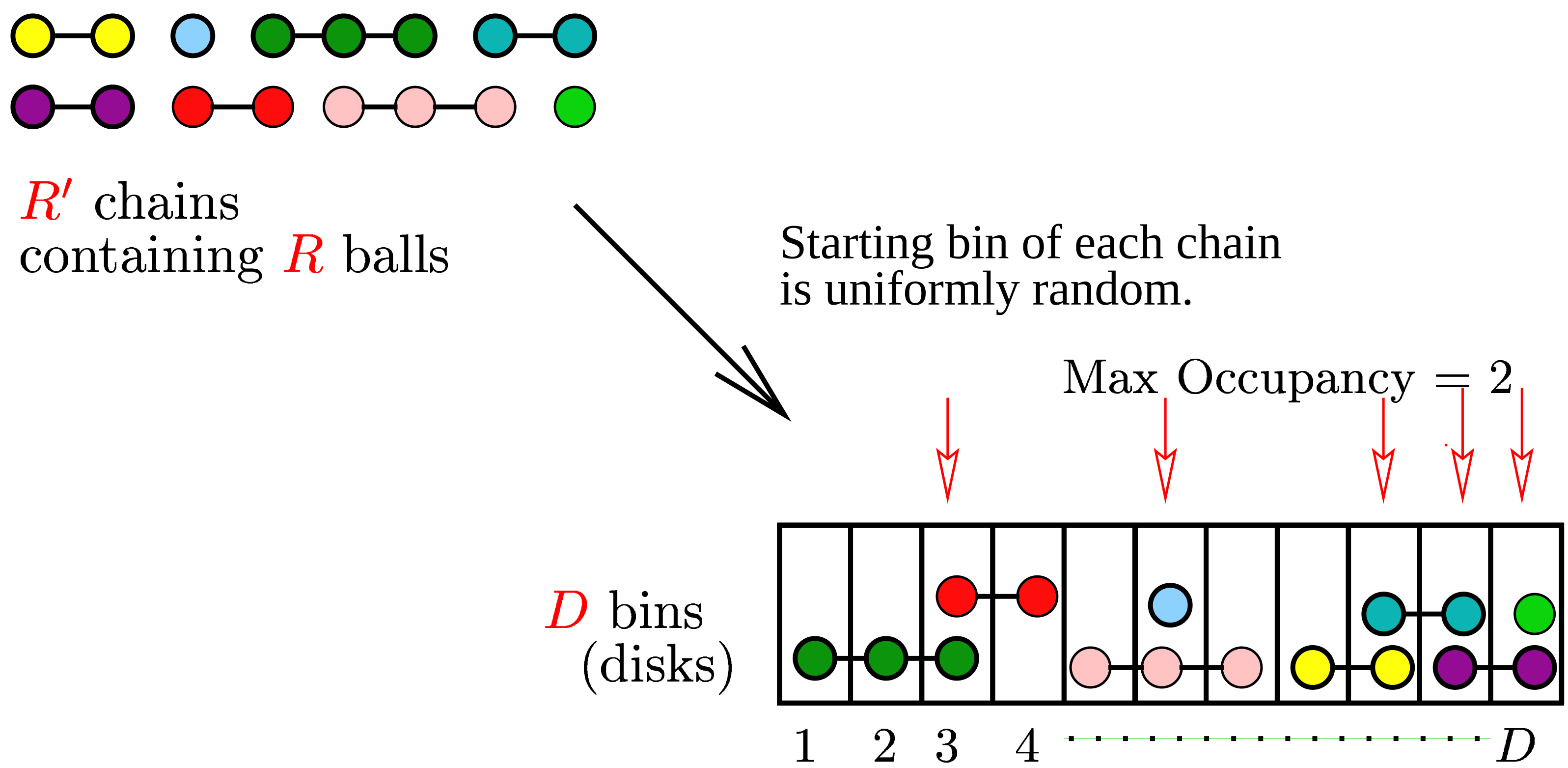
- ☆ Each run is striped starting at a randomly chosen disk.
- ☆ *At any time*, the disk containing the leading block of any run is uniformly random.
- ☆ *Forecast and Flush* buffer management policy.

Classical Maximum Bucket Occupancy



$$E[\text{Classical Max Occupancy}] \sim \begin{cases} \dfrac{\ln D}{\ln\ln D} \cdot \dfrac{R}{D} & \text{if } \dfrac{R}{D} = 1 \\[2ex] c \cdot \dfrac{R}{D} & \text{if } \dfrac{R}{D} = \Theta(\log D) \\[2ex] \dfrac{R}{D} & \text{if } \dfrac{R}{D} \gg \log D \end{cases}$$

Occupancies with Dependency [BGV97, Knuth98]



Conjecture:

$$
\begin{aligned}
&E[\text{Max Occ}(n_1, n_2, \ldots, n_{R'})] \\
&\qquad \leq \quad E[\text{Max Occ}(n_1, n_2, \ldots, n_{R'-1}, n_{R'} - 1, 1)] \\
&\qquad \leq \quad E[\text{Classical Max Occupancy}].
\end{aligned}
$$

I/O Performance of SRM, $R = \frac{m}{2}$

$$E[\#\text{ reads}] \sim \begin{cases} \dfrac{\ln D}{k \ln\ln D} \cdot \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} \approx k. \\ c \cdot \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} = \Theta(\log D). \\ \dfrac{n}{D} \log_m n & \text{if } \dfrac{m}{2D} \gg \log D. \end{cases}$$

Simulation of I/O performance ratio $\dfrac{\mathrm{IO}_{\mathrm{SRM}}}{\mathrm{IO}_{\mathrm{DSM}}}$ for $m \approx (2k+4)D$:

SRM is better than striping!

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	0.56	0.47	0.37
$k = 10$	0.61	0.52	0.40
$k = 50$	0.71	0.63	0.51

Distribution Sort with D Disks

☆ Distribution (bucket) sort

- Select $S = \Theta(m)$ or $\Theta(\sqrt{m})$ partitioning elements that divide the file evenly into buckets.
- Sort the buckets recursively.
- Append together the sorted buckets.

☆ The number of levels of recursion is $\log_S n = \log_m n$.

☆ If each level of recursion uses $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ I/Os

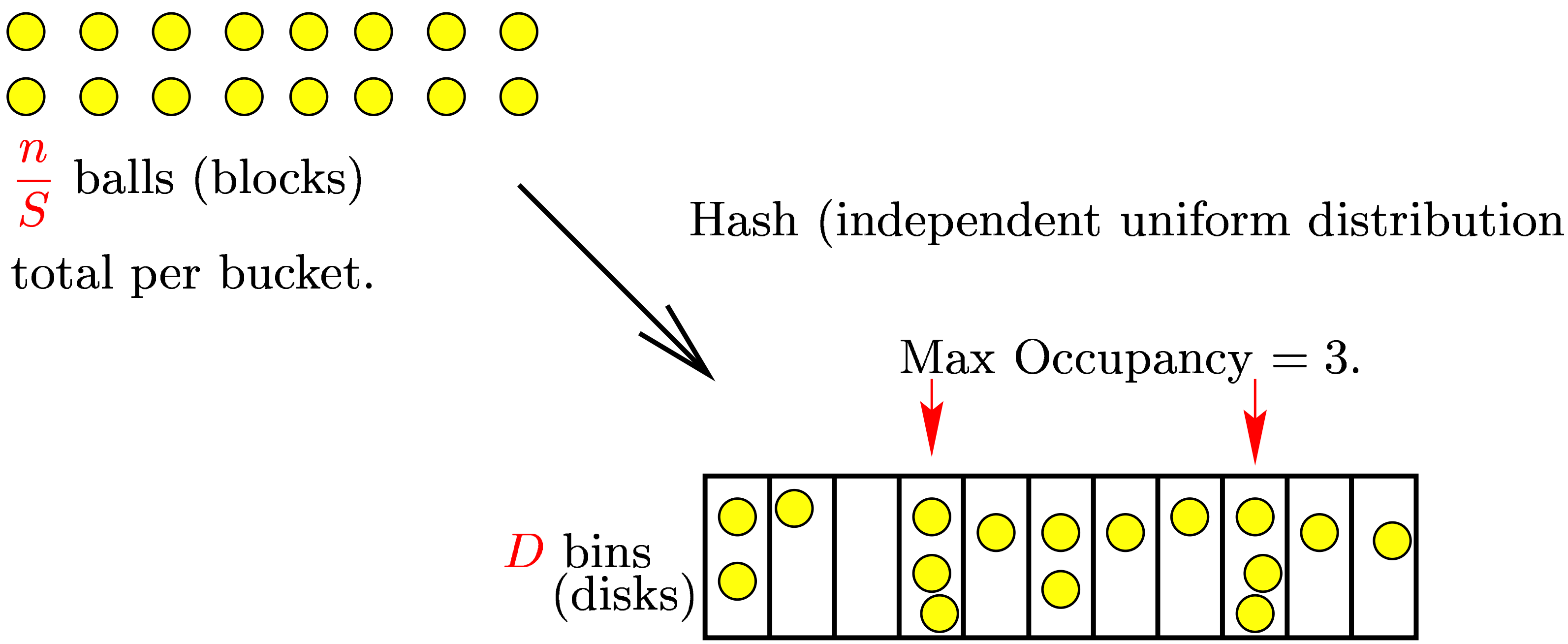
$$\Longrightarrow \quad \#\text{ I/Os} = O\left(\frac{n}{D}\log_m n\right).$$

☆ Difficulty is to store each bucket evenly across the disks, given that the buckets are stored in an online manner.

Bucket Sort [VS94]

☆ If N is large or $\dfrac{M}{DB} > \log D$, then random assignment to disks works well.

☆ S simultaneous load balancing problems (one per bucket).



Bucket Sort [VS94]

- ☆ If N (and S) are small and $DB \approx M$ (so that random assignment is not "balanced"), a "typical" memoryload contains more than $S \log S$ blocks (and is therefore well-balanced among the S buckets.)
- ☆ Get a "typical" memoryload by permuting each memoryload and then extracting a part from each of several memoryloads.
- ☆ Output each memoryload by a round-robin placement (perfect shuffle) of the S buckets onto the D disks.

BalanceSort [NV93]

☆ Online tracking of bucket distribution on disks.

☆ Let $num_b = \#$ items in bucket b processed so far.

☆ Let $num_b(d) = \#$ items in bucket b written to disk d, i.e., $num_b = \sum_{1 \le d \le D} num_b(d)$.

☆ Maintain invariant that the $\left\lfloor \frac{D}{2} \right\rfloor$ largest values of $num_b(1), num_b(2), \ldots, num_b(D)$ differ by at most 1.

$$\Longrightarrow \quad num_b(d) \le 2\frac{num_b}{D}, \text{ for each bucket } b.$$

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.

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.

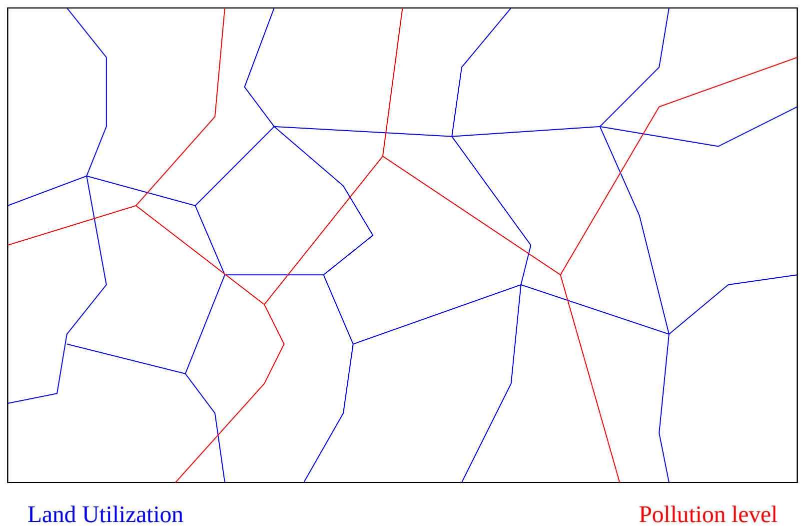
Use of virtual memory $\Longrightarrow$ $\Omega\big((N+Q)\log_B N + Z\big)$ I/Os.

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

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

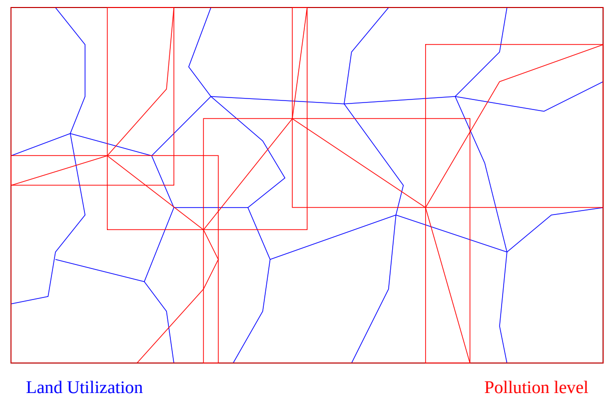
Map Overlay/Spatial Join

☆ Find all farmland with level of pollution over certain threshold.



☆ Red-blue line segment intersection important subproblem.

Map Overlay/Spatial Join

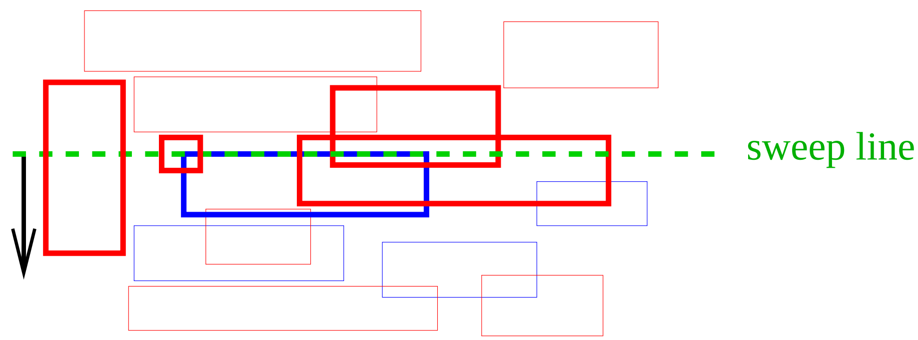


☆ In database literature often solved in two steps:

- 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.

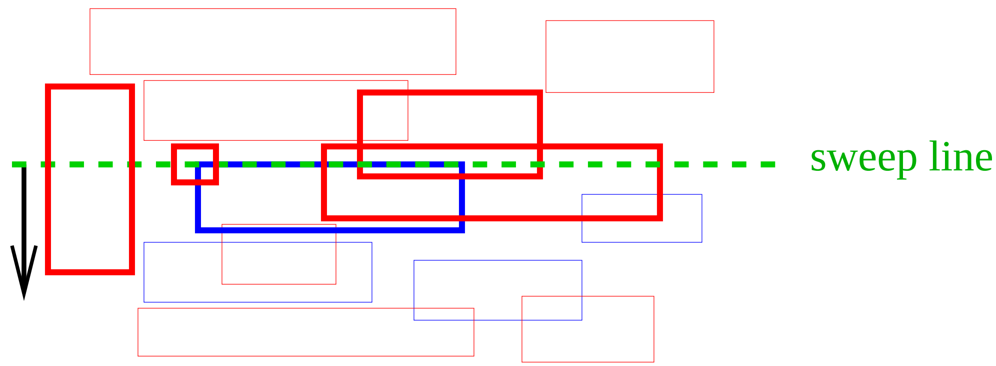
  (ii) Find intersections with rectangles in red active list.

- When bottom of blue rectangle is reached:

  (i) Remove rectangle from blue active list.

☆ Red rectangles are handled similarly.

Red-Blue Rectangle Intersection



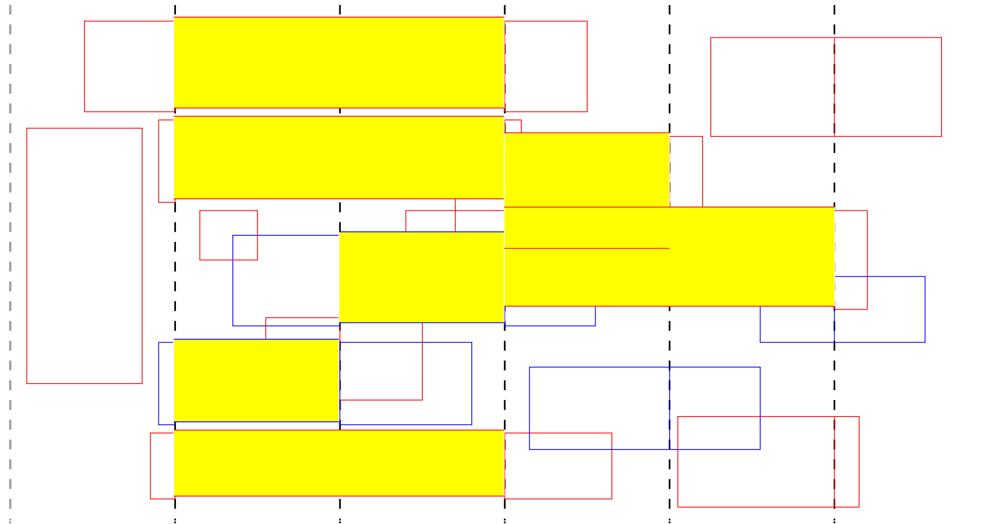
- ☆ Algorithm performs badly ($> N$ I/Os) if size of active lists $> M$.

- ☆ 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]:



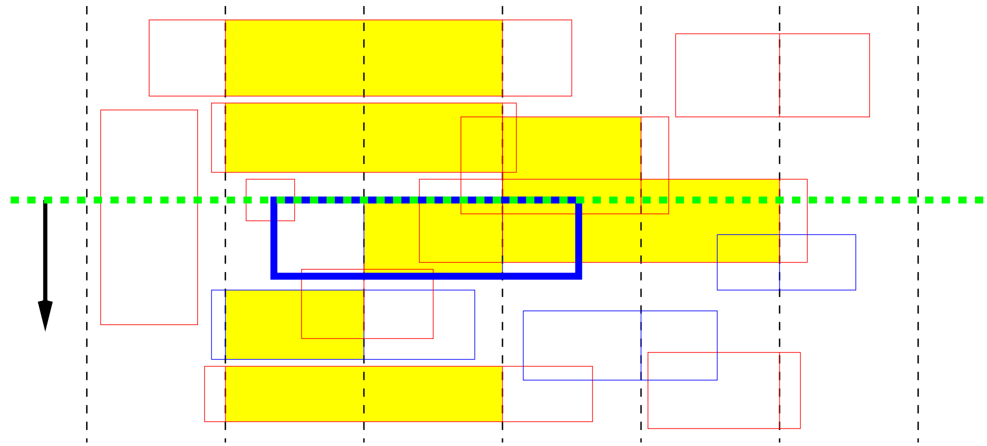
1. Divide plane into $\sqrt{m}$ slabs, each with $O(N/\sqrt{m}\,)$ endpoints.
2. Find Z' intersections involving the part of a rectangle completely spanning slabs.
3. Recursively solve problem in each slab.

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

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

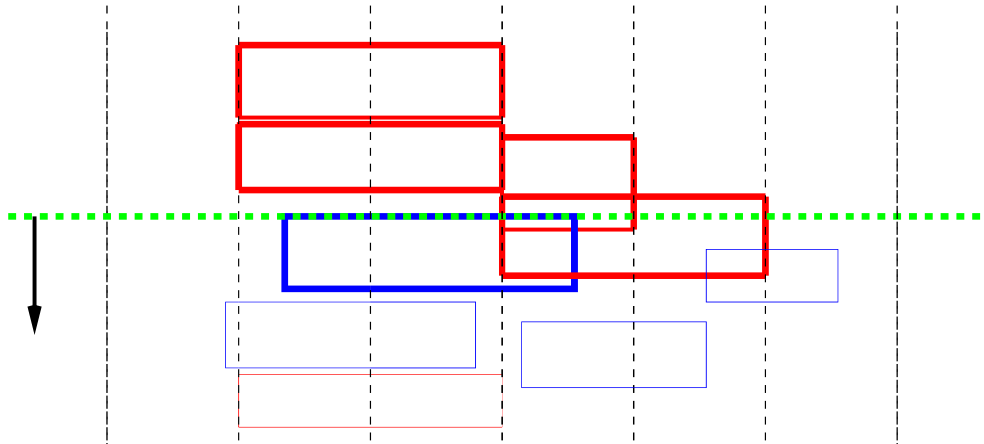
$$\Longrightarrow \quad O(n \log_m n + z) \text{ I/Os total.}$$

Key Idea



- ☆ Use $\sqrt{m}$ slabs
- ☆ $\Longrightarrow$ $O(m)$ multislabs (continuous ranges of slabs)
- ☆ $\Longrightarrow$ 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!

General Technique: Colorable Problems [APRSV98]

☆ Algorithm is special case of general technique:
Batched dynamic version of static "colorable" problems can be solved using extra I/O factor of $O(\log_m n)$.

☆ Proven using external segment tree [Arge95].

☆ Technique can be used recursively (by decreasing number of slabs further) $\implies$ higher-dimensional results.

☆ 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.

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.

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

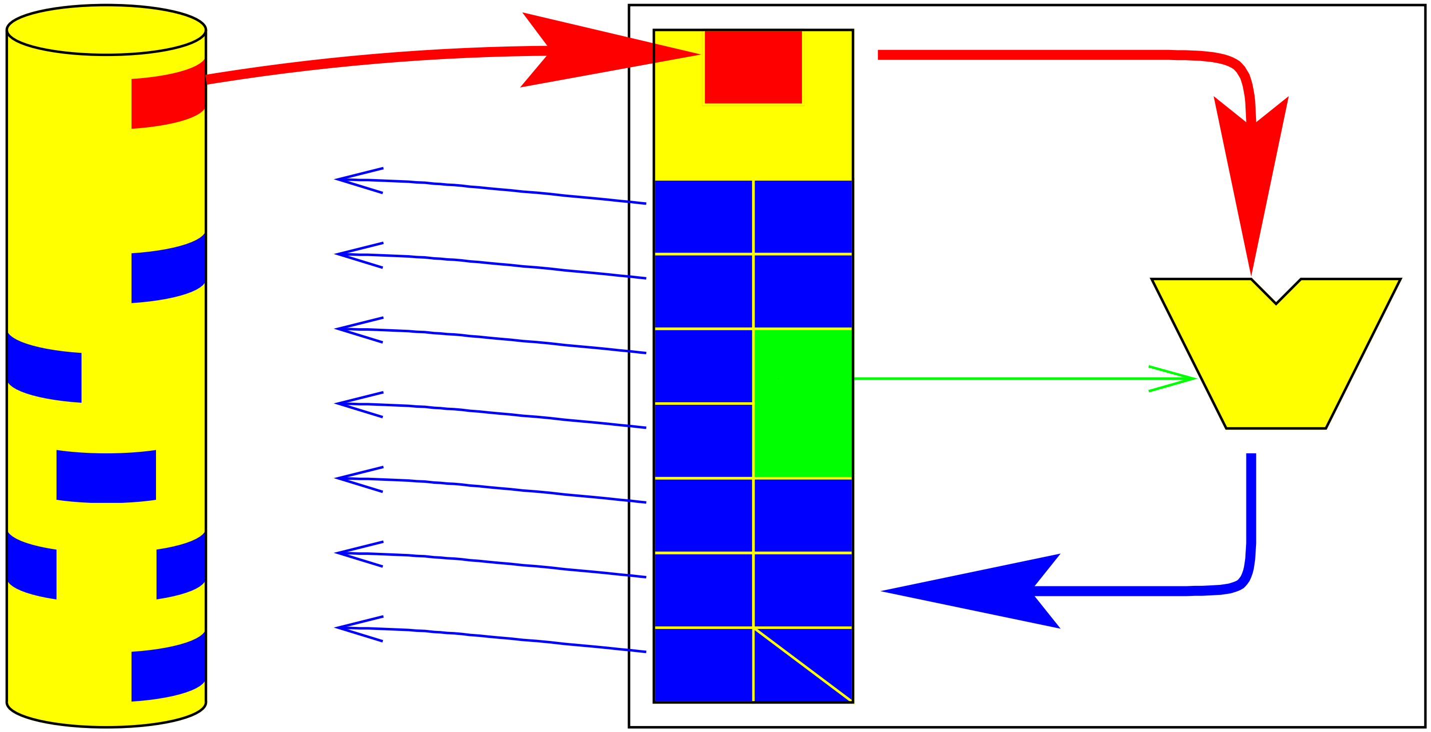
☺ Many problems can be solved using small number of paradigms.

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

⋆ TPIE originally designed by Darren Vengroff [Ven94]:

- 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: Under development (for index structures).

TPIE's Distribution Access Method

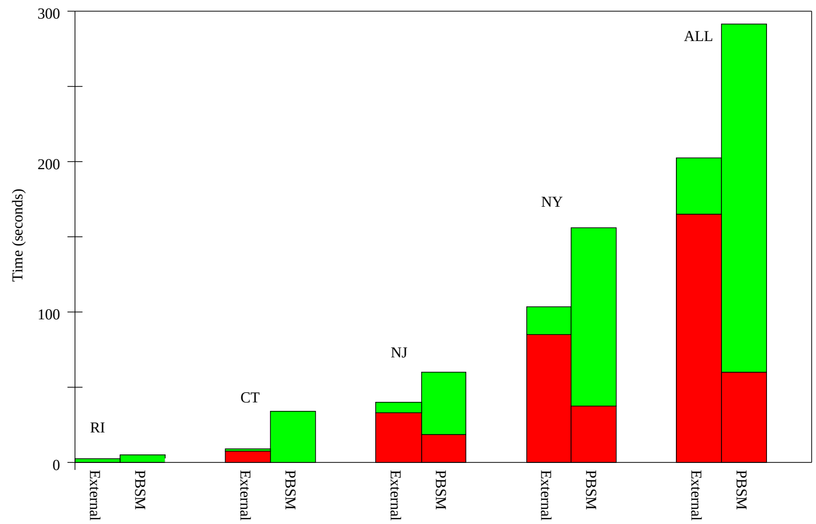


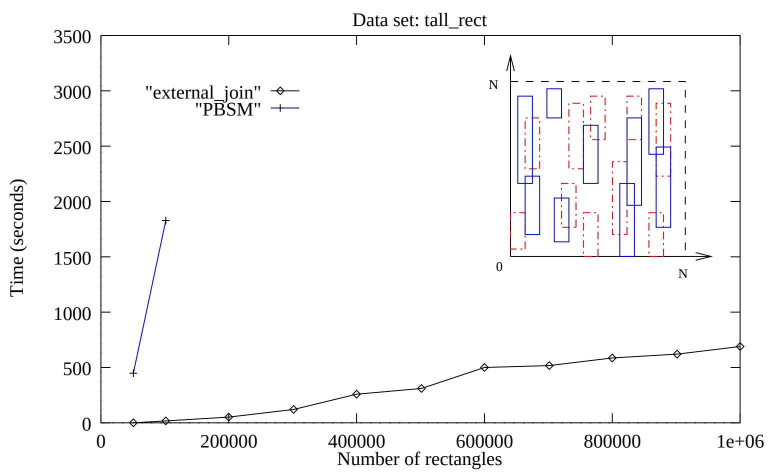
TIGER/Line Data

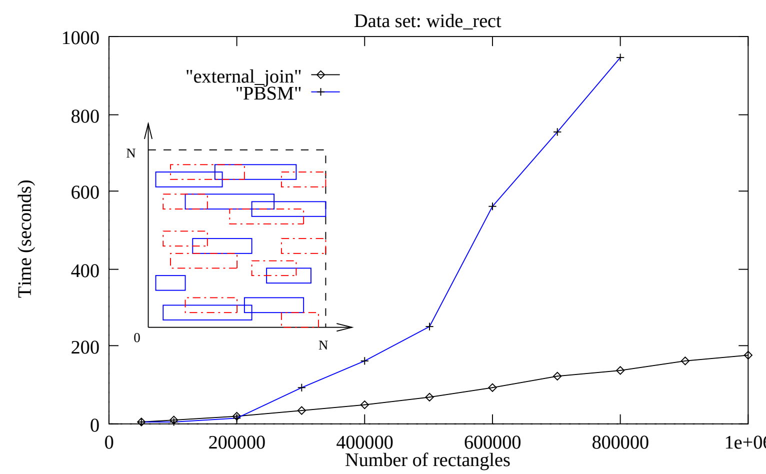
☆ TIGER/Line data from U.S. Census Bureau
(standard benchmark data for spatial databases)

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]



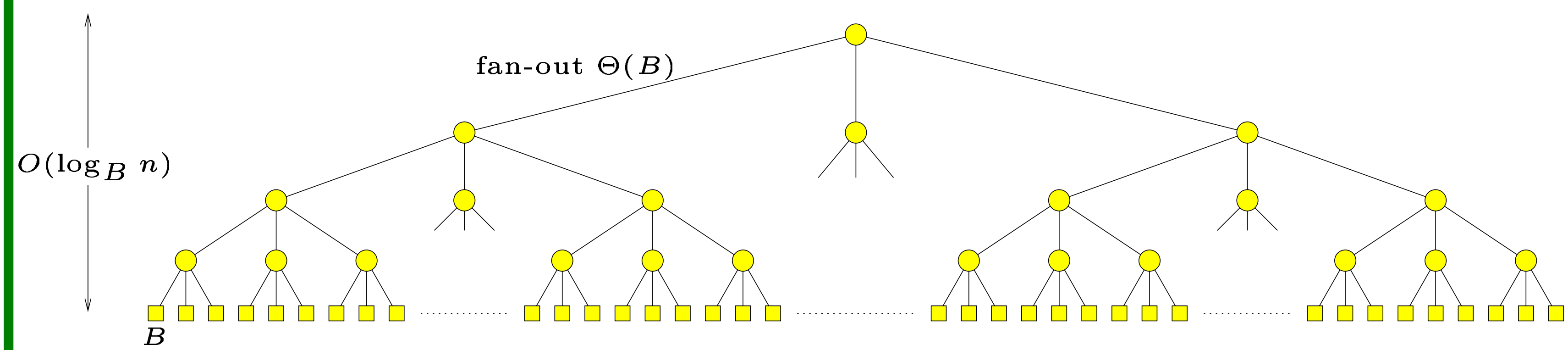




Sun SparcStation 20 (Solaris 2.5) , 32MB memory (TPIE 12MB)

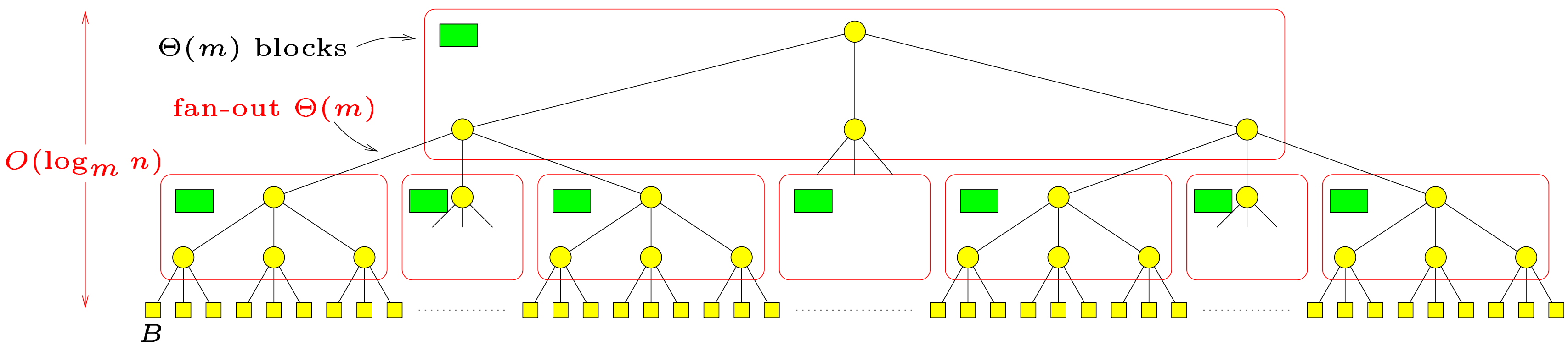
B-tree Construction

[BMc72]



- ☆ Each B-tree node fits in a block:
  - `Insert`, `Delete`, `DeleteMin`, `Query` in $O(\log_B N)$ I/Os.
  - Optimal if each operation is handled individually.
- ☆ Building B-tree:
  - Repeated insertion $\Longrightarrow$ $O(N \log_B n)$-I/O algorithm.
  - Can we take advantage of blocking and obtain $O(n \log_m n)$?

The Buffer Tree [Arge 95]



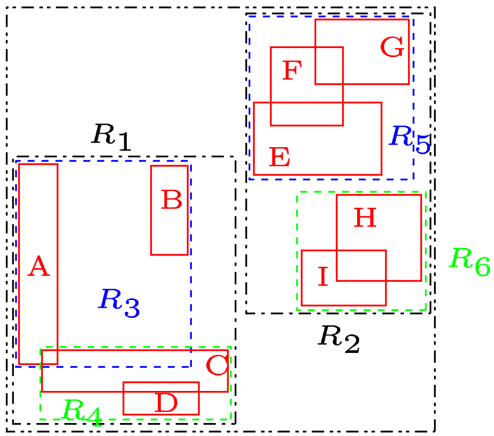
Main idea: Logically group nodes together and add buffers:

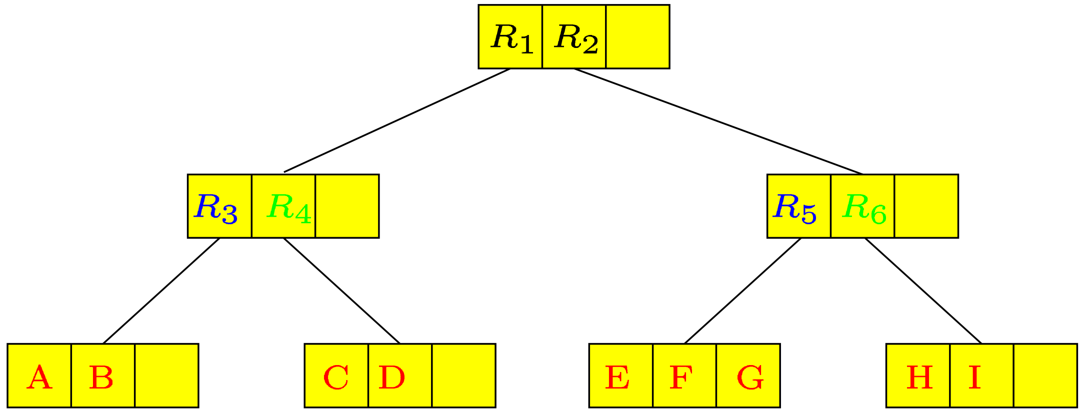
- ☆ Insertions are done "lazily"—items inserted into buffers.
- ☆ When a buffer runs full, its items are pushed one level down.
- ☆ Buffer-emptying in $O(m)$ I/Os
  - $\Longrightarrow$ every *block* touched $O(1)$ times on each level.
  - $\Longrightarrow$ inserting N items (n blocks) in $O(n \log_m n)$ I/Os.

The Buffer Tree Technique

- ⋆ `Delete`, and `Query` operations can be handled similarly.
- ⋆ Efficient `DeleteMin` operation can be designed.
- ⋆ Buffer technique has been used in a number of results:
  - Improved graph algorithms (list ranking) [Arge95]
  - External tournament tree (improved graph algorithms) [KS96]
  - Ordered Binary Decision Diagram manipulation [Arge95b]
  - External heap [FVKT98]
  - String sorting [AFGV97]
  - Bulk operations on R-trees [vdBSW97, AHVV98]
- ⋆ Practically important: [HMSV97, vdBSW97, AHVV98]

R-trees





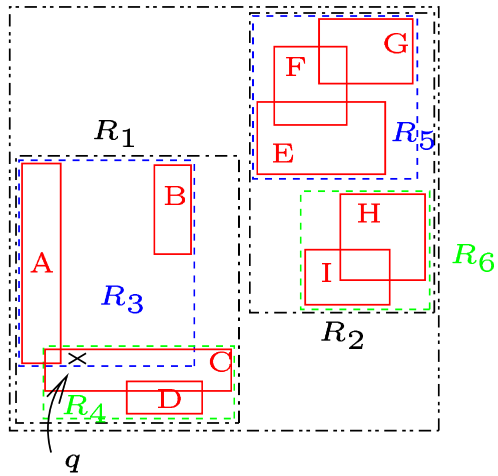
☆ Structure for storing d-dimensional rectangles.

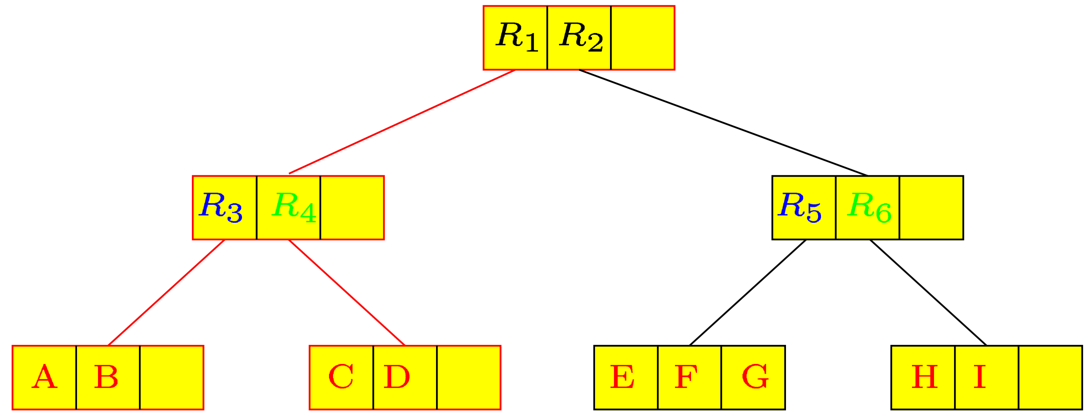
☆ Structured like B-tree:

- 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:

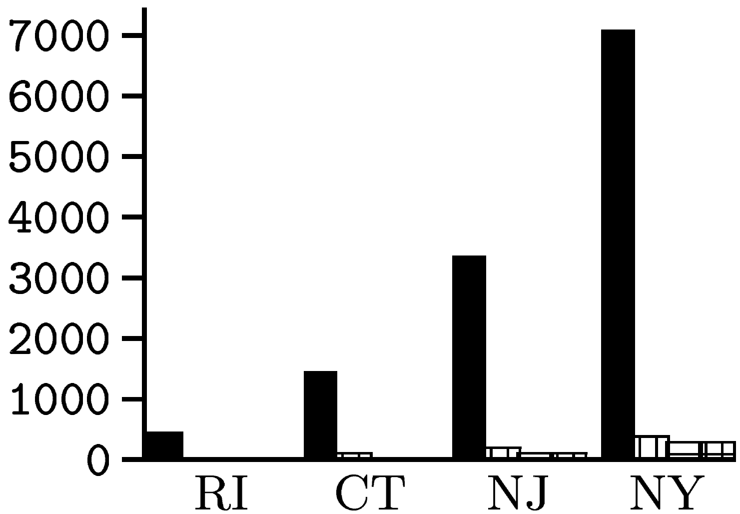
- Small overlap or perimeter desirable.
- Several insert/split heuristics (R$^+$-trees, R*-trees, Hilbert trees, ...) have been proposed, surveyed in [G89, GG98].

Constructing ("Bulk Loading") an R-tree

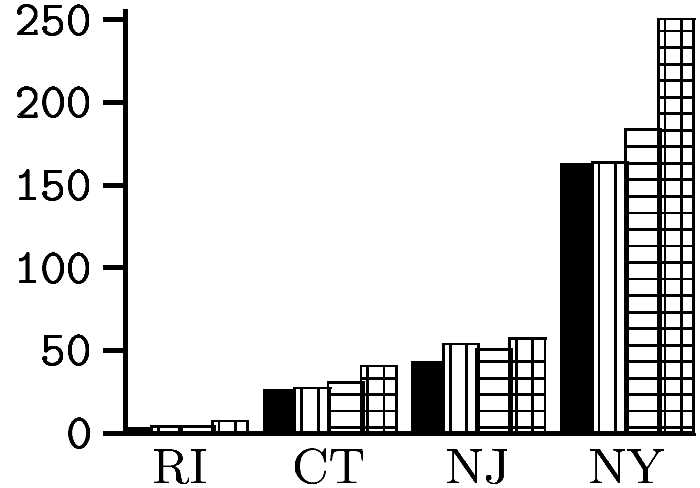
- ☆ Using repeated insertion takes $O(N \log_B n)$ I/Os.
- ☆ Bottom-up algorithms [RL85, KF93, DKLPY94, LLE96, vdBSW97]
  - Rectangles are sorted (using space-filling curve)
    $\Longrightarrow \quad O(n \log_m n)$-I/O algorithm.
  - Can only handle construction—not e.g. "bulk updates."
  - Questionable 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.

Experimental Results: R-tree

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



Building R-tree on road data
(I/Os in thousands)



Query with (1/10) hydro data
(I/Os in thousands)

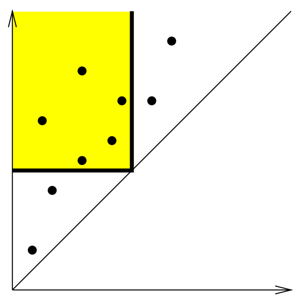
☆ Naive repeated insertion: ■

☆ Buffer: Size of buffer $\frac{B^2}{4}$ $\frac{B^2}{2}$ $2B^2$

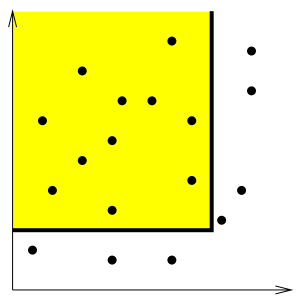
Experimental Results: I/O Costs for R-trees

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

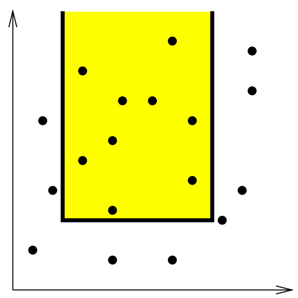
Online Range Searching



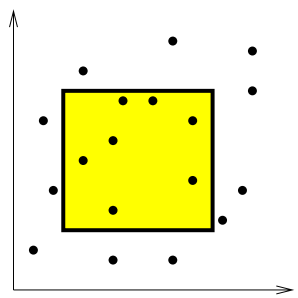
diagonal corner query



2-sided query

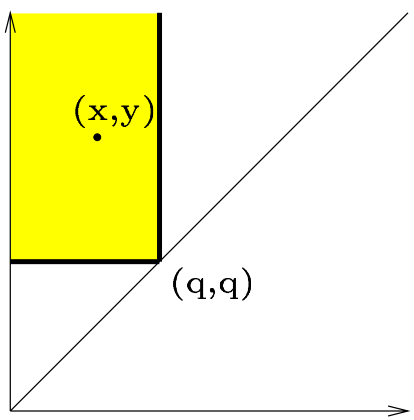


3-sided query

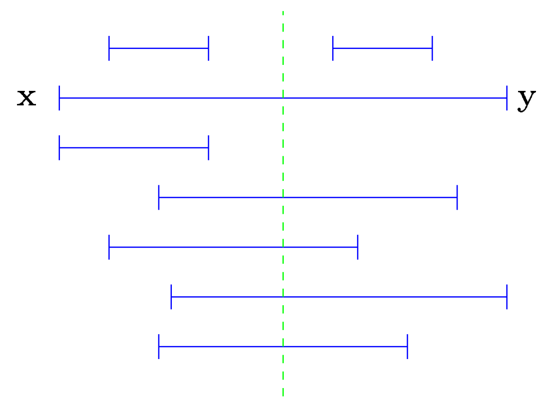


4-sided query

For example, indexing constraints in constraint query languages:



diagonal corner query



stabbing query with q

External Range Searching Results

☆ *Corner (Interval tree):* $O(n)$ space, $O(\log_B n + z)$ I/Os query, $O(\log_B)$ 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, $O\big(\log_B n + z + IL^*(B)\big)$ I/Os query, $O\big(\log_B n + \frac{1}{B}(\log_B n)^2\big)$ I/Os amortized updates [SR95]

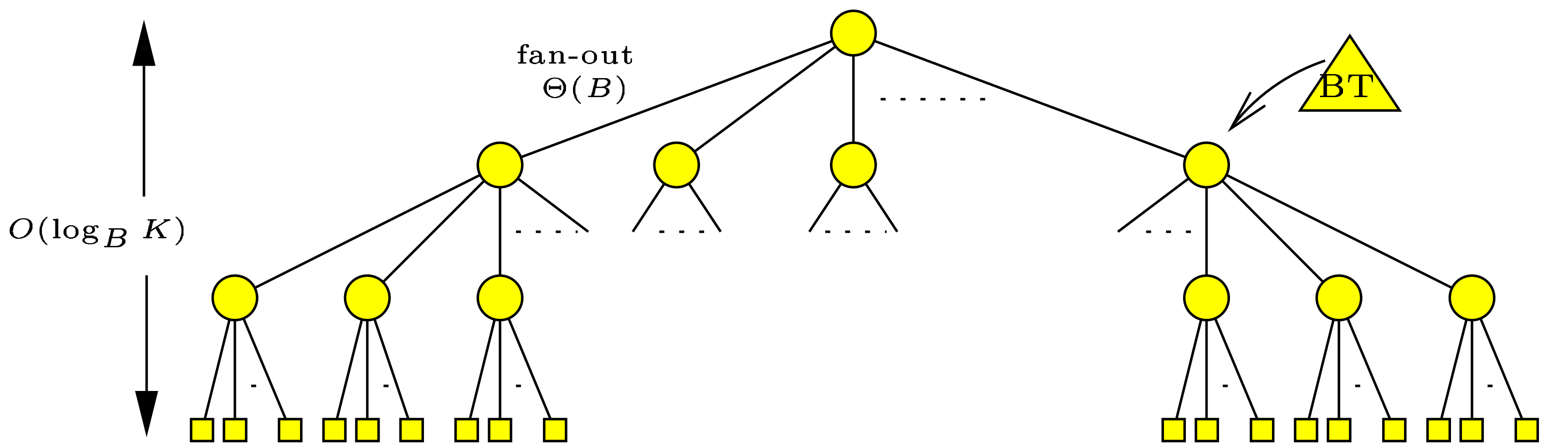
☆ *4-sided:* $O\big(n(\log N)/\log\log_B n\big)$ space, $O(\log_B n + z + IL^*(B))$ I/Os query [SR95]

☆ 3-d range queries: $O\big((\log\log\log_B n)\log_B n + z\big)$ 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 $O\big(n(\log n)/\log\log_B n\big)$ space, $O\big((\log_B n)^c + z\big)$ I/Os query.

SB-tree: a String B-tree [FG95]



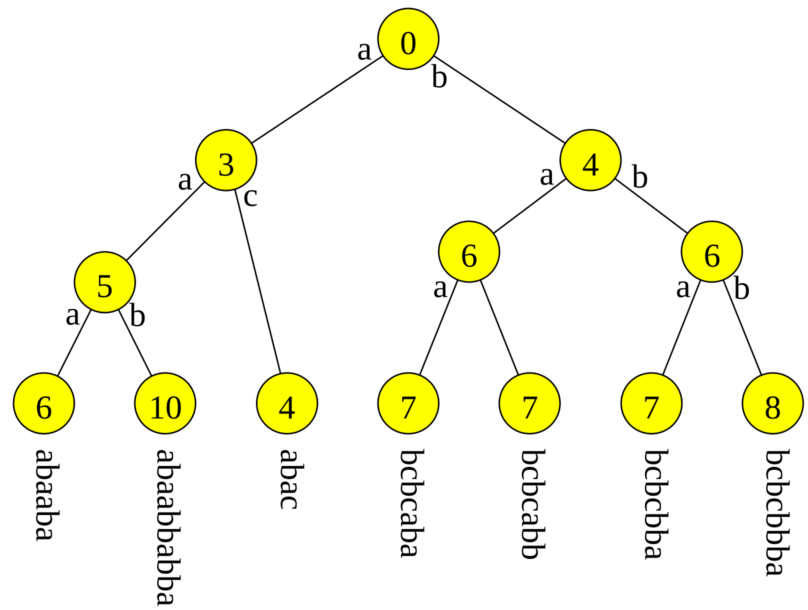
☆ B-tree on set of pointers to strings (in lexicographical order).

☆ Update possible in $O\left(\log_B K + \frac{|S|}{B}\right)$ I/Os.

☆ Pointers to $\Theta(B)$ strings associated with internal node.

- Need to be able to find position of string S efficiently.
- Every internal node is a Blind Trie (BT).

Blind Trie [Ajtai et al 84]



- ☆ Blind Trie on set of $B-1$ strings has $< 2B$ characters.
- ☆ Blind Trie contains characters from all $B-1$ strings.

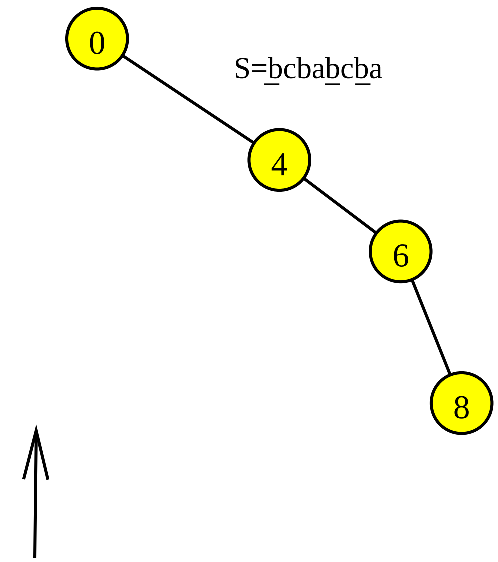
Blind Trie [Ajtai et al 84]

a b b c b c

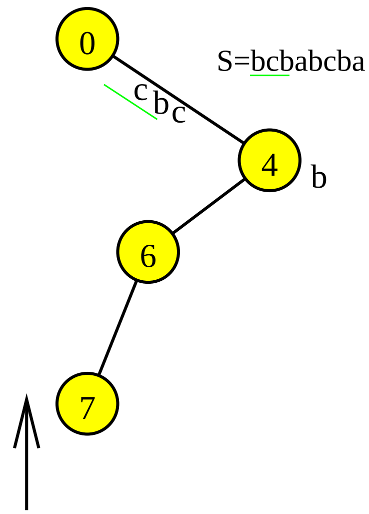
b b b

a a

Blind Trie [Ajtai et al 84]



Blind Trie [Ajtai et al 84]



Sorting K Strings

☆ Internal memory:

- Three-way quick sort [BS97].
- $\Theta(K \log_2 K + N)$ time.

☆ External memory:

- $\Theta(k \log_m k + n)$ I/Os?
- *No!* $\Theta(k \log_m k + n)$ I/O bound is too low.
  Actual I/O bound depends on
  * Whether breaking string into characters is allowed.
  * Number K_2 of long strings.
  * Total length n_1 of short strings.
- Use "lazy trie" or SB-tree.

Note: Comparison model

External Sorting of K Strings

	Model A *Strings indivisible*	**Model B** *Strings indivisible in external memory*
Long:	$\Theta(K_2 \log_m K_2 + n_2)$	$\Theta(K_2 \log_M K_2 + n_2)$
Short:	$\Theta(n_1 \log_m n_1)$	$O\bigl(\min\{K_1 \log_M K_1, n_1 \log_m n_1\}\bigr)$

K_1 = # of short strings ($|\text{length}| < B$).

K_2 = # of long strings ($|\text{length}| \geq B$).

N_1 = total # of characters in short strings; $n_1 = N_1/B$.

N_2 = total # of characters in long strings; $n_2 = N_2/B$.

Note: $n_1 < K_1$ and $n_2 \geq K_2$.

External Sorting of K Strings

☆ Upper Bounds:

- Model A: Merge sort with “lazy” Trie in internal memory.
- Model B: Using Buffer-tree technique on SB-tree [FG95].

☆ Lower Bounds:

- Short strings:
  - ∗ Consider “equal length case” (all strings of length $\frac{N_1}{K_1}$).
  - ∗ Use counting argument [AV88].
  - ∗ In model B use “permutation counting argument”.
- Long strings: Look at first B characters.

Dynamic Memory Allocation Model

[Barve & Vitter 98]

- ☆ EM algorithm is allocated memory in an allocation sequence $\sigma = m_1, m_2, \ldots$ of *allocation phases.*

- ☆ ith phase: Algorithm owns m_i blocks of memory for $2m_i$ I/Os.

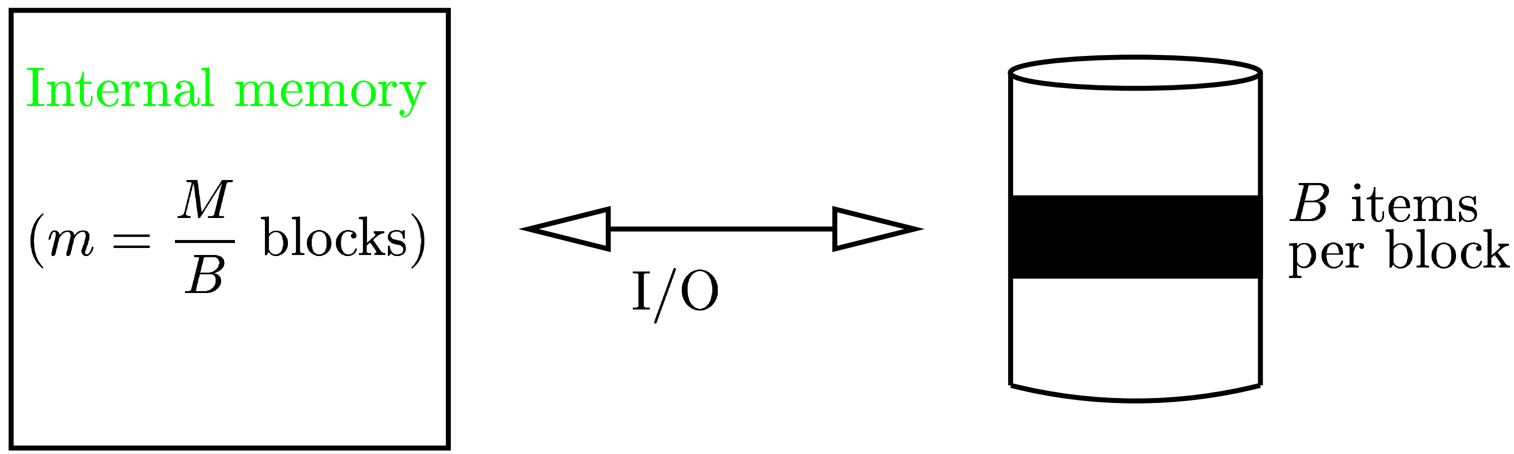
- ☆ *Dynamically Optimal Algorithms:*

  - Adversary chooses allocation sequence $\sigma = m_1, m_2, \ldots$
  - Suppose that $\mathcal{A}$ solves problem $\mathcal{P}$ during σ.
  - $\mathcal{A}$ is dynamically optimal for $\mathcal{P}$ iff no other algorithm $\mathcal{A}'$ can solve problem $\mathcal{P}$ more than a constant number of times during σ.

- ☆ Mergesort based on [PCL93] is general but nonoptimal.

- ☆ Dynamic memory model considered in [ZL93] is not general.

Traditional I/O Lower Bound for Sorting

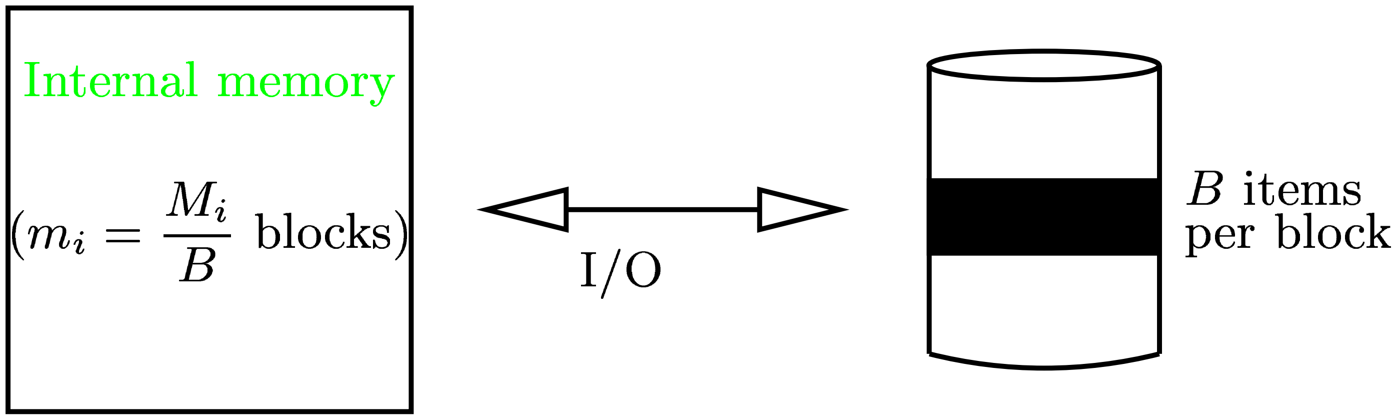


$$\begin{array}{c}\#\text{ possible outcomes}\\ \text{to comparisons per I/O}\end{array} = \begin{cases} B! \times \binom{M}{B} & \text{if previously unread block} \\ \binom{M}{B} & \text{otherwise.} \end{cases}$$

$$(B!)^{N/B} \binom{M}{B}^{T} \geq N! \quad \Longrightarrow \quad T \log m = \Omega(n \log n)$$

Dynamic Memory Lower Bound for Sorting

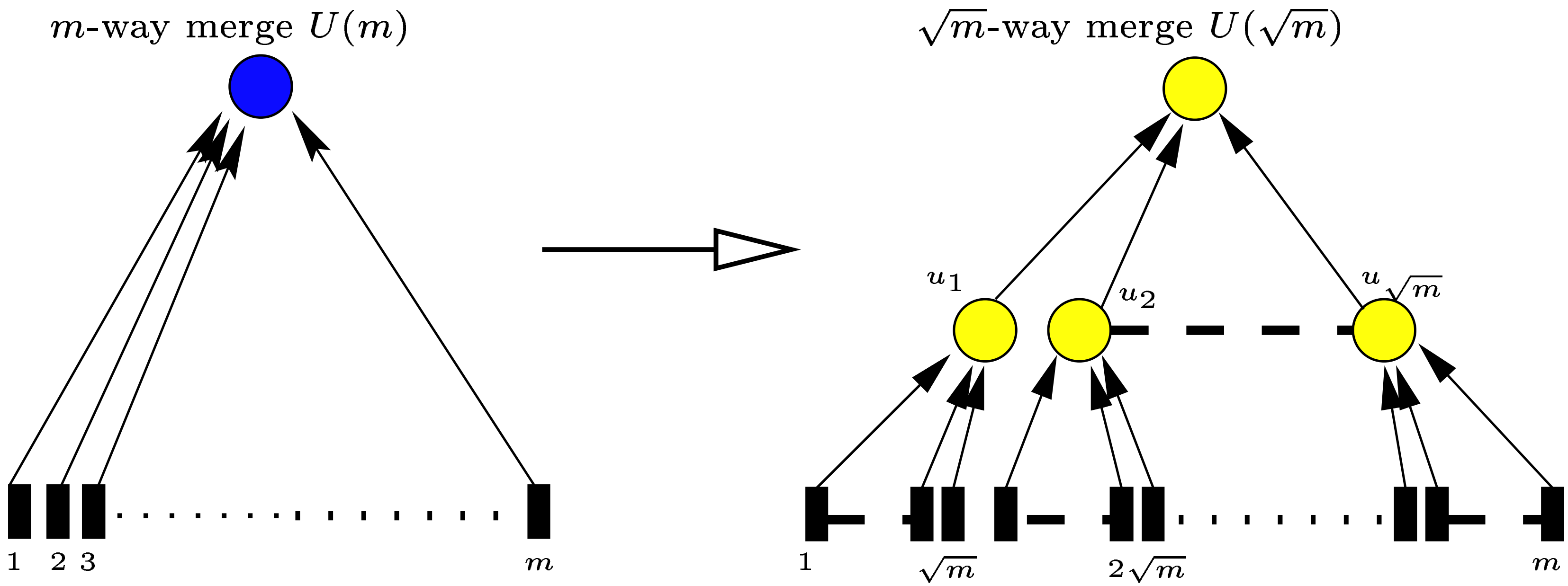
ith phase:



$$\begin{array}{c} \#\text{ possible outcomes} \\ \text{to comparisons per I/O} \end{array} = \begin{cases} B! \times \binom{M_i}{B} & \text{if previously unread block} \\ \binom{M_i}{B} & \text{otherwise.} \end{cases}$$

$$(B!)^{N/B} \prod_i \binom{M_i}{B}^{2m_i} \geq N! \implies \sum_i 2m_i \log m_i = \Omega(n \log n)$$

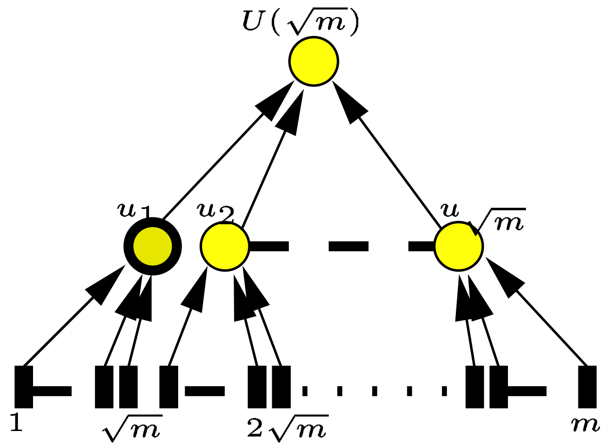
Data Structure $\Delta_{[\sqrt{m},m]}$



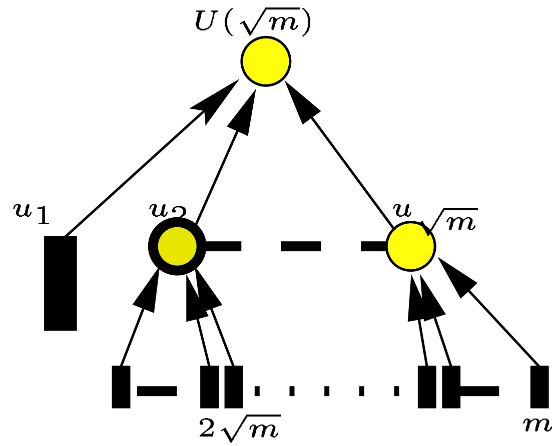
Data structure $\Delta_{[\sqrt{m},m]}$

- ☆ $\Delta_{[\sqrt{m},m]}$ is an organization of an m-way merge in terms of $\sqrt{m}$-way merge operations.
- ☆ If the allocation size m_i is in $[\sqrt{m}, m]$, we execute a $\sqrt{m}$-way merge operation "from" $\Delta_{[\sqrt{m},m]}$.

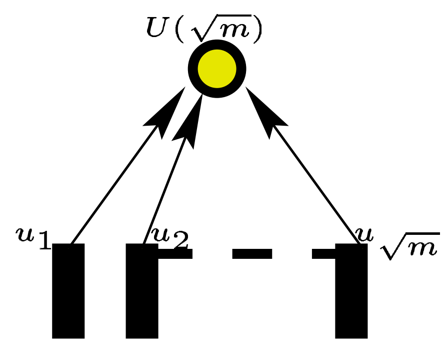
Allocation size $[\sqrt{m}, m]$'s active merge



u_1 is the active merge.

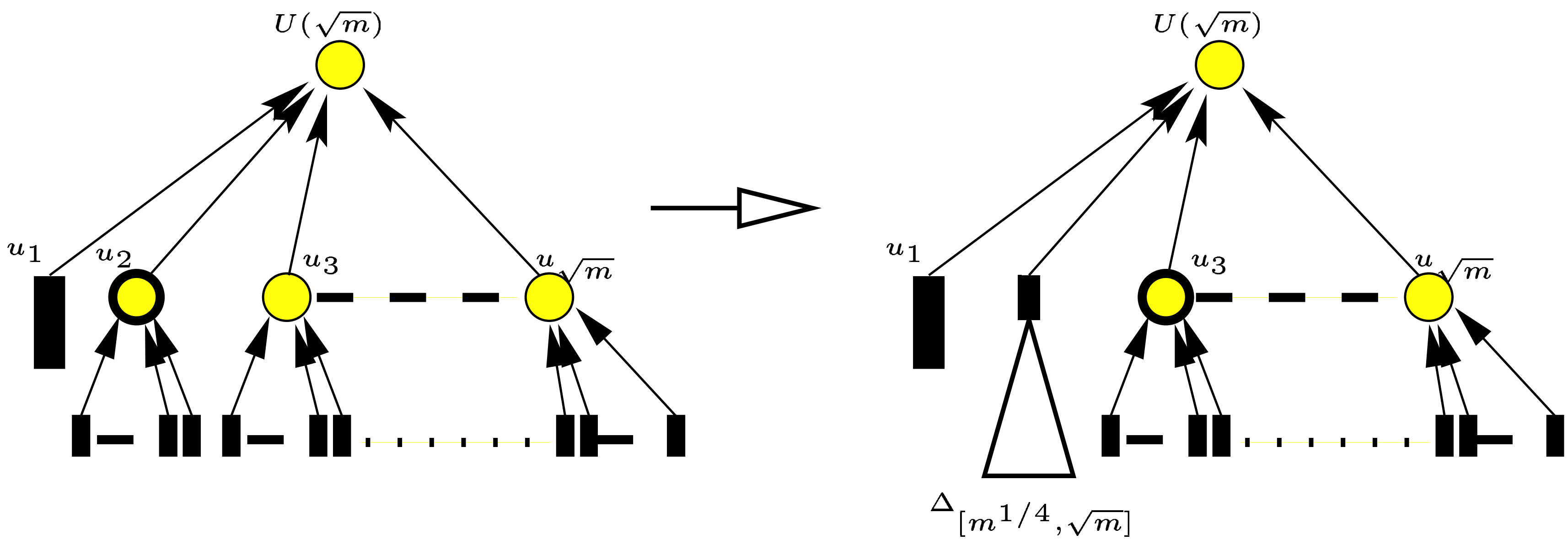


u_2 is the active merge.



$U(\sqrt{m})$ is the active merge.

Alloc Size drops from $[\sqrt{m}, m]$ to $[m^{1/4}, \sqrt{m}]$



In response, transform the active merge u_2 into data structure $\Delta_{[m^{1/4},\sqrt{m}]}$ for allocation sizes $[m^{1/4}, \sqrt{m}]$.

Conclusions and Open Problems

- ☆ Indivisibility assumption.
- ☆ New models, clusters of workstations, memory hierarchies, ...
- ☆ TPIE, see http://www.cs.duke.edu/TPIE/
- ☆ Handling many disks, large merge orders, many partitioning elements, large fanouts. (Don't use square root trick.)
- ☆ Fundamental graph problems (e.g. topological sorting, BFS, DFS, connectivity).
- ☆ Online dynamic data structures (e.g. dynamic point location, range-searching).
- ☆ GIS applications (e.g. practical red-blue line segment intersection, spatial join).
- ☆ Support of indexing/data structures (e.g. implementation of fundamental structures, contour line structure, point location).
- ☆ String processing.
- ☆ Dynamic memory allocation.