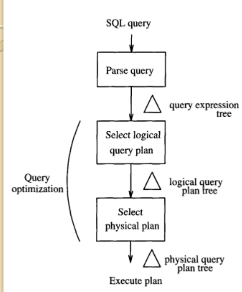
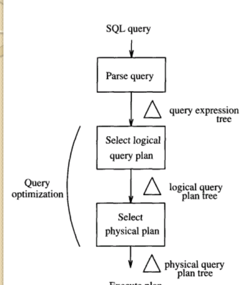
Evaluating Queries

Query Processing

Query Processing: Overview



Query Processing: Example



select lname
from employee
where ssn = '123456789';

query expression tree?

π_{lname}
|
$\sigma_{ssn='123456789'}$
|
employee

physical query? (have operators: ROWACCESS, FILESCAN, INDEXSCAN)

Classical Example: Sorting

- why?
  - ORDER BY
  - duplicate removal (intersection, union, DISTINCT)
  - sort/merge for join
- how?
  - in main memory easy: quick sort
  - issue: large relations don't fit in main memory

Scheduling

- scheduling of access is important
- Example:
  - 3 pages of main memory,
  - least recently used replacement policy
  - pages in main memory:
    - page 1: 1, 5, 9, …, 37
    - page 2: 2, 6, 10, …, 38
    - page 3: 3, 7, 11, …., 39
    - page 4: 4, 8, 12, …, 40
  - how many page accesses to read 1-40?

Another Query Plan Example

select e.name, s.name
from employee as e, employee as s
where e.superid = s.id

QEP?

External Sorting I

- goal: minimize page transfers
- assumptions:
  - data stored in n pages
  - m << n pages fit into main memory
- solution: sort in runs (temporary sorted subfiles that get merged on disk)

External Sorting: Algorithm

- partition input file into blocks of m pages
- sort internally, write-out into n/m initial "runs"
- at each level of the recursion
  - merge m-1 runs into a new run
  - use 1 page in main memory for each run
  - 1 page in main memory to create merged page
  - write output page if full/reload input page when processed

External Sorting: Performance

- analyze set-up phase
- how many page I/Os at each recursive level?
- how many recursive levels?
- overall analysis?
- internal sort?

Rectangle Intersection, Again

- original solution: sweepline
  - event list: left/right x-coordinates of rectangles
  - active list: rectangles at current x-value
- needed for spatial join (on overlap)
- external technique: distribution sweeping

Orthogonal Line Segments

input: S set of vertical/horizontal line segments

output: pairs of intersecting segments

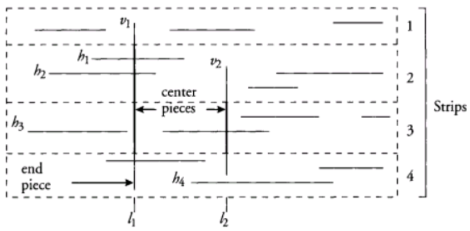
sweepline algorithm:

- events: x-min coordinates
- active list: horizontal segments at x-value
- when vertical segment is encountered: range query
- works in time O(n log n + k)

Orthogonal line segments (EM)

- assumption
  - n pages of data
  - m buffer pages in main memory
- external sort (x-min): $O(n \log_m n)$
- split into m horizontal strips of n/m horizontal segments each
- one active list (stored externally) for each slab (can be read in parallel with others one block at a time, since we have m pages in main memory)

Process center pieces



- why can't we process end-pieces the same way?
- what to do about end-pieces?

End-pieces

- apply method recursively
- analysis
  - initial set-up
  - each recursive level
  - depth of recursion?
- overall running time analysis

Rectangle Intersection

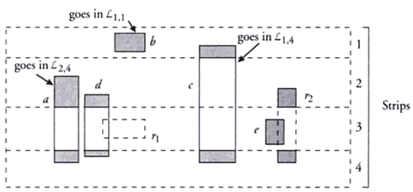
input: B, R sets of rectangles
output: all (b,r) in BxR with b intersecting r

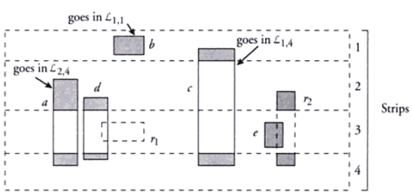
solution:

- similar to line segment problem, separate lists for red and blue rectangles
- but: rectangles don't fit into strips, they can span multiple strips
- problem: intersection between (b,r) might be reported multiple times
- so naïve adaptation gives factor of m

Rectangle Intersection

- Solution:
  - separate list for each interval of strips and each color: $\mathcal{L}^b_{h,k}$ $\mathcal{L}^r_{h,k}$





- If r consists of a single end portion contained in strip i, we scan the blue lists $\mathcal{L}^b_{h,k}$ such that $h < i < k$. In each list, *all* blue rectangles b with $r.x_{min} < b.x_{max}$ do intersect r. The bs such that $r.x_{min} > b.x_{max}$ can be removed from the list. In addition, r is inserted in $\mathcal{L}^r_{i,i}$.

  For instance, rectangle r_1 in Figure 7.6 is inserted in $\mathcal{L}^r_{3,3}$, and we scan the blue lists $\mathcal{L}^b_{h,j}$ for $h < 3$ and $j > 3$. The list $\mathcal{L}^b_{2,4}$ is scanned and an intersection with the blue rectangle d is reported.
- If r contains a center portion over the strips $i, \ldots j$, we scan all lists $\mathcal{L}^b_{h,k}$ with $i \leq h$, $j \geq k$ and compute the intersections. In addition, r is inserted in $\mathcal{L}^r_{i-1,j+1}$.

  In Figure 7.6, rectangle r_2 has a center portion that spans strip 3. The list $\mathcal{L}^b_{3,3}$ must be scanned and the intersection with e is reported.

Analysis?

Spatial Join

- join on: topology (overlap, disjoint, contain, …), geometry (distance, direction)
- consider overlap only
  - filter: overlap of mbbs
  - refinement: overlap of geometries
- depends on indexes available

Spatial Join Algorithms

- no indexes
  - distribution sweep
  - hash-join algorithm
- single index
  - INL (indexed nested loop)
- two R-tree indexes
  - synchronized tree traversal
- two linear trees

single index: INL

for each o in non-indexed relation
  perform range query with o.mbb
  on indexed relation

no index

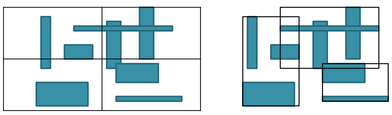
hash-join algorithm

- assume buckets fit into main memory
- hash keys of relations R, S into buckets
- load smaller bucket, compare to corresponding bucket

Example: R: 2000 records, S: 500 records
hash into 100 buckets
page I/O? (read R, S, write buckets, join)

no index for spatial data

- hash-join depends on join condition being equality: overlapping rectangles won't hash to same bucket
- solution:
  - buckets determined by rectangles
    - may overlap (no redundancy) or be disjoint (redundancy)



hash-join (overlapping)

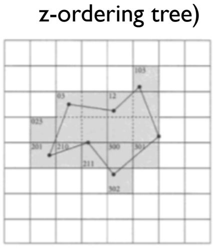
1. partition R
   - all buckets roughly same size
   - buckets should fit into main memory
   - minimal overlapping
2. assign rectangles of S to buckets of R
   - S rectangles might be duplicated
3. join buckets (load smaller bucket into main memory, scan other bucket)

(a) Objects from set *R* (build input) in three partition buckets

(b) Filtering and replication of objects from set *S* (probe input)

joining two linear trees

- raster trees: traditional join
- general linear tree (e.g. linear quadtree or z-ordering tree)

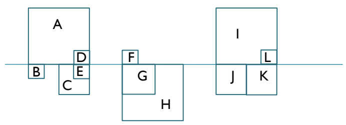


Property:
C_z is contained in $C_{z'}$
if and only if
z' is prefix of z

can use to test overlap

joining two z-ordering trees

- replace each entry (z, oid) with intervals (z, ssc(C_z)), where ssc(C_z) is lower-right corner of C_z
- two squares overlap iff their intervals overlap
- store each list in a stack



R-trees

- naïve recursion
- restricted recursion
- sweep-line

R-trees recursively

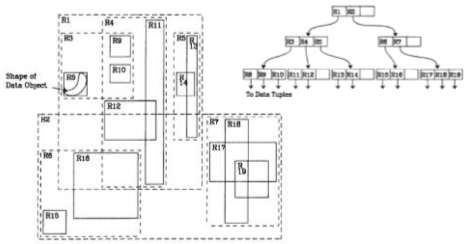
- STT (Synchronized Tree Traversal)

```
STT (Node N1, Node N2): set of pairs of ids

begin
  result: set of pairs of ids, initially empty
  for all e1 in N1 do
    for all e2 in N2 such that e1.mbb ∩ e2.mbb ≠ ∅ do
      if (the leaf level is reached) then
        result += {(e1, e2)}
      else
        N'1 = READPAGE (e1.pageID); N'2 = READPAGE (e2.pageID);
        result += STT (N'1, N'2)
      end if
    end for
  end for
  return result
end
```

- I/O performance ok
- CPU cost high

R-trees recursively, improved



- STT(R1, R2)
- do we need to look at every combination of {R3, R4, R5} and {R6, R7}?

R-trees, sweepline

- why not use red/blue intersection algorithm we saw earlier?
- Asymptotics vs constants

- greedy approach:

order red/blue sets

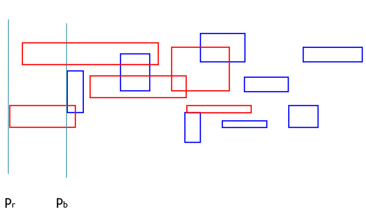
keep picking leftmost rectangle r

keep testing rectangles s of opposite color so that

s.xmin < r.xmax

remove leftmost rectangle

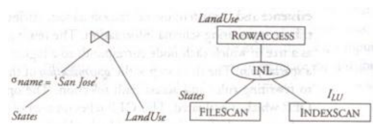
Example



- analysis (bad case?)

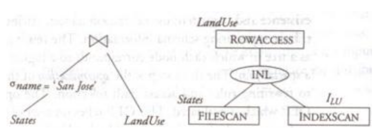
Building Query Execution Plans

select intersect(l.shape, c.shape)
from county c, land_use l
where c.county_name = 'San Jose'
and overlaps(l.shape, c.shape);



pipelined execution possible:

Iterators

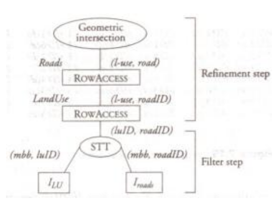


pipelined execution possible
- iterators (open, next, close)
- e.g. rowaccess, retrieve one record at a time

issues:
- refinement not included yet
- CPU time plays a role

Example

- spatial join between roads and land-use
- both relations have R-tree



Problems:
- random access to records (high I/O)
- repeated access to same records

- so refinement needs to be done carefully

Sequencing

lID 1(u)	rID 1(b)	lID 1(u)	rID 1(b)	l1 rID 1(b)		l3 rID 3(a)	
lID 2(v)	rID 2(c)	lID 1(u)	rID 6(d)	l1 rID 6(d)		l5 rID 3(a)	
lID 3(w)	rID 3(a)	lID 8(u)	rID 1(b)	l8 rID 1(b)		l1 rID 1(b)	
lID 4 (v)	rID 4(d)	lID 2(v)	rID 2(c)	l2 rID 2(c)		l8 rID 1(b)	
lID 5(w)	rID 5(c)	lID 4(v)	rID 4(d)	l4 rID 4(d)		l2 rID 2(c)	
lID 1(u)	rID 6(d)	lID 3(w)	rID 3(a)	l3 rID 3(a)		l5 rID 5(c)	
lID 5(w)	rID 3(a)	lID 5(w)	rID 5(c)	l5 rID 5(c)		l1 rID 6(d)	
lID 8(u)	rID 1(b)	lID 5(w)	rID 3(a)	l5 rID 3(a)		l4 rID 4(d)	
(a)		(b)		(c)		(d)	

- assume 4 pages fit into main memory; look at schedule (a)

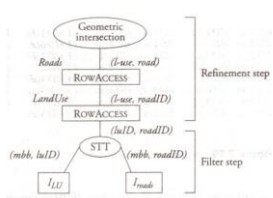
Sequencing: Segment Sort

lID 1(u)	rID 1(b)	lID 1(u)	rID 1(b)	l1 rID 1(b)		l3 rID 3(a)	
lID 2(v)	rID 2(c)	lID 1(u)	rID 6(d)	l1 rID 6(d)		l5 rID 3(a)	
lID 3(w)	rID 3(a)	lID 8(u)	rID 1(b)	l8 rID 1(b)		l1 rID 1(b)	
lID 4 (v)	rID 4(d)	lID 2(v)	rID 2(c)	l2 rID 2(c)		l8 rID 1(b)	
lID 5(w)	rID 5(c)	lID 4(v)	rID 4(d)	l4 rID 4(d)		l2 rID 2(c)	
lID 1(u)	rID 6(d)	lID 3(w)	rID 3(a)	l3 rID 3(a)		l5 rID 5(c)	
lID 5(w)	rID 3(a)	lID 5(w)	rID 5(c)	l5 rID 5(c)		l1 rID 6(d)	
lID 8(u)	rID 1(b)	lID 5(w)	rID 3(a)	l5 rID 3(a)		l4 rID 4(d)	
(a)		(b)		(c)		(d)	

- k: number of pairs (Lx, RIDy) that fit into m-1 pages
- load k pairs (LIDx, RIDy) into m-1 pages
- sort on LID, access land-use replace LID with L records
- sort on RID, load records from Road using mth page, perform refinement step for each record

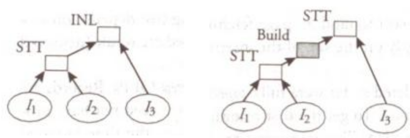
Why Not

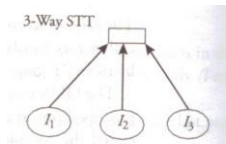
- sort (LID, RID) after STT ?



- means we can't pipeline: sorting is a *blocking* operator

Multiway Joins





Sources

- Garcia-Molina, Ullman, Widom, Database Systems; the complete book, Pearson, 2009.
- Vassilakopoulos, Papadopoulos, Spatial databases, IGI, 2005