

Lecture 7

10/1/09

Lab 1 Due Tomorrow

Lab 2 posted tonight.

2008/2009 SNAFU

Who downloaded 2008 code?

Team Meetings Next Week

Continuing with our discussion of access methods.

Seen heap files and hash files.

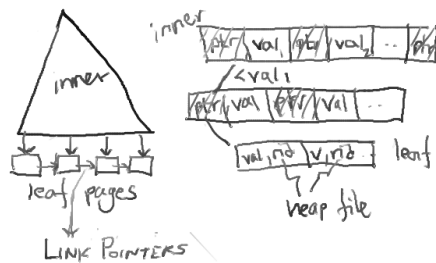
Today B+Trees.

R*Trees, briefly.

Buffer pool.

B+-Trees:

Show example tree:



search for records with specific value descends into only 1 page

Discussion points:

- what is point of link pointers
- why not store data in intermediate nodes
- page occupancy (why does this matter)

- key size?
- balanced (why important?)
- how many levels in practice
 - disk page = 4096 bytes; values 4 bytes; ptrs 4 bytes = 512 ptrs / node --> $512^4 = 68$ billion

- logn for lookup/insert/delete; how many i/os in practice?
(1, maybe 2)

- scan is random
- node format
 - "Fill factor" percentage of each page that is used.
 - Every except root node is at least 50% full

Why would we not want this to be 100%?

Ideally, 67% full (where from)?

Can be clustered or unclustered
 clustered means that data is physically stored in order of index
 often, leaves of index contain actual data records

R*-trees:

(why are we discussing)

(what is idea)

fill factors

What does an R-tree do?

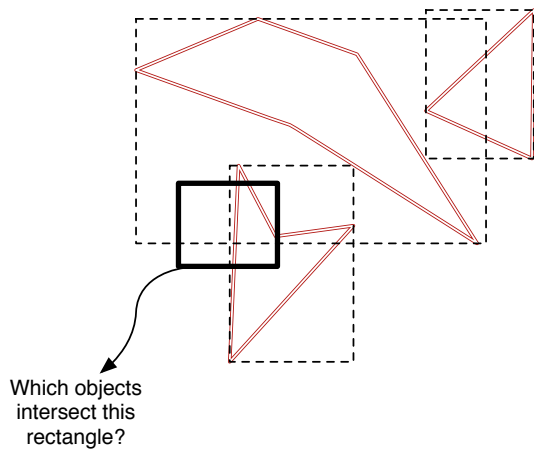
indexes geo-spatial data

Why are R-trees important?

very important in all sorts of GIS applications

Why aren't B-Trees sufficient?

imagine storing lat/long coordinates as the key of B+Tree
 suppose I want to find the which objects intersect a point



B+Tree is a 1-D structure -- can't easily use it B+Tree to figure out which bounding boxes might overlap a multidimensional object (lots of conditions, so of which return many results.)

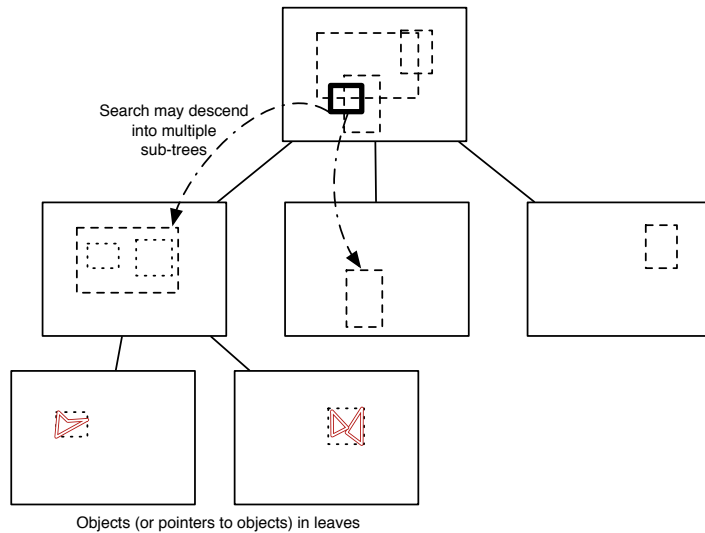
How does it do that?

by storing bounding boxes.

What kinds of queries do we want to support?

containment, overlap, equality, etc.

Simple illustration of structure:
 (note that queries will need to go to multiple subtrees)
 (so some kind of 'heuristics' to get good subtrees are essential)
 (show what pages look like?)



Intuitively, what would we want in an R-Tree?

- Little overlap between rectangles
- Relatively tight bounding boxes
- Little storage overhead

What do they mean by "engineering approach"?

(They pick algorithms by trying a huge number of possibilities.)

Need:

- chooseSubTree
 - where to insert
 - add to subtree with minimum overlap enlargement (then minimum area enlargement)

why does overlap matter more than area?

claim its for "robustness"

helps for small query rectangles on r-trees with non-uniformly distributed small objects

(overlap is bad b/c it means you will have to explore more subtrees)

- split
 - how to reallocate nodes between buckets for each axis
 - sort by lower then upper on each axis
 - try all groups of size at least m
 - choose axis that minimizes sum of margins (perimeter) of groups (long and skinny is bad, b/c it increases the probability of overlap in future inserts_

- once axis has been selected, choose distribution with the minimum overlap value (same reason as insert)

what happens on delete?

What are the innovations offered by the R* authors:

- overlaps and perimeters matter more than areas (minimizing area can lead to long skinny boxes that overlap everything!)
- reinsertions help increase storage utilization (a lot), and don't cost that much in practice. (can make "bad" group decisions early on) (idea is not to reload everything, but to reload a fraction of the rectangles periodically)

So what affects the performance of an indexing method:

- coverage (bad lossy subtree predicates mean we have to explore lots of the tree)
- poor clustering of data into leaves (doesn't match query search patterns)
- poor space utilization (big keys, underfull pages)

why not use something other than rectangles?

(lookup quality vs. compactness.)

DBMIN

Buffer pool:

cache of recently used pages

convenient "bottleneck" through which references to underlying pages go
useful when checking to see if locks can be acquired or not

Shared between all queries running on the system.

Diagram:

pg id	lock	ptr
1	R/W, TID 2	0xABCD
2	...	0xCDEF

Cache -- so what is the best eviction policy?

LRU? What if a query just does one sequential scan of a file -- then putting it in the cache at all would be pointless. So you should only do LRU if you are going to access a page again, e.g., if it is in the inner loop of a NL join.

For the inner loop of a nested loops join, is LRU always the best policy? No, if the inner doesn't fit into memory, then LRU is going to evict the record over and over.

E.g., 3 pages of memory, scanning a 4 page file:

pages	A	B	C	read	hit/miss
	1			1	m
	1	2		2	m
	1	2	3	3	m
	1 4	2	3	4	m
	1 4	2 1	3	1	m
				2	m

Always misses?. What would have been a better eviction policy? MRU!

pages	A	B	C	read	hit/miss?
	1			1	m
	1	2		2	m
	1	2	3	3	m
	1	2	3 4	4	m
	1	2	4	1	h
	1	2	4	2	h
	1	2 3	4	3	m
	1	3	4	4	h
	1	3	4	1	h
	1 2	3	4	2	m

Here, MRU hits 2/3 times.

DBMIN tries to do a better job of managing buffer pool by

- 1) allocating buffer pools on a per-file-instance basis, rather than a single pool for all files
- 2) using different eviction policies per file

What is a "file instance"?

(Open instance of a file by some access method.)

Each time a file is opened, assign it one of several access patterns, and use that pattern to derive a buffer management policy.

(What does a policy consist of?)

Policy for a file consists of a number of pages to allocate as well as a page replacement policy.

(What are the different types of policies?)

Policies vary according to access patterns for pages. What are the different access patterns for pages in a database system?

SS - Straight Sequential (sequential scan)

~~CS - Clustered Sequential (merge join) (skip)~~

LS - Looping sequential (nested loops)

SR - Straight Random (index scan through secondary index)

~~CR - Clustered Random (index NL join with with secondary index on inner, with repeat foreign keys on outer)~~
(skip)

SH - Straight Hierarchical (index lookup)

LH - Looping Hierarchical (repeated btree lookups)

So what's the right policy:

SH - 1 page, any access method

CS - size of cluster pages, LRU

LS - size of file pages, any policy, or MRU plus however many pages you can spare

SR - 1 page, any access method

CR - size of cluster pages, LRU

SH - 1 page, any access method

LH - top few pages, priority levels, any access method for bottom level

How do you know which policy to use?

(Not said, presumably the query parser/optimizer has a table and can figure this out.)

Multipage interactions.

Diagram:

Buffer pool per file instance, with locality set for that instance, plus "global table" that contains all pages.

Each page is "owned" by at most one query. Each query has a "locality set" of pages for each file instance it is accessing as a part of its operation, and each locality set is managed according to one of the above policies.

Also store current number of pages associated with a file instance (r) and the maximum number of pages associated with it (l).

How do you determine the maximum number of pages?

Using numbers above.

What happens when the same page is accessed by multiple different queries?

- 1) Already in buffer pool and owned locally
- 2) Already in buffer pool, but not owned

- a) If someone else owns, nothing to be done
 - b) If no owner, requester becomes owner
- 3) Not in buffer pool - requester becomes owner, evict something from requester's memory

How do you avoid running out memory?

Don't admit queries into the system that will make the total sum of all of the I_{ij} variables > total system memory.

Metacomments about performance study.

(It's good.) Interesting approach. What did they do?

Collect real access patterns and costs, use them to drive a simulation of the buffer pool.

(Why?) Real system would take a very long time to run, would be hard to control.

How much difference did they conclude this makes?

As much as a factor of 3 for workload with lots of concurrent queries and not much sharing. Seems to be mostly due to admission control. With admission control, simple fifo is about 60% as good as DBMIN.

DBMIN is not used in practice.

What is? (Love hate hints).

What's that? (When an operator finishes with a page, it declares its love or hate for it. Buffer pool preferentially evicts hated pages.)

Not clear why (this would make a nice class project.)

Perhaps love hate hints perform almost as well as DBMIN and are a lot simpler. They don't capture the need for different buffer management policies for different types of files.

(What else might you want the buffer manager to do?)

Prefetch.

(Why does that matter.)

Sequential I/O is a lot faster. If you are doing a scan, you should keep scanning for awhile before servicing some other request, even if the database hasn't yet requested the next page.

Depending on the access method, you may want to selectively enable prefetching.

Interaction with the operating system

(What's the relationship between the database buffer manager and the operating system?)

Long history of tension between database designers and OS writers. These days databases are an important enough application that some OSes have support for them.

(What can go wrong?)

- Double buffering -- both OS and database may have a page in memory, wasting RAM.
- Failure to write back -- the OS may not write back a page the database has evicted, which can cause problems if, for example, the database tries to write a log page and then crashes.
- Performance issues -- the OS may perform prefetching, for example, when the database knows it may not need it.

Disk controllers have similar issues (cache, performance tricks.)

(What are some possible solutions?)

- Add hooks to the OS to allow the database to tell it what to do.
- Modify the database to try to avoid caching things the OS is going to cache anyway.

In general, a tension in layered systems that can lead to performance anomalies.