

Lecture 6

9/29/09

Project Teams Due Today

Those of you who don't have groups -- send us email!

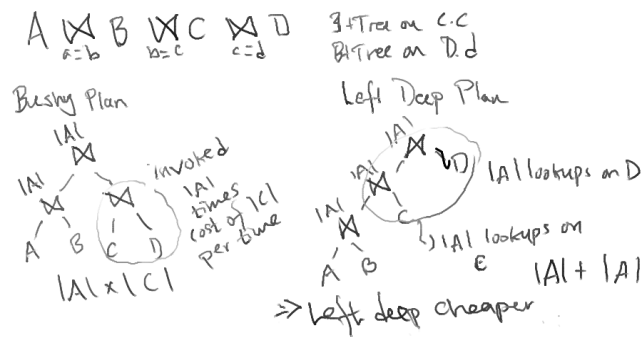
Sample Projects Posted

Lab 1 Due Tomorrow

Team Meetings Next Week

Continuing with our discussion of database internals.

Last time:



access methods

query execution (iterator model)

Buffer management and storage system:

What's the "cost" of a particular plan?

CPU cost (# of instructions)	- 1 ghz == 1 billions instrs / sec, 1 nsec / instr
I/O cost (# of pages read, # of seeks)	- 100 MB / sec = 10 nsec / byte
Random I/O = page read + seek	- 10 msec / seek = 100 seeks / sec

Random I/O can be a real killer (10 million instrs/seek) .

When does a disk need to seek?

Which do you think dominates in most database systems?

(Depends. Not always disk I/O. Typically vendors try to configure machines so they are 'balanced'. Can vary from query to query.)

For example, fastest TPC-H benchmark result (data warehousing benchmark), on 10 TB of data, uses 1296 74 GB disks, which is 100 TB of storage. Add'l storage is partly for indices, but partly just because they needed add'l disk arms. 72 processors, 144 cores -- ~10 disks / processor!

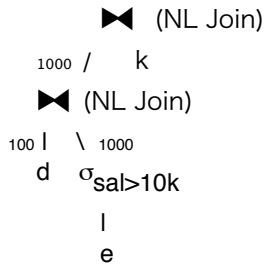
But, if we do a bad job, random I/O can kill us!

100 tuples/page	select * from
10 pages RAM	emp, dept., kids
10 KB/page	where e.sal > 10k

10 ms seek time
 100 MB/sec I/O

emp.dno = dept.dno
 e.eid = kids.eid

ldeptl = 100 = 1 page
 lempl = 10K = 100 pages
 lkidsl = 30K = 300 pages



1st Nested loops join -- 100,000 predicate ops; 2nd nested loops join -- 3,000,000 predicate ops

Let's look at # disk I/Os assuming LRU and no indices

if d is outer:
 1 scan of d
 100 sec scans of e
 (100 x 100 pg. reads) -- cache doesn't benefit since e doesn't fit

1 scan of e: 1 seek + read in 1MB
 10 ms + 1 MB / 100 MB/sec = 20 msec
 20 ms x 100 depts = 2 sec

10 msec seek to start of d and read into memory

2.1 secs

if d is inner
 read page of e -- 10 msec
 read all of d into RAM -- 10 msec
 seek back to e -- 10 sec
 scan rest of e -- 10 msec, joining with d in memory

Because d fits into memory, total cost is just 40 msec

k inner:
 1000 scans of 300 pages
 3 / 100 = 30 msec + 10 msec seek = 40 x 1000 = 40 sec

if plan is pipelined, k must be inner

So how do we know what will be cached?

That's the job of the buffer pool.

Buffer pool is a cache for memory access. Typically caches pages of files / indices.

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

Since disk >> memory, this is a cache

Questions:

- eviction policy (LRU?) for NL inner that doesn't fit into RAM, is LRU the best idea?
- prefetching policy

Will revisit buffer pool management strategies in a few classes.

So we want to provide "access methods" that allow us to get the data we want with a minimum of I/Os.

These access methods are dictionary structures we are familiar with (e.g., hash tables, trees), that are specially built to be "external" -- that is, disk resident. These are indices.

As opposed to a file with records that we have to scan -- obviously, this is a win.

Hence, database system uses indices when it can.

Why do I say "when it can"?

(Some indices are good for different things -- e.g., hash tables don't support range queries)

Why not create indices on every attribute?

Update costs! (Indices are big, and so are typically stored on disk.)

So let's look at different types of access methods:

types of access methods

heapfiles

(sorted files)

index files

leaves as data (primary index)

leaves as rids (secondary index)

clustered vs. unclustered

types of indexes

hash files

b-trees

r-trees

...

tradeoffs between different storage representations
scan vs. search vs. insert vs. delete

N - number of records
P - pages in index/file
R - pages in range

(units are # I/Os)	Heap File	Hash File	B+Tree
Search	P/2	O(1)	O(log _b N)
Scan	P	-	O(log _b N) + R
Delete	P/2	O(1)	O(log _b N)

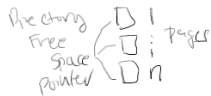
heap files

search cost ~ scan cost ~ delete cost

- linked list
- directory
- array of objs

file organization

pages
records
rids (page number, offset in record)



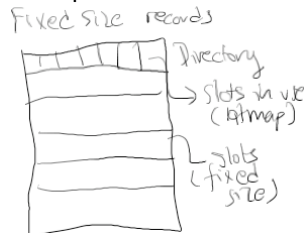
page layout

fixed length records
page of slots, with free bit map

"slotted page" structure for var length records
slot directory (slot offset, len)

big records?

example:



tuple layout

fixed length

variable length

field slots

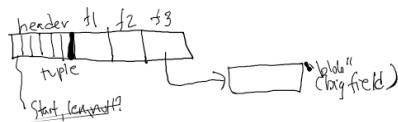
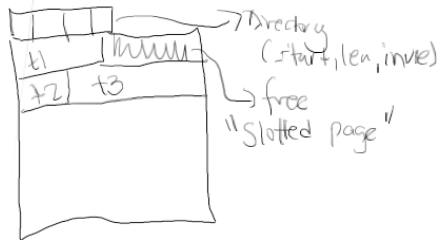
delimiters

half fixed/variable

null values?

example:

Variable sized records



hash files

search cost ≈ 1

insert cost ≈ 1

delete cost ≈ 1

range scan is equal to sequential scan

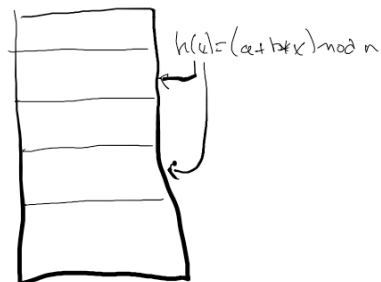
suppose we have a hash-table that for employees stored on disk hashed on the 'name' attribute.

$h(\text{name}) \rightarrow \{1..k\}$

$h(x) = (a + b*x) \bmod k;$

We have k buckets, and one page per bucket. Store to pages by hashing, appending the record to that page.

Then, if a query looks for 'morris',
we can find him in a single I/O (we know exactly where to go to find him.)



How many buckets do we need? (XXX)

What if we have too few buckets? (long chains that we have to follow)

Extensible hashing:

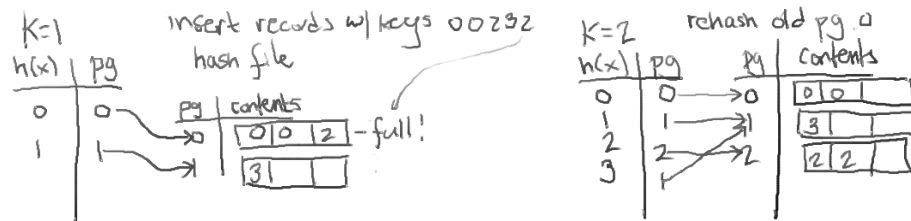
gave example before -- in principle, we can overflow a page, in which case we need an overflow chain. these overflow chains can get long and slow down the performance of our alg.

so, instead, we split hash buckets when they get full. do this with a family of hash functions (switch to the next level when we get too many in the current level).

define $h(v) = \{0, 1, \dots, b\}$
 $h_k(v) = h(v) \bmod 2^k$
 $h_1(v) = \{0, 1\}$
 $h_2(v) = \{0, 1, 2, 3\}$
 ...

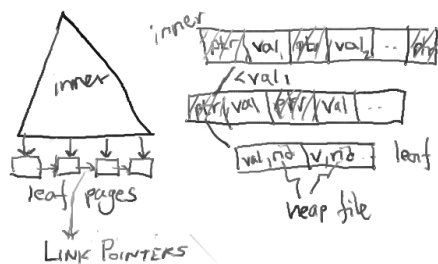
maintain a current hash function, its "levels", and a directory that tells you where each bucket is on disk

example:



B+-Trees:

Show example tree:



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?

How does it do that?

by storing bounding boxes.

What kinds of queries do we want to support?

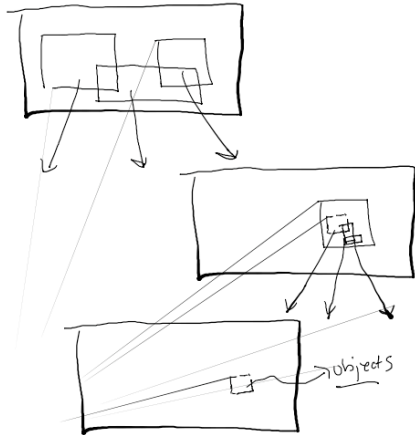
containment, overlap, equality, etc.

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.)