6.830 Lecture 6
3/9/2021

PS2 out today. Lab 2 out today. Lab 1 due today - how was it?

Project Teams Due Wednesday
Those of you who don't have groups -- send us email, or hand in a sheet with just your name on it saying you are looking for partners. If you are planning to do the programming competition, please say so.

Team Meetings w/ Staff in Next Few Weeks

Show slide -- where are we

Last time:

cost estimation basics

This time:

access methods

Last time:
<u>What's the "cost" of a particular plan?</u>

CPU cost (# of instructions)	- 1 ghz == 1 billions instrs / sec, 1 nsec / instr
Spinning Disk	
I/O cost (# of pages read, # of seeks)	- 100 MB / sec = 10 nsec / byte
Random I/O disk = page read + seek	- 10 msec / io operation = 100 io ops / sec
Flash disk	
	- 300-500 MB/sec seq rd/write
	- 10K + random iops / sec (100 usec / io operation) 4K pages ~= 40 MB/sec

Random I/O can be a real killer (10 million instrs/seek) . <u>When does a disk need to seek?</u>

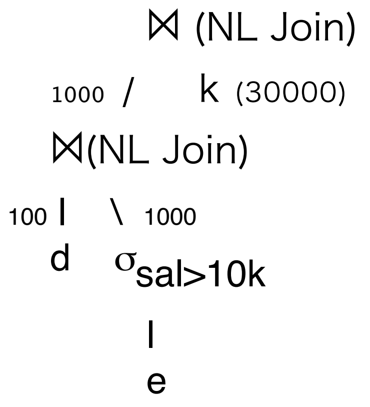
Example (slides):

100 tuples/page
10 pages RAM
10 KB/page

10 ms seek time
100 MB/sec I/O

```
select * from
emp, dept., kids
where e.sal > 10k
emp.dno = dept.dno
e.eid = kids.eid
```

|dept| = 100 records = 1 page = 10 KB
|emp| = 10K = 100 pages = 1 MB |kids| = 30K = 300 pages = 3 MB



1st Nested loops join -- 100,000 predicate ops;   2nd nested loops join -- 30,000,000 predicate ops

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

Join 1

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
NL # predicates evaluated doesn't depend on inner vs outer, but I/O cost does (due to caching effect!)

<u>Join 2</u>

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

<u>So what will be cached?</u>

That's the job of the <u>buffer pool</u>.

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

Convenient "bottleneck" through which references to underlying pages go. When page is in buffer pool, don't need to read from disk. Updates can also be cached.

<u>**** STUDY BREAK ****</u>

Access methods:

Because data is large, we may want to use "access methods" that allow us to get the data we want with a minimum of I/Os.   Especially true if just trying to look up one record (e.g., a particular employee) in a large database.

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.

Already saw heap files and indexes at a high level - will study the details now.

<u>Why not create indices on every attribute?</u>

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

Lets 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

(Slides on sequential vs random and cost models.)

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(logb N)
Scan	P	-	O(logb N) + R
Delete	P/2	O(1)	O(logb N)

Note that simply quantifying the amount of data accessed doesn't capture sequential vs random.   (Slides)

<u>heap files</u>

search cost ~= scan cost ~= delete cost

<u>hash files</u>

search cost ~= 1
insert cost ~= 1
delete cost ~= 1
range scan is equal to sequential scan

map(key) -> {rid} -- could also store map(key) -> {record}

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

h(name) -> {1..k}

h(x) = x mod k; //not very uniform but works
(consider performance when number of distinct values is small, or all numbers even)

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

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

(Show slide)

How many buckets do we need? (hard to tell!)

Too many? -- Wasteful
Too few? -- Long chains that we have to follow

Extensible hashing:

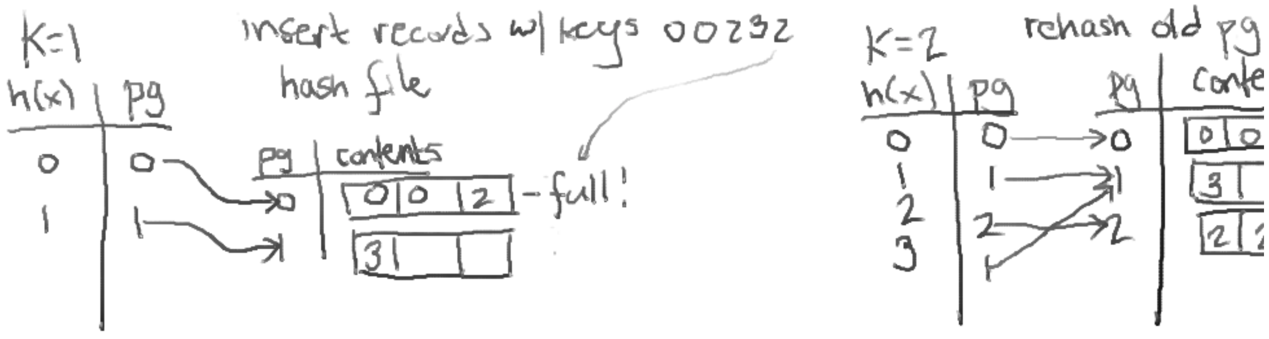
Gave example before -- in principle, we can overflow a page, in which case we need an overflow chain. These overflow chains can get long and slow down the performance of our algorithm.

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

define h(v) = {0, 1, ... , b}
h_k(v) = h(v) mod 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:

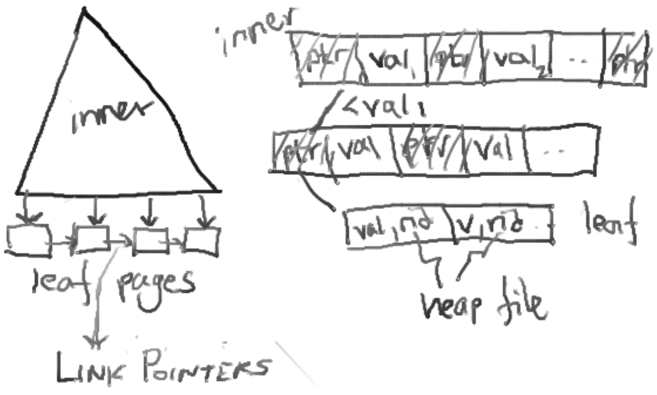


(Show slide)

(Lecture ends here -- B+Trees next time)

<u>B+-Trees:</u>

Show example tree:

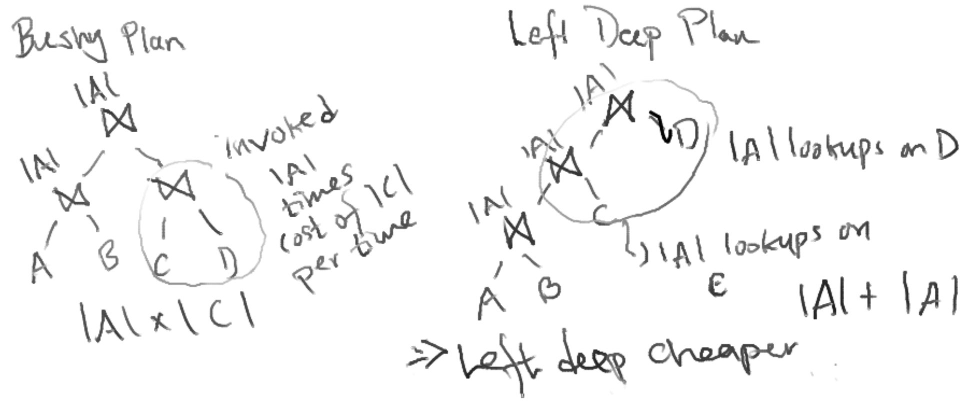


Not going to discuss details of how insertion/splitting/rebalancing work.

<u>Discussion points:</u>

$A \underset{a=b}{\bowtie} B \underset{b=c}{\bowtie} C \underset{c=d}{\bowtie} D$

B+Tree on C.c
B+Tree on D.d



- 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

- log(n) for lookup/insert/delete; how many I/Os in practice?
  (1, maybe 2, since top levels will fit in RAM,
  E.g., top 2 levels are just 513 * 4096 bytes == 2MB)
- Scan is random, unless index is clustered

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

  <u>Why would we not want this to be 100%?</u>
  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

Lots of other indexes, e.g., R+Trees.