

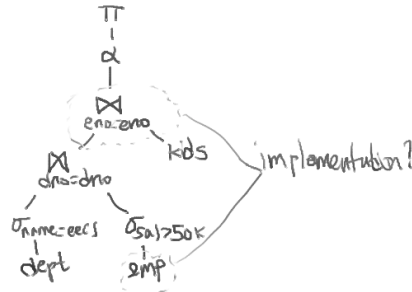


and dept.name = 'eecs'  
 group by ename  
 having count(\*) > 7

What is the equivalent relational algebra?

$\pi(\alpha_{\text{count}(*), \text{ename}, \text{count}(*)>7} ( \text{kids} \bowtie_{\text{eno}=\text{eno}} ( \sigma_{\text{sal} > 50k} \text{emp} \bowtie_{\text{dno}=\text{dno}} ( \sigma_{\text{name}=\text{eecs}} \text{dept} ) ) ) )$

What is the equivalent query plan?



Generating the best plan is the job of the optimizer -- 2 steps;

- 1) logical -- ordering of operators
- 2) physical -- operator selection / implementation (joins and access methods)

Several different approaches to building an optimizer:

- 1) heuristic -- a set of rules that are designed to lead to a good plan (e.g., push selections to leaves, perform cross products last, etc.)
- 2) cost-based -- enumerate all possible plans, pick one of lowest cost -- we will discuss how this works in a couple weeks

### **Physical storage:**

In order to understand how these queries are actually executed, we need to develop a model of how data is stored on disk.

**Storage Recap:** what's different between disk (spinning or flash) and memory? Why not just use the same data structures we use in memory (e.g., arrays, hash tables, linked lists, binary trees?) Disk is *block oriented*, which means we move relatively large chunks (say 4KB at a time) into / out of memory. Also, disk, especially spinning disk, has *long random access latencies*, on the order of msec.

All records are stored in a region on disk ("extent" in system R); probably easiest to just think of each table being in a file in the file system.

Tuples are arranged in pages in some order --> "heap file"

Heap file is a unordered collection of records split into fixed size pages

Because file is unordered, and records all belong to the same table, can write new records anywhere (in empty holes, for example). Very different than files in a file system!

Header on each page to indicate where tuples begin  
 Pages chained together (e.g., in a linked list)



In conventional design, fields of each tuple are stored together, e.g.:  
emp(id, name, dept)

header | id | name | dept

Variable length data (, e.g., name complicates things); soln: store length in header

header (h) = (null bits | len of name, ...)

h | id | dept | name ... |

Since every record takes up different amounts of space, also need a header on the page.

Page header stores:

mapping from recordid -> startoffset

free list -> (empty page regions)

*Access path* is a way to access these tuples on disk.

Several alternatives:

**heap scan** -- read all records in heap file by iterating through pages

**index scan** provide an efficient way to find particular tuples.

Why not just store tuples in sorted order? Would have to insert them in place -- moving all records down when we do this, or create a very long linked list (which would work poorly when data is laid out on disk.)

what is an index? what does it do?

insert (key, recordid) --> points from a key to a record on disk

lookup (key) --> {records}

lookup ([lowkey ... highkey]) --> {records}

Simplest design would just store this structure in memory, e.g., as a sorted list or possibly a binary tree. (Which would I prefer, and why?)

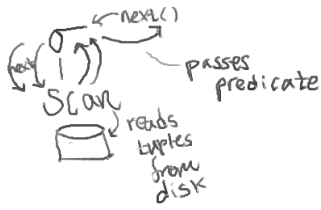
But to be fully general I need these index structures to not be always in memory. (For a 100 TB data set with 100 byte records, I will have 1 T entries in my linked list. With 8 byte pointers and 8 byte keys, this is at least 16 TB for each index.)

Hierarchical indices are the most commonly used-- B-Trees just B-ary trees where B is chose to be the size of a disk block

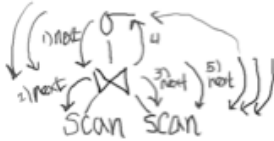
In most databases, indexes point from key values to records in the heap file.

diagram:





example 2:



Iterator code:

(show slides)

Note the “pull from top” processing

Why tuple at a time? What is good about this?

Allows programs and upstream callers to process one record at a time -- no need to hold the entire result of a scan in memory, and allows stopping early when done (e.g., processing a limit query)

What's bad about this?

Lots of function calls (branches) -- these are bad for performance because tends to result in lots of instruction cache misses , also limit data locality.

Alternatives:

Batch at a time (iterators pass arrays of tuples)

Relation at a time (each operator runs to completion)

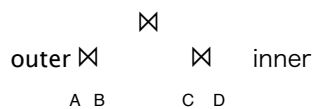
Batches / iterators allow pipelining — earlier result outputs, and each step in the plan can be running in parallel.

**Plan types:**

Left deep vs. bushy

(discuss pipelining)

pipelining -- means that results of one operator can be fed into another operator



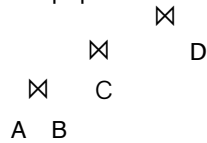
```

for t1 in outer
  for t2 in inner
    if p(t1,t2)
      emit join(t1,t2)

```

problem -- have to either store the result of  $C \bowtie D$ , or continually recompute it

"left deep" plan



No materialization necessary. Many database systems restrict themselves to left or right deep plans for this reason.

(Lecture will end here)

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?

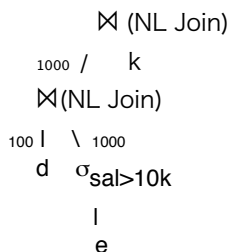
(Depends. Not always disk I/O. Typically vendors try to configure machines so they are 'balanced'. Can vary from query to query. )

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
	emp.dno = dept.dno
10 ms seek time	e.eid = kids.eid
100 MB/sec I/O	

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/costas assuming LRU and no indices**

if d is outer:

1 scan of d  
100 sequential 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  
 $10 \text{ ms} + 1 \text{ MB} / 100 \text{ MB/sec} = 20 \text{ msec}$   
 $20 \text{ ms} \times 100 \text{ depts} = 2 \text{ sec}$

10 msec seek to start of d and read into memory

2.01 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  
 $3 / 100 = 30 \text{ msec} + 10 \text{ msec seek} = 40 \times 1000 = 40 \text{ sec}$

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.

Access methods -- main subject of next time. Want to quickly review the most basic access method: heap files:

heap files

search cost  $\approx$  scan cost  $\approx$  delete cost

- linked list
- directory
- array of objs

file organization

pages  
records  
rids

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: