Project partners due next Wednesday.

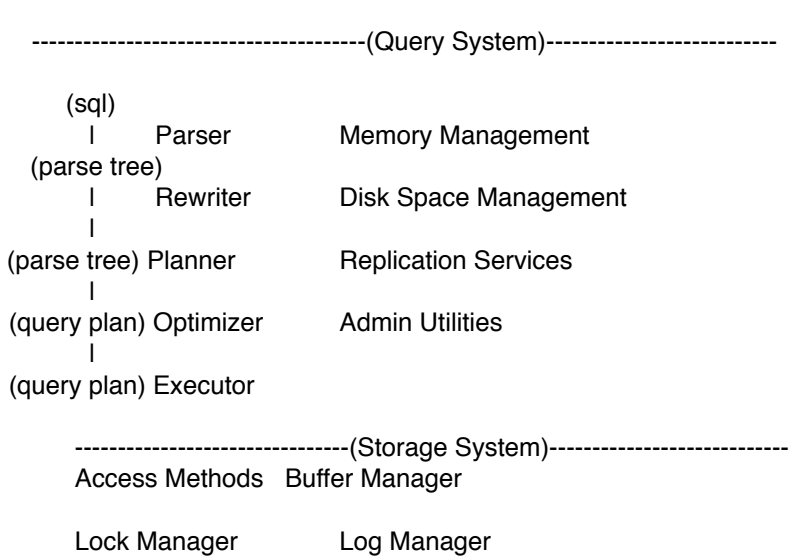
Lab 1 due next Monday — start now!!!

Recap Anatomy of a database system

Major Components:

Admission Control

Connection Management



(Show query rewrite example)

Rewrites are complicated — see “Query rewrite rules in IBM DB2 Universal Database”

step 2 : plan formation (SQL -> relational algebra)
notation

$\pi_{f1..\ fn}\ A$

$\sigma_{p}\ A$

$A \bowtie_{p} B$

$\alpha_{f1..fn,\ g1..\ gn,\ p}$

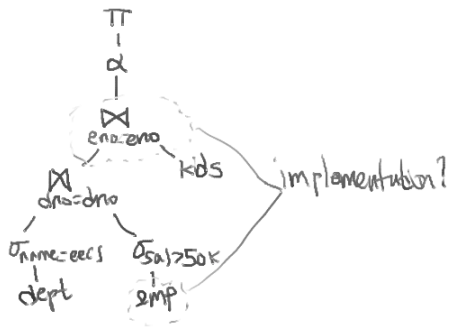
emp (eno, ename, sal, *dno*)
dept (dno, dname, bldg)
kids (kno, *eno*, kname, bday)

```
select ename, count(*)
from emp, dept, kids
and emp.dno=dept.dno
and kids.eno=emp.eno
and emp.sal > 50000
and dept.name = 'eecs'
group by ename
having count(*) > 7
```

What is the equivalent relational algebra?

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

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"

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

Several alternatives:

heap scan

heap file is a unordered collection of records split into fixed size pages
header on each page to indicate where tuples begin
pages chained together (e.g., in a linked list)

index scan provide an efficient way to find particular tuples.

what is an index? what does it do?

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

{records} = lookup (key)

{records} = lookup ([lowkey ... highkey])

Hierarchical indices are the most commonly used-- e.g., B-Trees

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

diagram:

Tree stores salaries in order; leaves point to records with those salaries

Typically, in a database, indexes are keyed on a particular attribute (e.g., employee salary), which allows efficient lookup on that attribute.

What does it mean to "cluster" an index? (arrange keys on disk so that they are in order of index)

Why is that good?

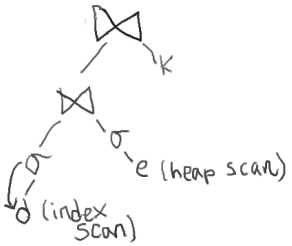
Typically, an access path also supports a "scan" operation, that allows access to all tuples in the table.

Because a given lookup or scan can return lots of tuples, most database indices use an "iterator" abstraction:

it = am.open(predicate)
loop:

tup = it.get_next()

We can place different access methods at the leaves of query plans:



- Heap scan looks at all tuples, but in sequential order
- Index scan traverses leaves of index, so may access tuples in random order if index is not clustered

*** study break -- postgres queries ***

Step 3 : query execution

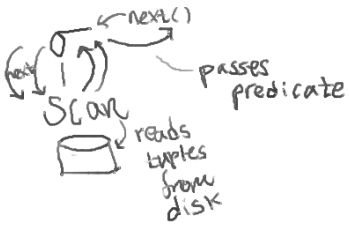
Database query plans -- iterator model

void open ();
Tuple next ();
void close ();

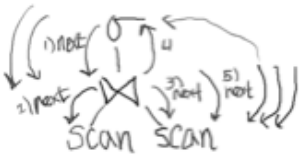
every operator implements this interfaces

makes it possible to compose operators arbitrarily

example 1:



example 2:



Iterator code:

(show slides)

Note the “pull from top” processing

Why tuple at a time? What is good about this?
What’s bad about this?

Alternatives:
- Batch at a time (iterators pass arrays of tuples)
- Relation at a time (each operator runs to completion)

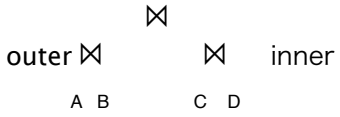
Batches / iterators allow pipelining — earlier result outputs, and each step in the plan can be running in parallel.

Plan types:

Left deep vs. bushy

(discuss pipelining)

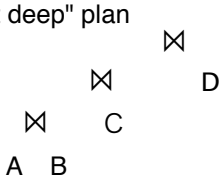
pipelining -- means that results of one operator can be fed into another operator




```
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 ⋈ 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:

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

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
10 pages RAM
10 KB/page

10 ms seek time
100 MB/sec 1/O

|dept| = 100 = 1 page
|emp| = 10K = 100 pages
|kids| = 30K = 300 pages

```
                ⋈ (NL Join)
          1000 /     k
          ⋈ (NL Join)
      100 |    \ 1000
          d    σsal>10k
               |
               e
```

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

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 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

2.01 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 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.

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

heap files

search cost ~= scan cost ~= 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: