

## Lecture 5

9/24/09

**Lab 1 Out -- Due Next Friday**

**Lab 1 Review -- Next Monday, 7 PM 32-G449 (Kiva)**

**PS 2 Out -- Due Oct 21, 2009**

Group assignments due next time--

if you don't have a group, just hand in a piece of paper with your name on it

Will leave time at the end of class to team up

Sample projects will be posted in the next week or so

Will set up short meetings with groups to discuss possible projects

Today:

Database internals

Last time:

Database arch

Runtime system

Parser

Rewriter

Planner

Optimizer

Executor

Storage system

Buffer pool

Xaction stuff

Today: Planner, Optimizer, Executor

Last time -- parsing and rewriting

View rewriting

Subquery flattening

constant elimination, logical predicates, etc.

e.g., WHERE sal > 1000 + 4000

predicate injection based on constraints

a.did = 10 ^ a.did = dept.dno ^ dept.dno = 10

removal of redundant predicates

a.sal > 10k and ~~sal > 20k~~

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

notation

$\pi_{f1..fn} A$

$\sigma_p A$

$A \bowtie_p B$

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

Query: # kids of each employee who makes more than 50 K and has more than 7 kids

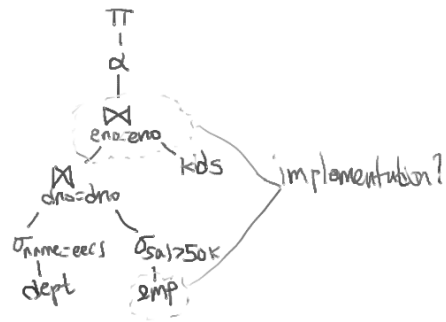
emp (eno, ename, sal, dno)  
dept (dno, dname, bldg)  
kids (kno, eno, kname, bday)

```
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
```

relational algebra:

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

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 (what is described in system R paper) -- 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

physical storage:

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 ("image" in system R)** provide an efficient way to find particular tuples.

**link** -- connection between tuples in two different files (not going to discuss)

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 common type used-- e.g., B-Trees

indices typically point from key values to records in the heap file

diagram:



typically, in a database, indices are keyed on a particular attribute (e.g., employee salary), which allows efficient lookup on that attribute. values are just the tuples themselves.

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

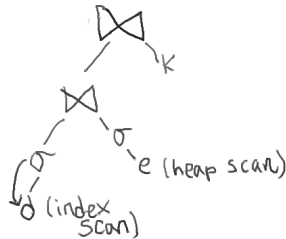
why is that good?

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

b/c 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 look sat all tuples, but in sequential order
- index scan traverses leaves of index, so may access tuples in random order if index is not clustered

### step 3 : query execution

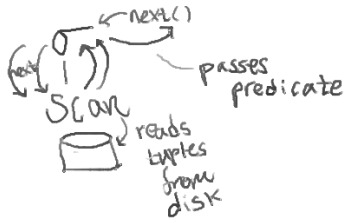
database query plans -- iterator model

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

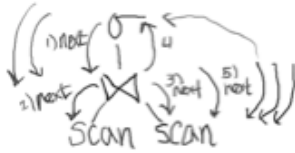
every operator implements this interfaces

makes it possible to compose operators arbitrarily

example 1:



example 2:



Iterator code:

```

class Select extends Iterator {
    Iterator child;
    Predicate pred;

    Select (Iterator child, Predicate pred) {
        this.child = child;
        this.pred = pred;
    }

    Tuple next() {
        Tuple t;
        while ((t = child.next()) != null ) {
            if (pred(t)) {
                return t;
            }
        }
        return null;
    }

    void open() {
        child.open();
    }

    void close() {
        child.close();
    }
}

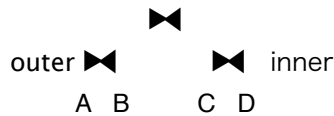
```

**plan types:**

left deep vs. bushy

(discuss pipelining)

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

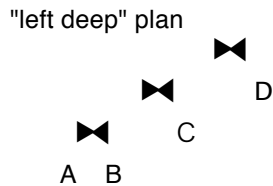


```

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



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

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!

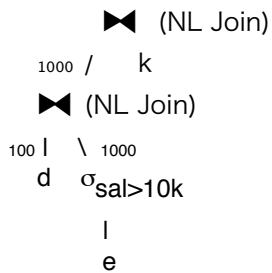
But, if we do a bad job, random I/O can kill us!

- 100 tuples/page
  - 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

```

- ldeptl = 100 = 1 page
- lemp1 = 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.

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:

