6.830 Lecture 4 3/1/2021

(Note that these notes will be used for the next 2-3 lectures)

HW 1 Due Today. How was it? Lab 1 Due Next Monday.

Project partners due next Wednesday (3/10). If you haven't found a partner, see Piazza. If you have no team we will pair you up. Teams of 3 students preferred. Will post suggested projects soon; this year we are planning default project that involves a competition to implement a fast database system in C. More details next time.

Recap FDs.

FD1: SSN -> Name, Address
FD2: Hobby -> Cost
FD3: SSN, Hobby, --> Name, Address, Cost (derivable from previous 2)

Normal Forms

A table that has no redundancy is said to be in BCNF "Boyce Codd Normal Form"

Formally, a set of relations is in BCNF if:

> For every functional dependency X->Y in a set of functional dependencies F over relation R
> X is a *superkey key* of R,
> (where superkey means that X contains a key of R)

go to our hobbies example
if we use the original example,

SSN -> Name, Address is not a superkey! So this is not in BCNF.

--> Redundancy
In non-decomposed hobbies schema Name, Addr repeated for each appearance of a given SSN

BCNF implies there is no redundant information -- e.g., that the association implied by any functional dependency is stored only once;

Observe that our schema after ER modeling is in BCNF (FDs for each table only have superkeys on the left side)

Decomposing into FD is easy -- just look at each FD, one by one, and check the conditions over each relation. If they don't apply to some relation R, split R into two relations, R1 and R2, where R1 = (X U Y) and R2 = R - (X U Y),

Start with one "universal relation"

While some relation R is not in BCNF
    Find an FD F=X–>Y that violates BCNF on R
        Split R into R1 = (X U Y), R2 = R – Y

Example:

FD2: SSN -> Name, Address
FD3: Hobby -> Cost

R = S,N,A,H,C
R is not in BCNF, b/c of FD2 (N, A is not a primary key of R)

R1 = S,N,A, FD2 ; R2 = S,H,C, FD1', FD3

R2 not in BCNF, b/c of FD3

R3 = H, C (FD3)
R4 = S, H (FD1'')

Is it always possible to remove all redundancy? No!

(see slides)

Denormalization

When and where do we want to do it?

Do we always want to decompose a relation? Why or why not?

Generally speaking, decomposition :

- decreases storage overhead by eliminating redundancy and
- increases query costs by adding joins.

This isn't always true! Sometimes it increases storage overhead or decreases query costs.

Sometimes (for performance issues) you don't want to decompose.

So how much does this really matter?

Eliminating redundancy really is important.
Adding lots of joins can really screw performance.

These two are sometimes at odds with each other.

In practice, what people do is what we did for hobbies -- think about entities, join them on keys. "Entity relationship" model provides a way to do this and will result in something in BCNF.

Today: Relational Database Systems

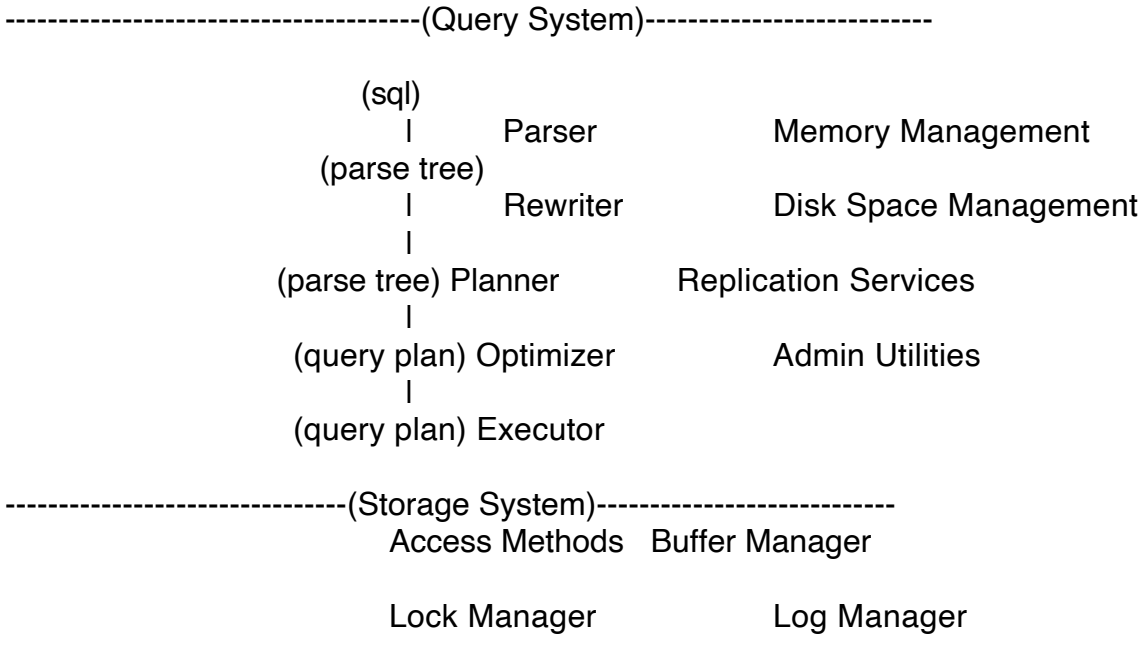
probably doesn't all make sense right now -- you should look at both of these papers through the semester for context

Anatomy of a database system

Major Components:

Admission Control

Connection Management



What is flow of a query?
Going to go through stages one by one over next lecture and a half.

We will skip multiprocessor stuff for now -- we will revisit when we talk about parallel and distributed DBs.

Process models --
	why does this matter? (what happens if you get it wrong)

	process per connection (why is this bad? why is it good? how does memory management work?)

	process per server, w/ worker threads
		additional processes for asynchronous I/O

	how many threads/processes should i have?
		- dispatch thread
		- worker threads
		- I/O for asynchrony

Query processing

step 0: admission control / authorization -- role based access control

step 1 : query rewrite (logical optimization)

view rewrite example

```
create view sals as (
	select dept, avg(sal) as sal
	from emp
	group by dept
)
```

emp : id, sal, age, dept

```
select sal from sals where dept = 'eecs';

select sal from
(
select dept, avg(sal) as sal
from emp
group by dept
)
where dept = 'eecs';
```

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~~

subquery flattening

```
select avg(sal) as sal
from emp
where dept='eecs'
group by dept
```

a little tricky; suppose view was:

```
create view sals as {
        select distinct dept, sal
        from emp
}
```

and query was

```
select avg(sal) from sals
```

This is equivalent to:

```
select avg(sal) from (
    select distinct dept, sal
    from emp
)
```

<u>but can we flatten more?</u>
e.g., to

```
    select avg(distinct sal) from emp
```

no! why? duplicates:

eid, dept, sal
1, eecs, 100K
2, eecs, 100K
3, me, 50K
4, arch, 50K

average = 75 K

sals:
eecs,100K
me,50K
arch, 50K

average = 66 K

Break / study question (end of class, unless extra time)

"rule based optimizer" -- situations in which subqueries can be merged!
(see 'query rewrite rules in IBM DB2 Universal Database')

Lecture 5 —
step 2 : plan formation (SQL -> relational algebra)
notation

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

$\sigma_{p}\ A$

$A \bowtie_{p} B$

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

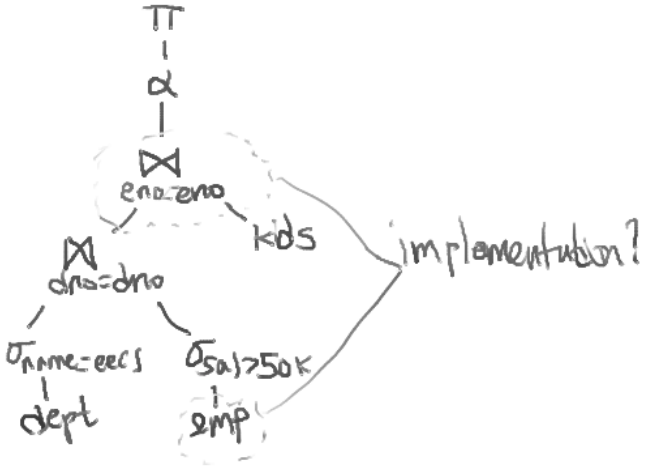
emp (<u>eno</u>, ename, sal, *dno*)
dept (<u>dno</u>, dname, bldg)
kids (<u>kno</u>, *eno*, kname, bday)

```
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?

π ($\alpha_{count(*), ename, count(*)>7}$ (
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 ("image" in system R) provide an efficient way to find particular tuples.

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

<u>what is an index? what does it do?</u>

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.

(Skip clustering)

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

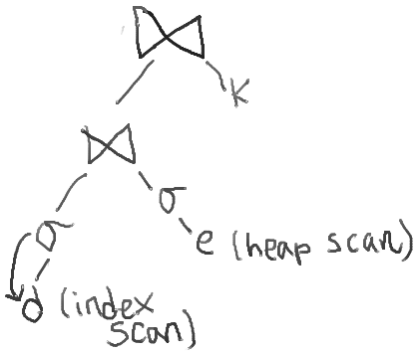
Why is that good?

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

Step 3 : query execution

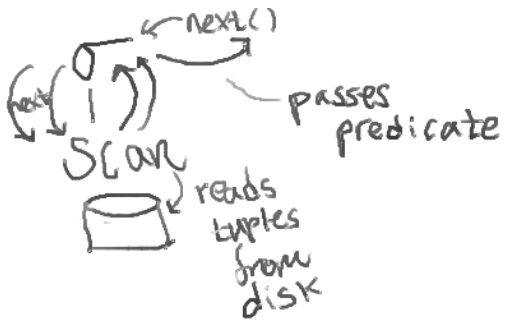
Database query plans -- iterator model

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

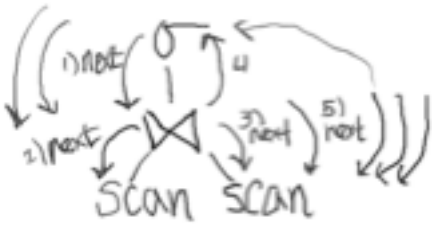
every operator implements this interfaces

makes it possible to compose operators arbitrarily

example 1:



example 2:



Iterator code:

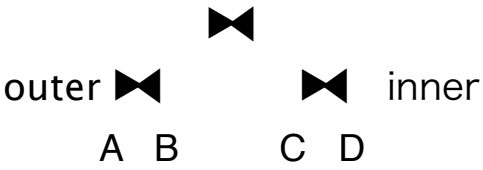
```
class Select extends Iterator {
        Iterator child;
        Predicate pred;

        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)

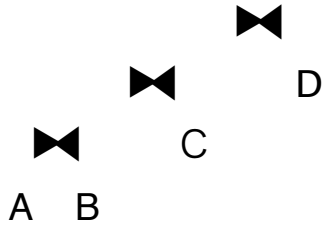
pipelining -- means that results of one operator can be fed into another operator




```
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 ⋈ 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.

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!

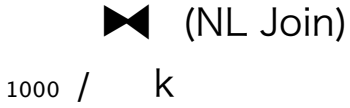
But, if we do a bad job, random I/O can kill us!

100 tuples/page
10 pages RAM
10 KB/page

10 ms seek time
100 MB/sec 1/O

|deptl = 100 = 1 page
|empl = 10K = 100 pages
|kidsl = 30K = 300 pages

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



⋈ (NL Join)

100 / \ 1000

d $\sigma_{sal>10k}$

|

e

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:

<u>tuple layout</u>

- fixed length
- variable length
  - field slots
  - delimiters
- half fixed/variable
- null values?

example: