Lecture 3

9/17/09

HW 1 Due Tuesday. Questions?

Last time:

Zoo schema:

keepers (kid, kname, kaddr)
animals (aid, aname, a_kid, a_cid)
cages (cid, bldg., size)

Data models.
Saw major problems with early models --

Lack of expressiveness (hierarchies)
Lack of
physical data independence (insensitivity to representational changes) or
logical data independence (insensitivity to logical changes)

Saw that relational model provides expressiveness of network model, arguably simpler

Let's see how it provides data independence

Recap major operations:

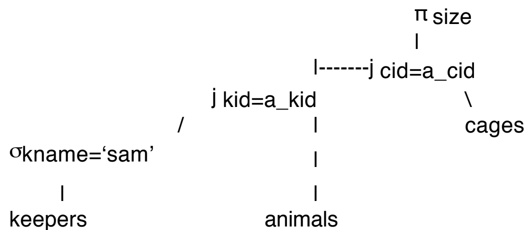
Project	π c1 … cn -- extra subset of columns
Select	s_p(A) -- extra rows that pass predicate p
Join	A j_p B -- s_p (A x B)
	if we just write A j B --> A j B on key-foreign key relationship; "natural join"

Find the size of the cages sam keeps
SQL:

```
SELECT size FROM cages, keepers, animals
WHERE kname = 'sam' AND a_kid = kid and a_cid = cid
```

Many possible relational expressions

π_{size} ((($\sigma_{kname='sam'}$ keepers) j $_{kid=a_kid}$ animals) j $_{cid=a_cid}$ cages)



π_{size} ((($\sigma_{kname='sam'}$ keepers) j $_{kid=a_kid}$ (animals j $_{cid=a_cid}$ cages))

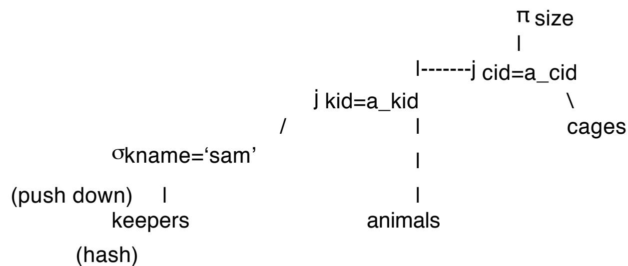
In general, all operators commute with all other operators:
A j B = B j A
(A j B) j C = A j (B j C)
Sp (A j B) = Sp(A) j b
πa (A j B) = (πa A) j b

Observe that SQL is equivalent to any of these relational expressions (SQL is more "declarative" since it doesn't imply an

order of operations.)

Note that SQL and relational algebra are describe in terms of implementation over tables, but in fact the physical representation may not be tables. Programmer is not aware of physical implementation. This is "physical data independence."

Example, in reality, keepers may be stored as hash table on name:



In general, these different physical representations ("access methods") must not constrain the operations that can be performed over them.

<u>So how do we provide logical data independence?</u>

Add a layer of indirection!

User writes queries over views, that don't necessarily have the same schema as the physical representation
Database system translates queries over logical views into queries over physical structures

Example in SQL:

change tables so that each keeper keeps one cage

```
ALTER TABLE cages ADD COLUMN c_kid int REFERENCES keepers.kid
SELECT sid,name,a_cid INTO animals2 FROM animals
DROP TABLE animals
CREATE VIEW animals as (SELECT sid, name, a_cid, c_kid FROM animals, cages WHERE a_cid = cid)
```

"Animals" view has the same logical schema as original animals table, so old programs (hopefully) still work; new programs can use new schema.

Will see how view rewriting works next time, but in general always possible to substitute in a view definition into a user's SQL query.

<u>Don’t views hurt performance</u>?

Yes. (Hence, materialized views.)

Don’t those hurt performance too (yes, update performance)

<u>Data independence == layer of indirection</u>

User interacts with a logical representation that is quite different from underlying physical storage.

Generally bad for performance, good for maintainability

(This is what Codd valued above all else.)

Have now seen basic relational algebra. Will soon talk about how we actually implement a system that executes this algebra. But first, let's focus on how we design a database and choose a set of tables. (My least favorite part of database systems.)

Schema Normalization

Q1 : <u>Why is redundancy a problem</u>?

- Because it leads to various anomalies when you try to update a table.

- Because it wastes space

ss#	name	address	hobby	cost
123	john	main st	dolls	$
123	john	main st	bugs	$
234	mary	lake st	bugs	$
345	mary	lake st	tennis	$$
456	joe	first st	dolls	$

What is the primary key? SS# + Hobby?

Types of anomalies (Codd calls "inconsistencies")

- **Update anomaly** - change one address -- need to change the other

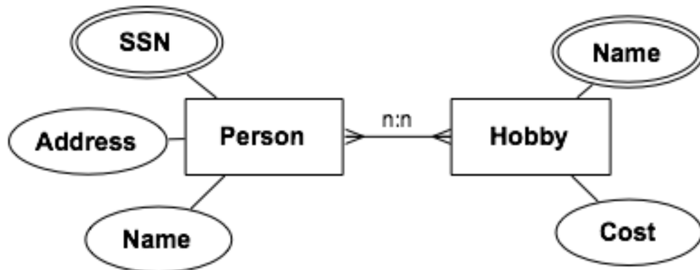
- **Insertion anomaly** - what if we want to add someone with no hobby?
  have to use a null?

Why is this a problem?
  what if the person later gets a hobby?

- **Deletion anomaly** - what if we want to delete someone's hobby?

Q2: What can we do to solve it? Normalize!

Way you normally do this is with ER diagrams:



n:n relationships imply a mapping table from key of left table to key of right table

so we'd have
  person (ssn, address, name)
  hobby (name, cost)
  personhobby (hobbyname, ssn)

1:n relationships imply a key-foreign key reference,
  e.g., if every person has one hobby, we could add a hobbyid field
1:1 relationships can be merged into the same table (imply a strict dependencies) -- e.g., each person
  has a unique hobby, then their hobby can just be stored with them

(Decomposes into

ss#	name	address
1	joe	main st
2	jenny	lake st
3	jimmy	south st

hobby	cost
dolls	$
bugs	$
tennis	$$

ss#	hid
1	dolls

1 bugs
2 tennis
2 bugs
3 dolls)

still have some issues (e.g., might need null addresses)

but we've eliminated the anomalies (only need to change one address, don't
need nulls for hobbies, etc.)

What about:

ss#
name
address
cost

hobby
ss#

No redundancy, but we have lost association between hobbies and costs. "lossy decomposition"

Let's formalize this idea a bit more to see why ER modeling leads to a good decomposition.

So how do we decide how to normalize?
We use **functional dependencies**: a type of *integrity constraint (IC)*

Functional Dependencies.

Formally a functional dependency F is a constraint on a schema R

X->Y means
"X implies Y"
where X and why are subsets of the attributes in R

A relation instance r satisfies F if for every pair of tuples a and b in r,
if a and b agree on all attributes in X, they agree on all attributes in Y.

Think about this as a function: every input maps to exactly one output.

Our example:

FD1: SSN, Hobby -> SSN, Name, Address, Hobby, Cost
FD2: SSN -> Name, Address
FD3: Hobby -> Cost

Where do FDs come from?

Domain knowledge of database designer (not derived from data, though can check that
data satisfies them!)

Types of normalization

- 1NF : no set-valued (non-atomic) attributes -- example with hobbies
- 2NF : skip, described in reading

BCNF :

What is point of BCNF?

Want to partition tables such that:

For every functional dependency X->Y in a set of functional dependencies F
over relation R:

either:

Y is a subset of X

or

X is a *superkey* of R

superkey means that X contains a key of R
key is a set of attributes that uniquely determine the other attributes

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

SSN -> Name, Address is neither trivial nor a superkey of the wide table! So this is not in BCNF.

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

because SSN is not a super key, SSN may appear multiple times, and this FD may be repeated redundantly

Note that you can't tell what the FDs are just by looking at a table:

A	B	C	D
1	2	3	4
2	2	3	4

If FD is A->BCD no "redundancy" here

If FDs ar

B->CD and
A->B

Redundancy!

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

Note that FD 1 (SSN, Hobby -> SSN, Hobby, Name, Addr, Cost) hasn't been lost b/c it is implied by other two:

SSN -> Name, Addr
Hobby -> Cost

Clearly

SSN, Hobby -> SSN, Hobby, Name Addr
SSN, Hobby -> SSN, Hobby, Cost

- <u>3NF</u> :

Also allows FDs X-->Y such that

Y is a subset of K for some key K of R (not that Y is a part of a superkey!)

Example:

Account	Client	Office
a	joe	1
b	mary	1
a	john	1
c	joe	2

Client, Office ->, Account
Account -> Office

has an insertion anomaly -- change the office that an account
belongs to, and we have to change a number of records

this is in 3NF, but not BCNF

because office is a subset of the key {Client, Office}

Any relation in BCNF is also in 3NF (but not vice versa).

Decomposition:

Our goal is to decompose a given schema into a collection of schemas
that satisfy BCNF (usually).

Want a **lossless** decomposition, which means that:

for all instances r of R that satisfy FDs, a lossless decomposition of r
into x and y will have the property that

x join y = r (e.g., don't put names and address in different tables -- couldn't reconnect them)

Also might want decompositions to be **dependency preserving.**

This is true only if the closure of the union of the dependencies of all the decomposed relations
is equivalent to the original set of dependencies.

Important so that we can check semantics of data (e.g., all hobbies have the same cost)

Is decomposition of hobbies dependency preserving? (yes)

BCNF decomposition of account, client, office:

Account Office (Account -> Office)
Account Client ( )

Is this dependency preserving?

Now, it's BCNF, but we lost our primary key constraint that client and office uniquely
determine C, O, A.

This is problematic.

In general, always possible to find a 3NF decomposition that is dependency preserving, but as we saw, it may still have some redundancy. So there's a tradeoff between which normal form is better.

ER modeling is a simple way to perform this decomposition. Instead of starting with FDs, in ER model you start with entities, but observe that entities are generally the things on the left side of your FDs (e.g., hobbies, or SSNs.) Things on right side are properties of the FDs. Generally ER modeling results in BCNF decomposition.

BCNF decomposition algortihm

Idea: given a schema not in BCNF, split it into two schemas, one of which is in BCNF. Repeat until all schemas are in BCNF.

```
Input R -- relation,F -- set of FDs over R
D = {(R,F)}
while there is a (schema,FD set) pair  (S,F') in D that is not in BCNF, do
        given X->Y as an FD in F' that violates BCNF, do
                replace (S,F') in D with S1=(XY,F1) and S2= (S-Y) U X,F2)
                where F1 and F2 are the FDs in F over S1 or S2, respectively

return D
```

Example:

SSN, Name, Hobby, Cost, Addr

1) SSN->Name, Addr
2) Hobby->Cost
3) SSN,Hobby -> Name, Addr, Cost

D = {(SNHCA, {S->NA, H->C, SH -> NAC})}

Step 1:

S->NA violates BCNF
D = {(SNA, {S->NA}),
(SHC, {H->C, SH -> NAC})}

Step 2:

H->C violates BCNF
D = {(SNA, {S->NA}),
(HC, {H->C}),
(SH, {})}

SSN, Name, Addr
Hobby, Cost
SSN, Hobby

Denormalization. <u>When and where do we want to do it?</u>

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.

<u>So how much does this really matter?</u>

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 (see Stonebraker paper, or book). Will result in something in BCNF.

<u>Properties of FDs:</u>

"Armstrong's Axioms"

Reflexitivity: if Y is a subset of X, then X->Y : "trivial FD" -- any relation satisfies these!

Name, Hobby -> Name (obviously)

Augmentation: If we have a relation defined as XYZ, If X->Y, then XZ -> YZ

SS# -> Name
SS#, Addr -> Name (obviously)

Transitivity: X->Y, Y->Z means X->Z

animal		cage	keeper
monkey	1	joe	
donkey		2	bill
rabbit	1	joe	

animal -> cage

cage -> handler

animal -> handler

Union of FDs:

X->Y and X->Z satisfies X->YZ

X->YX (augmentation)
YX -> YZ (augmentation)
X->YZ (transitivity)

Decomposition of FDs:

given X->YZ
satisfies
X->Y and X->Z

YZ-> Y (reflexivity)
X->Y (transitivity)

Closure:

Want to find all attributes A such that X->A is true, given a set of functional dependencies F.

define closure of X as X*
algorithm:

```
    closure = X
    repeat
        old = closure
        if there is an FD Z->V such that Z is contained in closure and
            V is not contained in closure then
        closure = closure U V
    until old = closure
    return closure
```

if X* -> A s.t. A = R, then X is a key of R

example: R = ABCDEF
AB->C
D->E
BC->DF

AB* = AB, ABC, ABCDF, ABCDEF (AB is a key!)

Get a complete set of FDs.