6.830 Lecture 3 **2.24.21**
PS1 Due Next Time (Tuesday!)
Lab 1 Out Today — start early!

Relational Model Continued, and Schema Design and Normalization

Animals(name,age,species,cageno,keptby,feedtime)
Keeper(id,name)

Main operations (from last time)

> Projection (π(T,c1, …, cn)) -- select a subset of columns c1 .. cn
> Selection (sel(T, pred)) -- select a subset of rows that satisfy pred
> Cross Product (T1 x T2) -- combine two tables
> Join (T1, T2, pred) = sel(T1 x T2, pred)

Plus various set operations (UNION, DIFFERENCE, etc)

(show slide)

Notice that basic ops are all set oriented, unlike record oriented in previous models

Ex: Keepers kept by joe: $\pi_{cageno}(\sigma_{name='joe'}(animals \bowtie_{keptby=kid} keepers))$

Though the relational algebra is very elegant, it was hard to understand. Part of the great debate was that Bachman et al didn't think mere mortals could program in it, and didn't believe it could be implemented efficiently.

SQL is an attempt to create something more reasonable

Instead of math expressions, set-oriented language; above query becomes

```
SELECT cageno                  SELECT cageno
FROM keepers, animals          FROM keepers JOIN animals
WHERE keeper = keptby          ON keeper = keptby
AND keeper.name = 'joe'        WHERE keeper.name = 'joe'
[ORDER BY]
[GROUP BY]
```

Translating SQL <----> Relational Algebra is straightforward

SELECT ==> Projection list
FROM ==> all tables referenced
JOIN ==> join expressions
WHERE ==> select expressions
(show slide)
No expectation that SQL query processors will evaluate expressions in order they are written.

Aggregates not in relational model.

Note that SQL is a "calculus", meaning that it specifies *what*, not *how* -- i.e., it is "declarative", whereas the relational algebra is more imperative because a relational expression implies a particular ordering of operations.

<u>Recap: SQL vs CODASYL</u>

- **Programming no longer navigates in N-D space** -- just write a program to extract the data they want. Simple, and elegant.

- Physical independence:

Note that the relational model says nothing about the physical representation at all.   Users just interact with operators on tables. Tables could be sorted, hashed, indexed, whatever

Programs will continue to run no matter what physical representation is chosen for the data.

What is supposed to happen is that:

- User isn't supposed to know about representation
- Administrator lays out data on disk in a way that is good for user's queries.
- Query optimizer picks best "access method" for query at hand

In practice, as a user of a database, you often need to work to get the best representation to make your queries run fast. But typically this doesn't involve schema changes.

- Logical independence: (How can you change the schema without breaking legacy programs?)

Some forms of logical independence are easy, e.g., adding a column or a table won't break anything in SQL (unlike in IMS!)

Other forms are harder, for example, if we want to refactor a schema.

Create a "view" that looks like the old schema. E.g., suppose we want to add multiple feedtimes:

Create a new table, feedtimes (aname string, feedtime time)

Rename the animals table to animals2

Remove the feedtime column from animals

Create a "view" with the name animals, defined in terms of animals2 and feedtimes:

```
CREATE VIEW animals AS
SELECT name, age, species, cageno, keptby,
  (SELECT feedtime
  FROM feedtimes
  WHERE aname=name
  LIMIT 1)
FROM animals
```

(show slide / code)

Now when animals2 gets updated, animals will still be valid table according to old schema.

Of course, old apps need to be rewritten if they need to know animals have multiple feed times.

Somewhat imperfect -- e.g., figuring out how to update a2 if animals is updated is hard (this is the "view update problem")

Can also introduce performance impacts, since queries over old schema have to be "rewritten" through views

We will talk more later, but *all* queries and many updates can be supported on views.

In summary -- 1970's there was a long debate on this topic

One one hand, non-relational camp argued for
- Low level efficient programming languages

Relational camp wanted:
- High level languages
- Data independence

Relational camp eventually won -- when IBM released its relational product.

Note that high level languages and data independence are possible in a graph language. Example, JSON support in Postgres (show slides), demo.

Now that we understand relational algebra, we are almost ready to 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.

(break)

Schema Normalization
Goal: Produce a collection of tables (schema) that is redundancy free
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	$

<u>What is the primary key?</u> SS# + Hobby?
people have names and addresses, hobbies have costs
people can have multiple hobbies, and hobbies can be practiced by multiple people

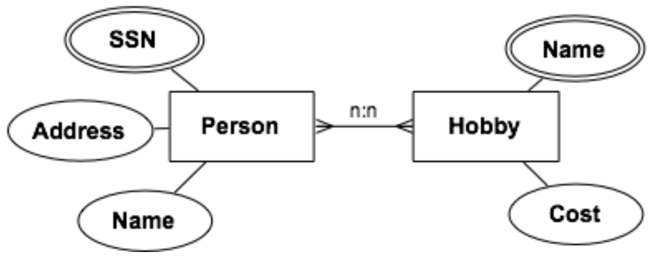
Types of anomalies (Codd calls "inconsistencies")

- **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?
  problem -- hobby is a part of the key!

Q2: <u>What can we do to solve it?</u> Normalize!

Most common way to do this is with an "entity relationship diagram"



Entity
Relationship
Diagram

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

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

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)

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 some information. "**lossy decomposition**"

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

How do we do this systematically?

Let's understand where the redundancy comes from.

Looking at our hobbies example, SS# is sufficient to uniquely determine name and address, but the key of the table includes hobby, which means a given SS# can repeat, and hence so can the names and addresses for the SS#.

One way to think about this is in terms of "functions" -- in the sense that a given a particular input value, a function always produces the same output. So, e.g., a given SS# always produces the same name.

We write these like this:

SS# ---> Name

These kinds of relationships are called functional dependencies. Redundancy arises when the LHS of one the functional dependencies over the attributes in a table is not the key of the table.

For our tables above, we can write down some FDs:

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

FD2 and FD3 sufficient to imply FD1 ==> "Armstrong's axioms"

Because for our "wide table", SSN is not a key, we can see that some info will be repeated each time an SSN is repeated with a hobby.

Where do FDs come from?

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

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")

<u>Is it always possible to remove all redundancy?</u>

Consider:
account, client, office:

Client, Office -> Account
Account -> Office

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

(key client,office)

Redundancy b/c the fact that account a is in office 1 is represented twice.
Not in BCNF b/c account is not a superkey of the table

This table is in "third normal form" (3NF) but not BCNF.

To put it into BCNF, we would have to decompose into

Account, Office
Client

But we've "lost" the first dependency now, in the sense that it can't be checked by looking at a single table.

3NF preserves all dependencies, but may have redundancy, and BCNF removes all redundancy but may drop some dependencies.
Comparable algorithms for 3NF, not going into details.

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 and will result in something in BCNF.