Administrivia

HW3 is out.

Project 1 is due next week

L7
Normalization is a Good Idea

Eugene Wu

Fall 2015

Steps for a New Application

Requirements
- what are you going to build?

Conceptual Database Design
- pen-and-pencil description

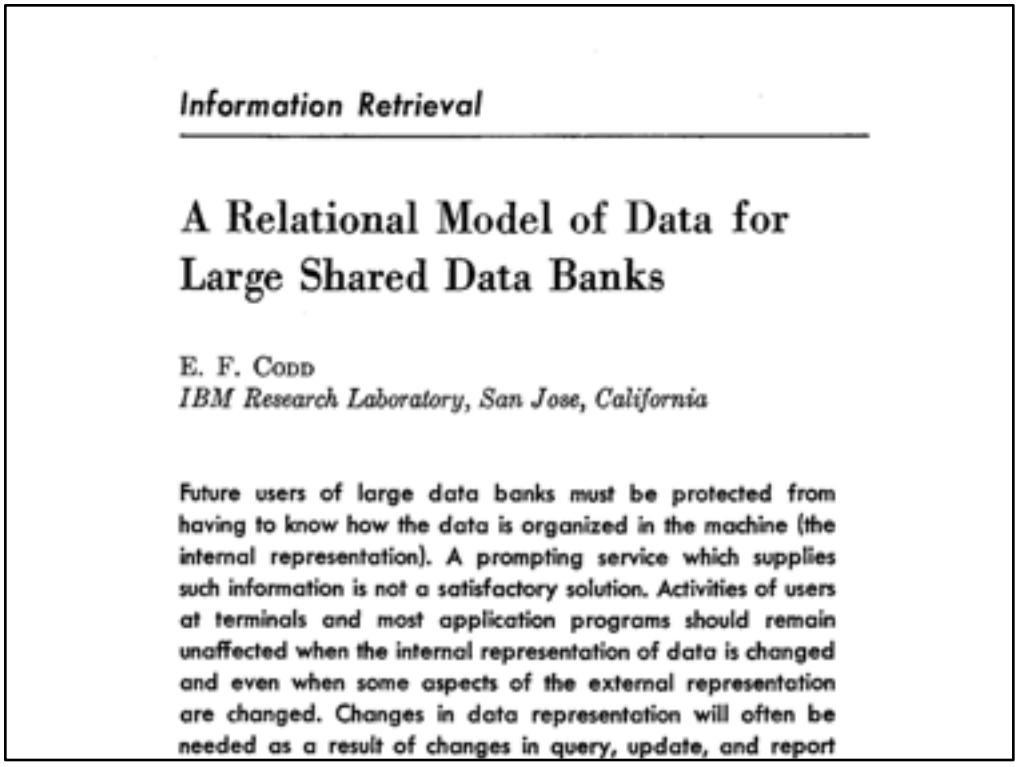
Logical Design
- formal database schema

Schema Refinement:
- fix potential problems, normalization

Normalization

Physical Database Design
- use sample of queries to optimize for speed/storage

App/Security Design
- prevent security problems



Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report

It's amazing that we've spent several months on this topic, and they were all, including this lecture, mentioned and foreseen by this single initial paper defining the relational model.

this is one of those topics where it is actually a good idea.
if you ever find yourself writing a lot of code to "fix things up" whenever data is updated or changed, take a look at the data model and the decomposition.

Redundancy is no good

Update/insert/delete anomalies. Wastes space

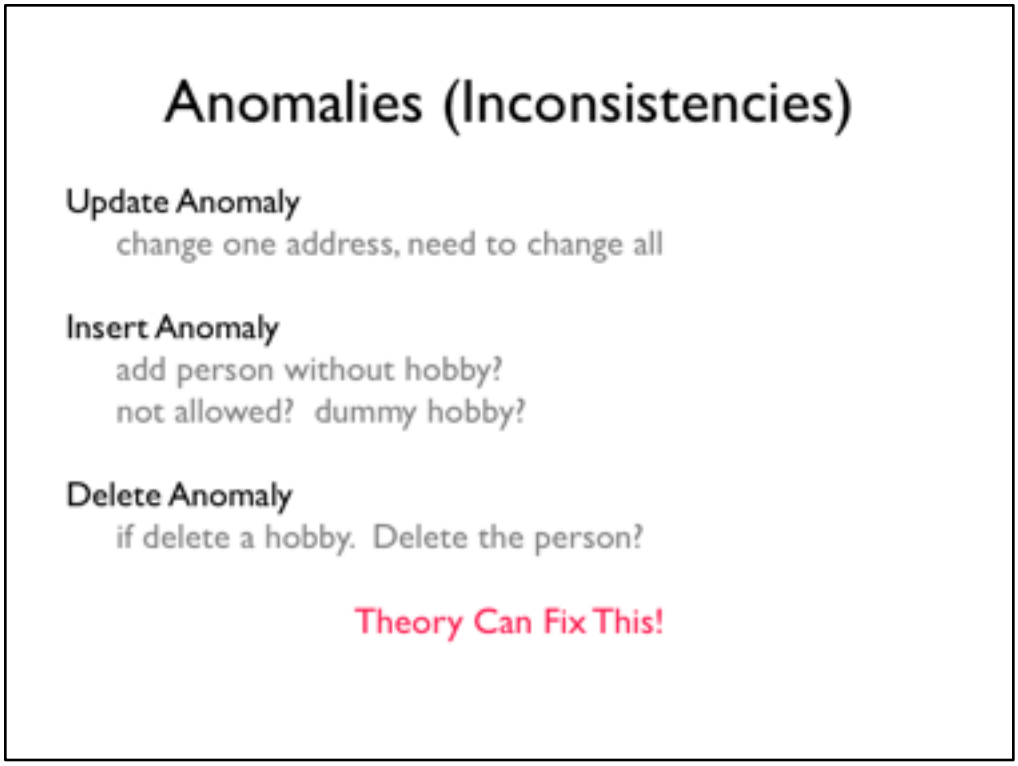
sid	name	address	hobby	cost
1	Eugene	amsterdam	trucks	$$
1	Eugene	amsterdam	cheese	$
2	Bob	40th	paint	$$$
3	Bob	40th	cheese	$
4	Shaq	florida	swimming	$

people have names and addrs
hobbies have costs
people many-to-many with hobbies
What's primary key? sid? sid + hobby?

Premise: redundancy = bad.
Let's see an example.
So far, we wouldn't store the data as a single table, but the process was play by feel (adhoc).
Want systematic way with guarantees.
Lots of students commented that it's not even clear how to guarantee that ER -> relational transformations are going to be correct.

Starting with all our data in a singel table, why is this bad? What are some reasons? examples of update, insert or delete anomalies.

Delete: if delete swimming, what happens to the hobby? if sid+hobby is key, then hobby can't be null, so should we delete the record? but what if that's the last record with shaq? then we lost shaq!

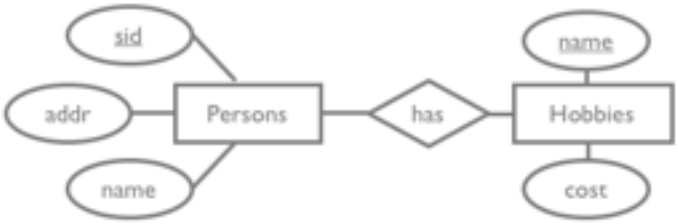


These anomalies are reminiscent of the issues we found with translating ER to relational, however we never formally defined them.
They are anomalies in the sense that on a change in the database (a simple insert/update/delete), additional possibly ambiguous operations are needed to keep the database in a consistent state with respect to our constraints.
Here we classify them:

delet anomaly: setting hobby info to NULL doesn't work, because sid,hobby is a key and hobby can't be null

A Possible Approach

ER diagram was a heuristic



We have decomposed example table into:

```
person(sid, addr, name)
hobby(name, cost)
personhobby(hobbyname, sid)
```

A Possible Approach

What if decompose into:

```
person(sid, name, address, cost)
personhobby(sid, hobbyname)
```

sid	name	address	cost
1	Eugene	amsterdam	$$
1	Eugene	amsterdam	$
2	Bob	40th	$$$
3	Bob	40th	$
4	Shaq	florida	$

sid	hobby
1	trucks
1	cheese
2	paint
3	cheese
4	swimming

but… which cost goes with which hobby?

lost information: *lossy decomposition*

but what makes the previous decomposition a good one? What if we decomposed it into the following?

lossy decomposition means that dependencies we cared about – hobby and its cost are related – are lost during the process of decomposition.

Clarify: anomalies are when the single table is not decomposed. lossy decomposition is an issue of losing information for a decomposition

Decomposition

Replace schema R with 2+ smaller schemas that

1. each contain subset of attrs in R
2. together include all attrs in R

ABCD replaced with AB, BCD or AB, BC, CD

Not free – may introduce problems!

1. **lossy-join:** able to recover R from smaller relations
2. **non-dependency-preserving:** constraints on R hold by only enforcing constraints on smaller schemas
3. **performance:** additional joins, may affect performance

Lossless: we better not lose data, so should be able to join the smaller relations together to recover original relation
otherwise called “lossy”

dependency: each of the functional dependencies that hold on R should hold on at least one of the smaller relations! otherwise we lost constraints!

Can we systematically
decompose our relation to

prevent
decomposition
problems

remove
redundancy?

Functional Dependencies (FD)

sid	name	address	hobby	cost
1	Eugene	amsterdam	trucks	$$
1	Eugene	amsterdam	cheese	$
2	Bob	40th	paint	$$$
3	Bob	40th	cheese	$
4	Shaq	florida	swimming	$

sid sufficient to identify name and addr, but not hobby

e.g., exists a function f(sid) → name, addr

sid → name, addr is a functional dependency

"sid determines name, addr"

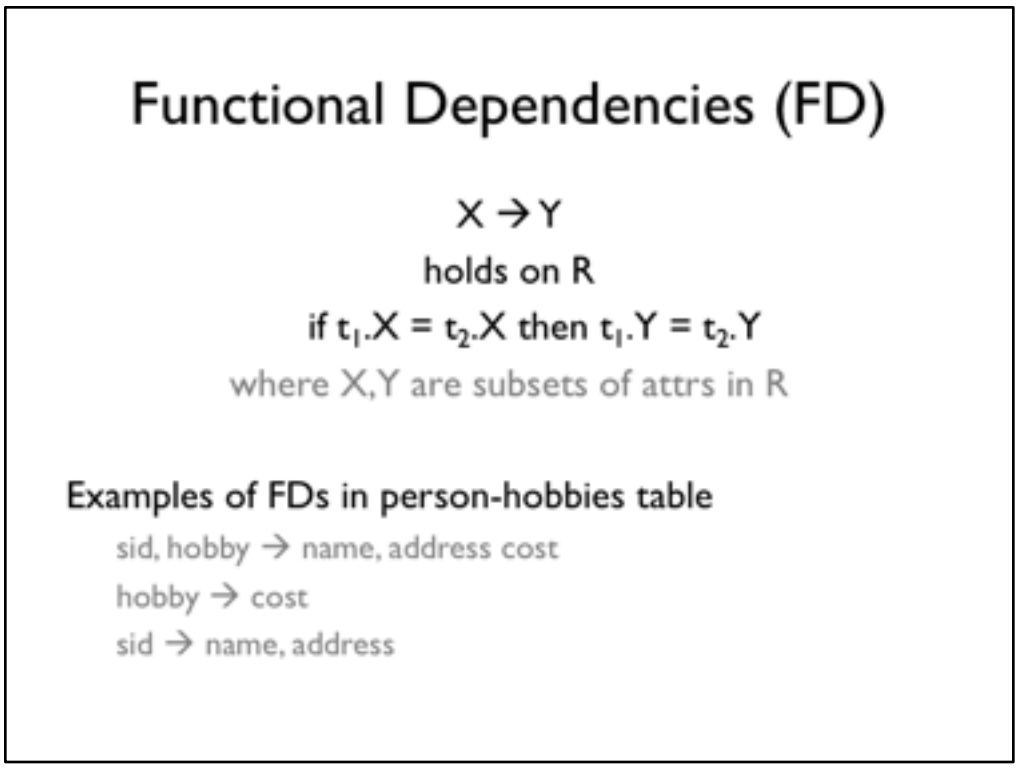
"name, addr are functionally dependent on sid"

"if 2 records have the same sid, their name and addr are the same"

So what are functional dependencies? Let's see an example:

We know that sid can tell us the name and address, meaning …

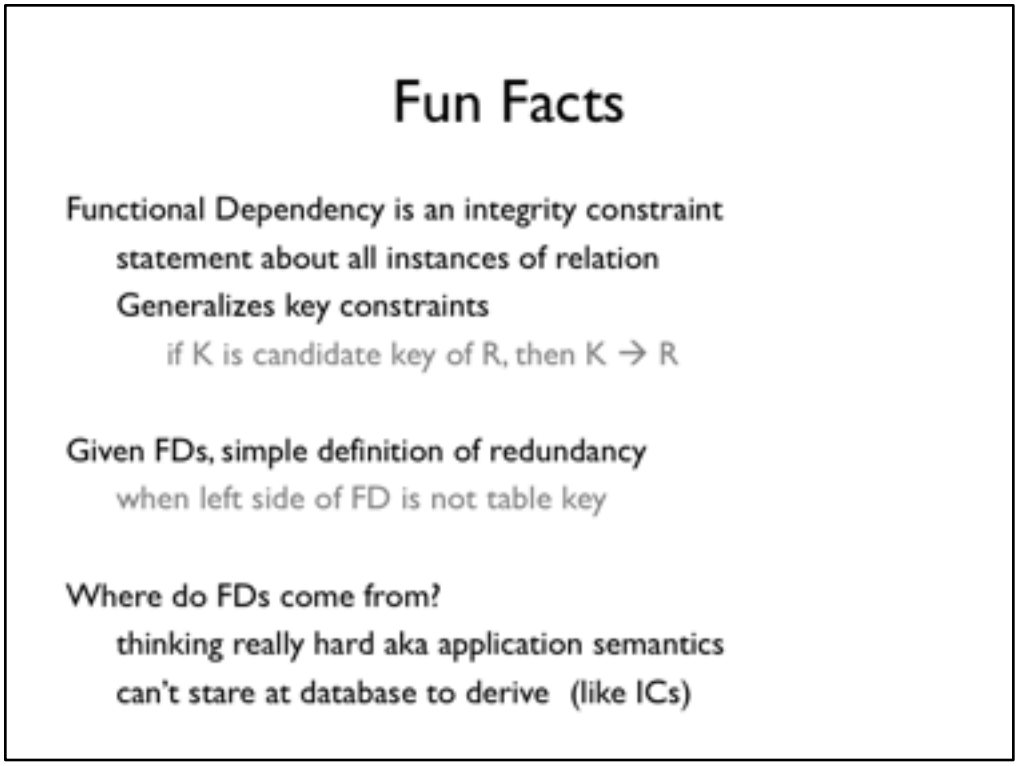
but function for sid → hobby doesn't exist
all rows with sid 1 have name eugene and address amsterdam



In simple words, if the values for the *X*attributes are known (say they are *x*), then the values for the *Y* attributes corresponding to *x* can be determined by looking them up in *any* tuple of *R* containing *x*.

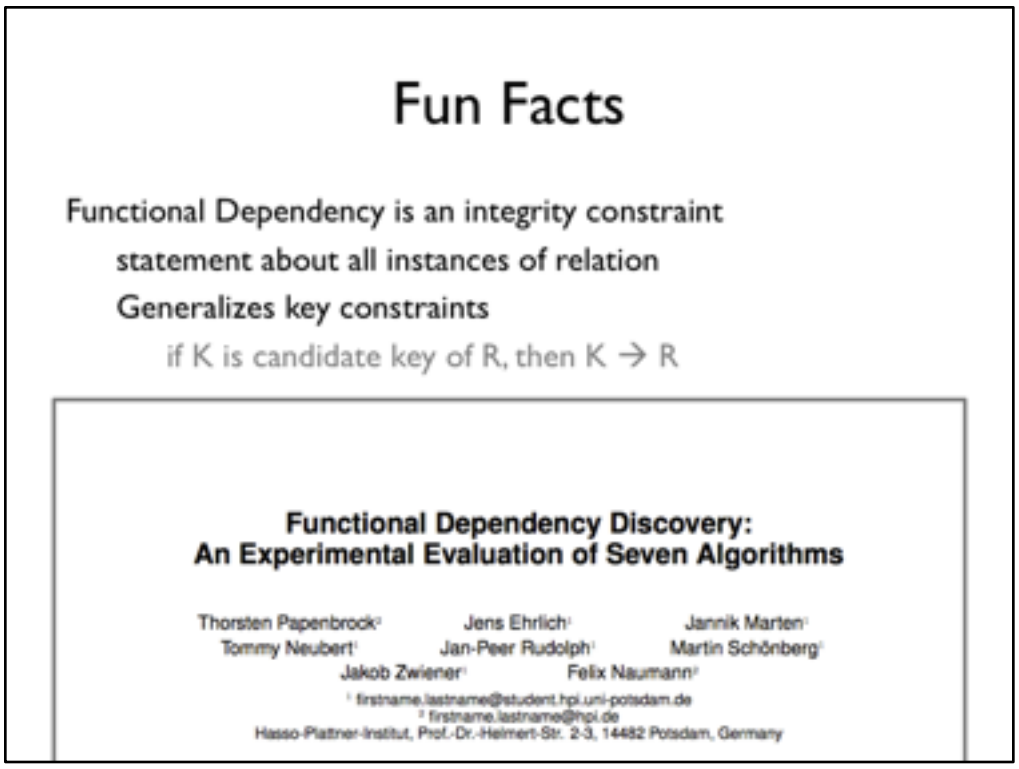
It's called a functional dependency because Y can be described as a FUNCTION of X

notice that this is a general version of key constraints we have seen: given a key X, it functionally determines all attributes in the relation (e.g., Y = all attrs in R)



Assuming we have all the functional dependencies (somehow), there is a clear definition of redundancy

remember? when we talked about integrity constraints, we said that they apply to all databases, so you can't stare at a database to figure out the ICs



Despite this truth, there are still people working on the problem of finding functional dependencies
The reality is you may not know all of them ahead of time, so you may build your application, extend it, get lots of data, and realize, oh man there are all these dependencies that I didn't realize existed.
So such tools can help you find them.

Normal Forms

Criteria met by a relation R wrt functional dependencies

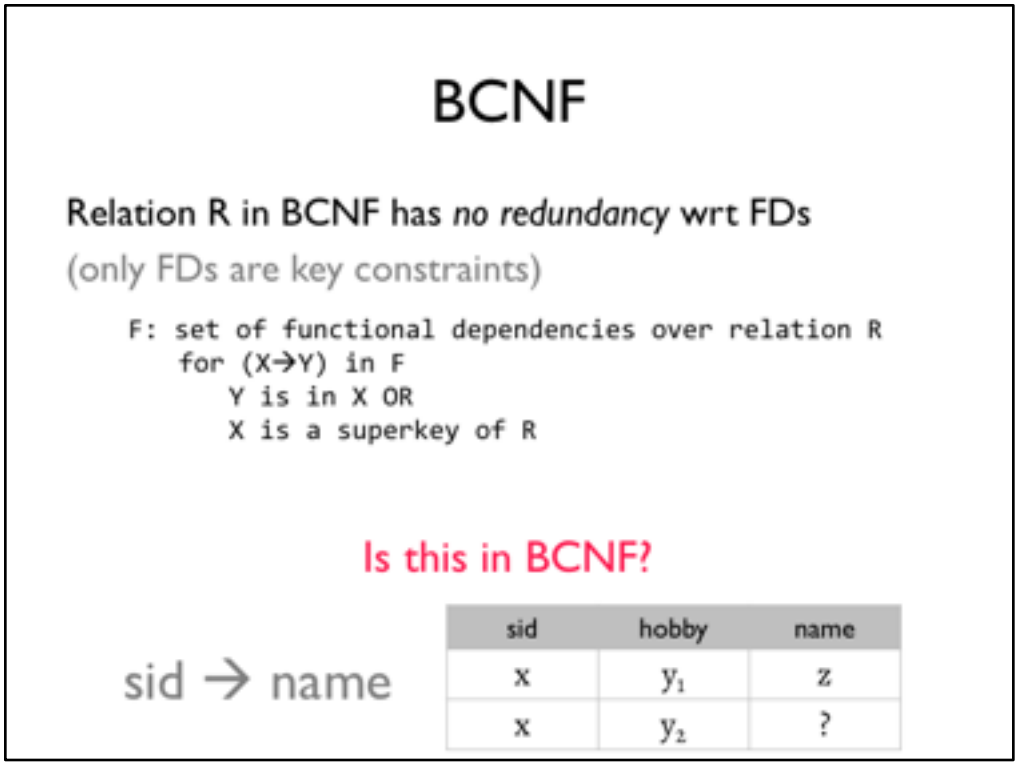
Boyce Codd Normal Form (BCNF)
No redundancy, may lose dependencies

Third Normal Form (3NF)
May have redundancy, no decomposition problems

Redundancy depends on FDs
consider R(ABC)
no FDs: no redundancy
if A→B: tuples with same A value means B is duplicated!

These normal forms provide certain guarantees about redundancy if a relation is in one of the normal forms

Say the only key is ABC in R, then A could certainly be repeated multiple times, but then B will be repeated event though its dependent



Basically, either the FD is trivial (uninteresting), or X is a key – the FD is a key constraint.
Certainly if this is the case for every relation, then each relation has a single, unambiguous candidate key and redundancy couldn't happen.

What can we deduce from this example?
name should be z

Let's say this is in BCNF, proof by contradiction
name is not a subset of sid, so sid must be a superkey, meaning that y1 must equal y2.
But relations are sets, and can only have one of any key, so this violates basic relational model
so not in BCNF

BCNF

Relation R in BCNF has *no redundancy* wrt FDs

(only FDs are key constraints)

```
F: set of functional dependencies over relation R
   for (X→Y) in F
      Y is in X OR
      X is a superkey of R
```

Functional Dependencies	What's in BCNF?	
SH → NAC (sid, hobby → name, addr, cost)	SHNAC	NO
H → C	SNA, SHC	NO
S → NA	SNA, HC, SH	YES

Notice that we cannot check these functional dependencies from these decomposed tables and need to join tables together in order to check it.
It turns out that for this example, it's ok, since SH→NAC is implied by the other two.

BCNF

Suppose we have
- Client, Office → Account
- Account → Office

What's in BCNF?
- R(Account, Client, Office)
- R(Account, Office) R(Client, Account)

Where did CO→A go? *Lost Dependency*
Can we preserve FDs *and* remove most redundancy?

We can't seem to apply CO→A by only looknig at the decomposed relations – we would need to join them together to have enough data to check CO→A

Why? Because in A→O, the O is part of another key in CO→A, so a decomposition that satisfies A→O could break up C and O into different tables!
Thus lost dependency!

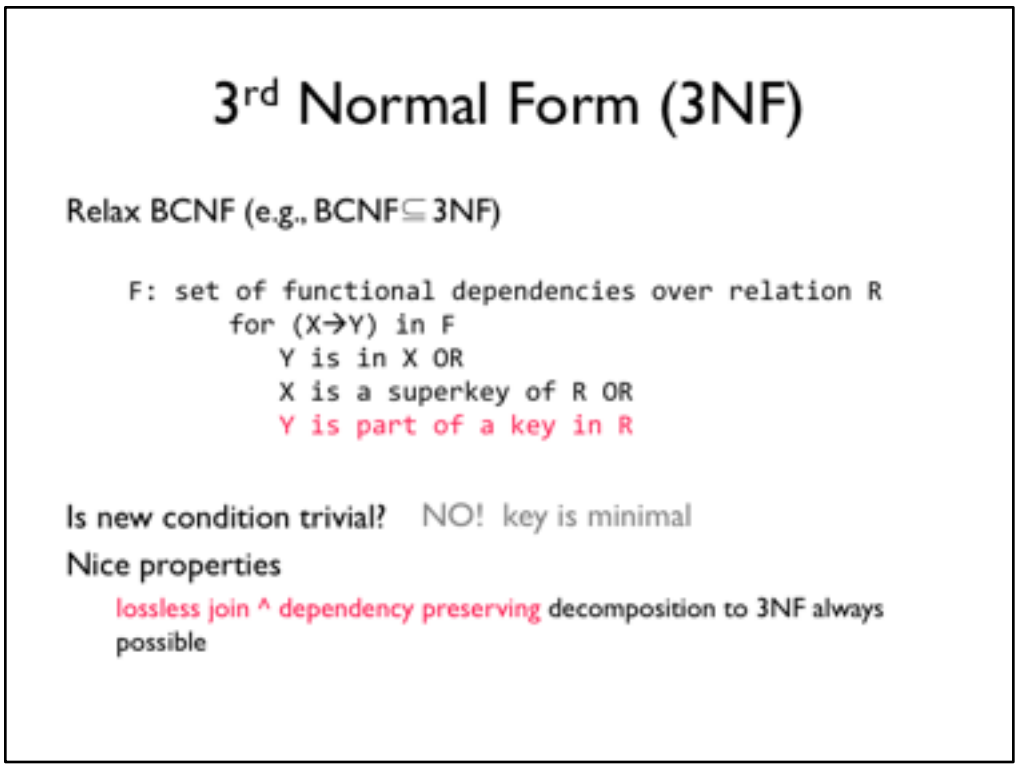
We could try to get around this by allowing some amount of redundancy so that the attributes needed for CO→A is preserved.

3rd Normal Form (3NF)

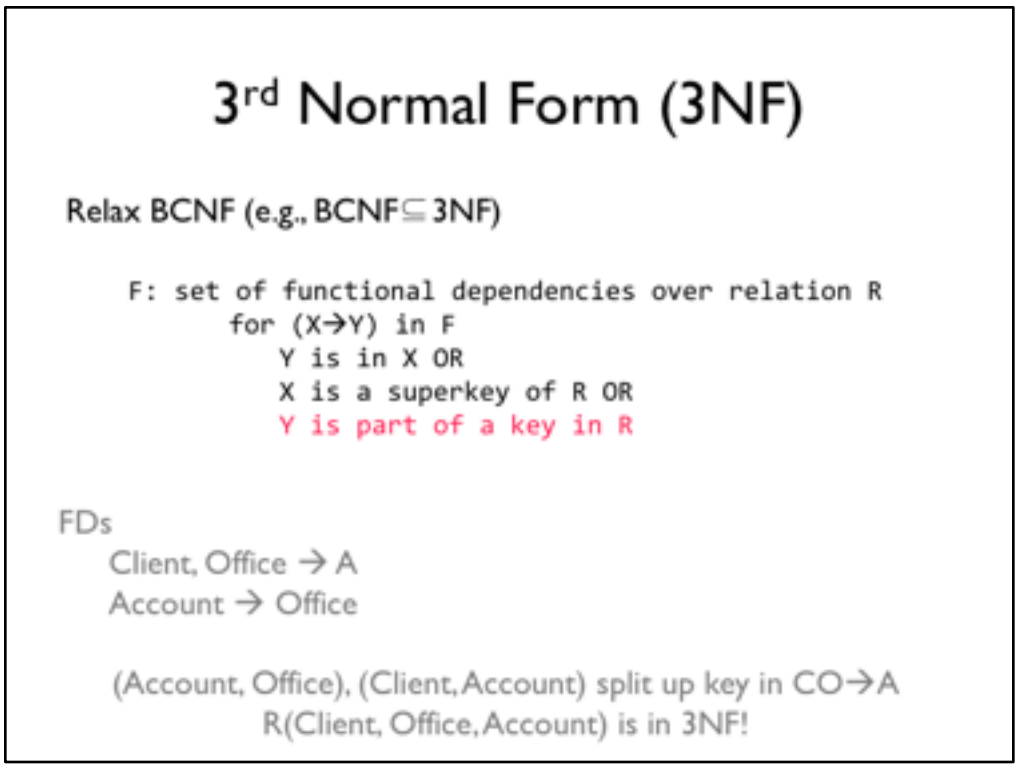
Relax BCNF (e.g., BCNF⊆3NF)

```
F: set of functional dependencies over relation R
      for (X→Y) in F
         Y is in X OR
         X is a superkey of R
```

think Y is always a part of the superkey that is the entire table
minimality of the key



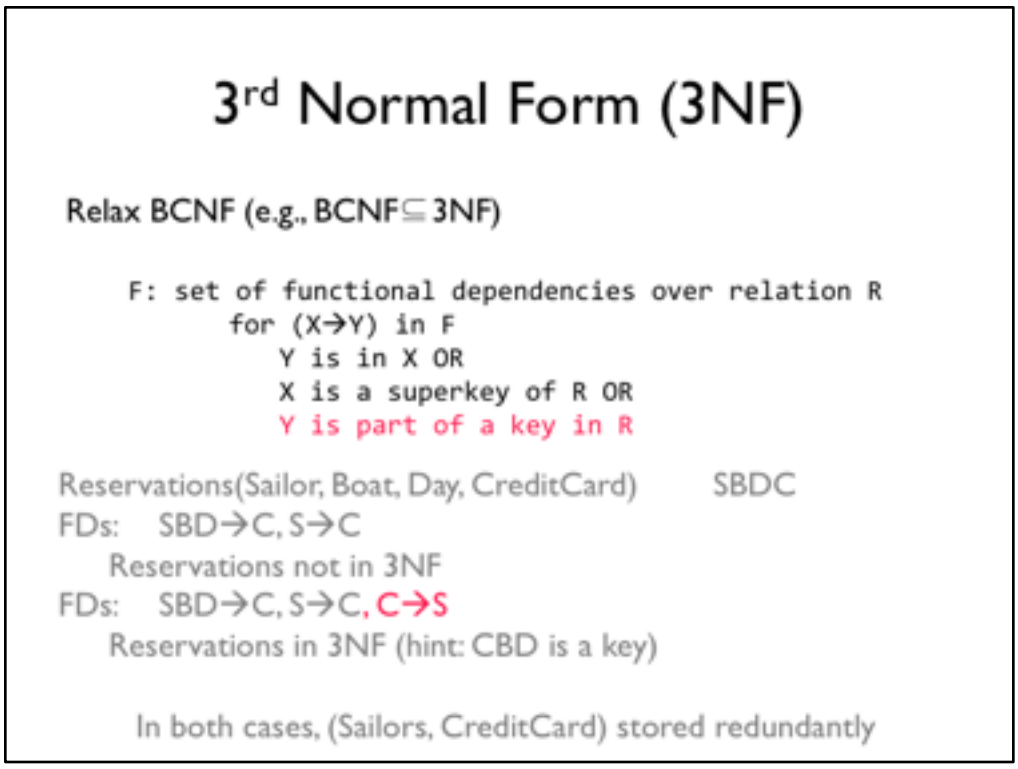
think Y is always a part of the superkey that is the entire table, thus instead must be part of a candidate KEY (minimal key).



For example, in this case Y could be the office attribute in the BCNF example. (CO→A, A→O)

intuition:
earlier we saw that BCNF loses dependencies because we have overlapping FDs (CO→A, A→O), but some decompositions can break up the key CO and cause us to lose it.
What if we instead say, let's be ok with not breaking up keys such as CO?
if X is NOT a superkey, then it's either a subset of a key or not part of a key. In either case, if it IMPLIES a part of a key,

Turns out the original table is in 3NF even though it's not in BCNF



Here is an example to show why the FDs matter in determining the normal form.
Sailors rent Boats on Days.
Sailors have credit cards.

sailor, Credit card is stored redundantly regardless of whether it is in 3ND or not

We're going to need some theory

Closure of FDs
- armstrong's axioms

Minimal FD Set

Principled Decomposition

BCNF & 3NF

Closure of FDs

If I know

Name → Bday and Bday → age

Then it implies

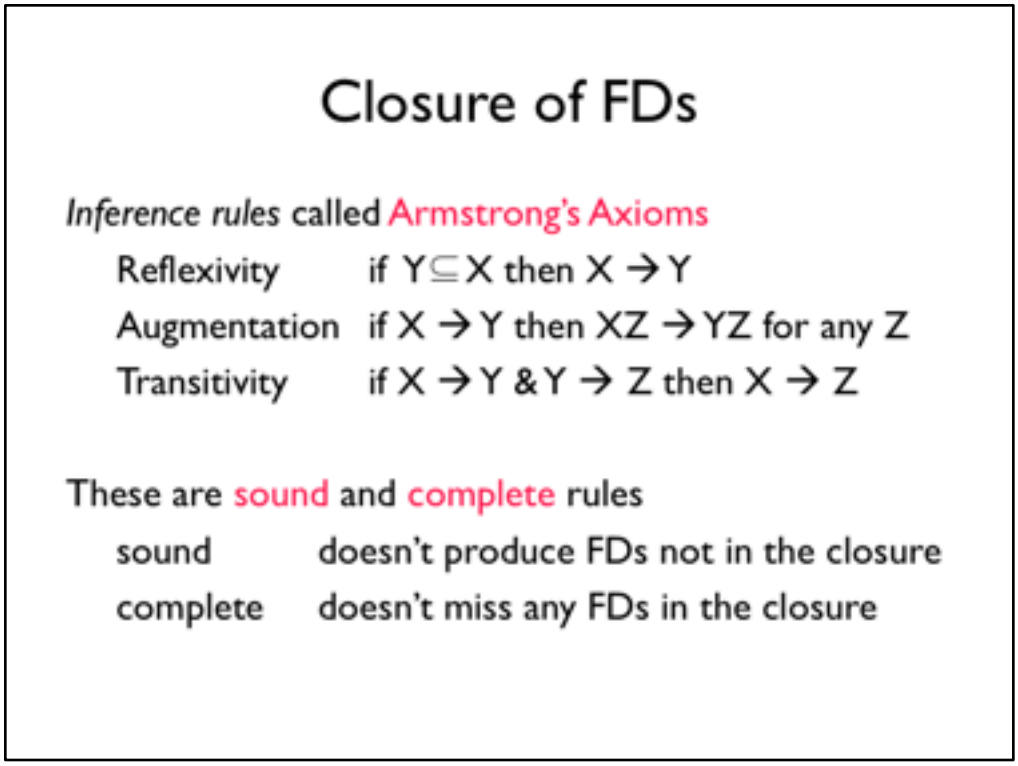
Name → age

f' is implied by set F if f' is true when F is true

F^+ closure of F is all FDs implied by F

Can we construct this closure automatically? YES

think of this as a way of compressing the functional dependencies so we don't have to list them all



They're inference rules meaning if we just keep applying these to our FDs until we don't make any more new ones, then that should give us the closure
there are additional useful rules in the book, union and decomposition

you'll hear different variations of soundness and completeness in computer science. for example, they're called false positive and false negatives in machine learning

Worth knowing

Closure of FDs

Can we compute the closure? YES. slowly

expensive. exponential in # attributes

Can we *check* if X → Y is in the closure of F?

X^+ = *attribute closure* of X (expand X using axioms)

check if Y is implied in the attribute closure

So now we have closures, so we can compare FDs in a meaningful way, and say if one set is the same as another set, or if one set is implied by another

Let's simply expand A using the rules in F

Closure of FDs

F = {A→B, B→C, CB→E}

Is A→E in the closure?

A → B	given
A → AB	augmentation A
A → BB	apply A→B
A → BC	apply B→C
BC → E	given
A → E	transitivity

So now we have closures, so we can compare FDs in a meaningful way, and say if one set is the same as another set, or if one set is implied by another

Let's simply expand A using the rules in F

We're going to need some theory

Closure of FDs

armstrong's axioms

Minimal FD Set

Principled Decomposition

BCNF & 3NF

Minimum Cover of FDs

Closures let us compare sets of FDs meaningfully

F1 = {A → B, A → C, A → BC}

F2 = {A → B, A → C}

F1 equivalent to F2

If there's a closure (a maximally expanded FD), there's a *minimal* FD. Let's find it

Minimum Cover of FDs

1. Turn FDs into *standard form*
   decompose each FD so single attr on the right side

2. Minimize left side of each FD
   for each FD, check if can delete left attr w/out changing closure
   given ABC → D, B→C can reduce to AB→D, B→C

3. Delete redundant FDs
   check each remaining FD and see if it can be deleted
   e.g., in closure of the other FDs

2 must happen before 3!

Working through (2) is the slide
abc → abcd
b → bc
ab → abc → abcd
ab → abcd → d

Minimum Cover of FDs

A→B, ABC→E, EF→G, ACF→EG

Standard form

A→B, ABC→E, EF→G, ACF→E, ACF→G

Minimize left side

A→B, AC→E, EF→G, ACF→E, ACF→G

reason: AC→E + A→B implies ABC→E

Delete Redundant FDs

A→B, AC→E, EF→G, ~~ACF→E, ACF→G~~

reason: ACF→E implied by AC→E, EF→G

let's see if ACF → E can be described using exsiting FDs. Yes:
ACF → E trivially
ACF → EF → G

We're going to need some theory

Closure of FDs
- armstrong's axioms

Minimal FD Set

Principled Decomposition

BCNF & 3NF

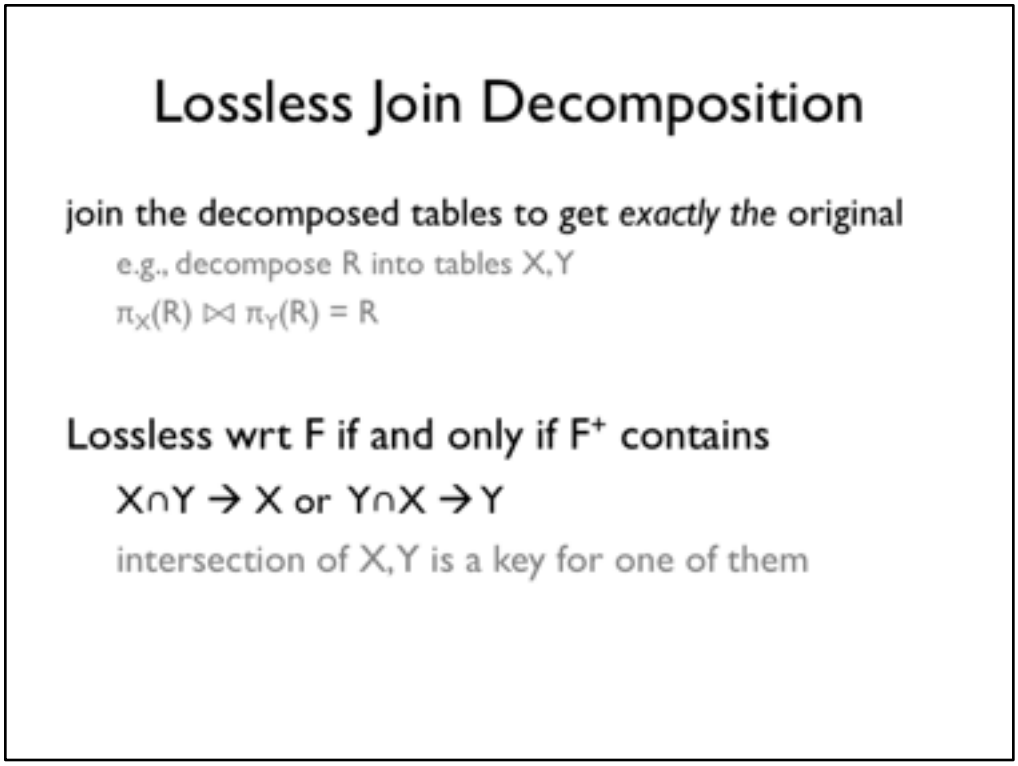
Decomposition

Eventually want to decompose R into $R_1 \ldots R_n$ wrt F

We've seen issues with decomposition.

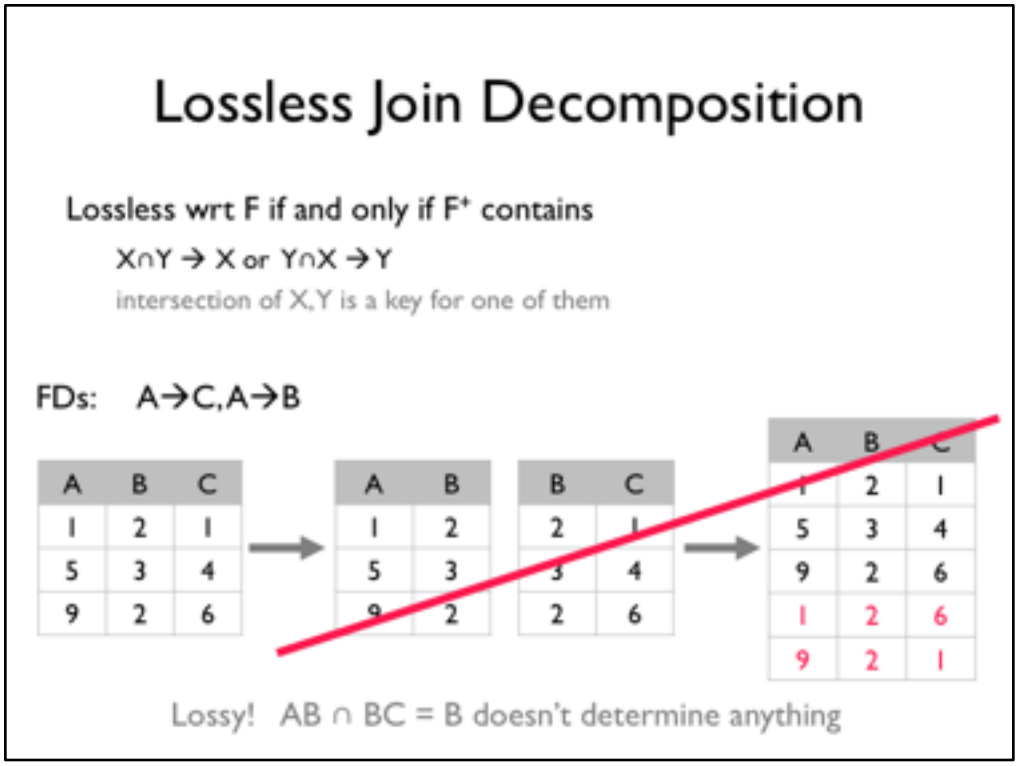
- Lost Joins: Can't recover R from $R_1 \ldots R_n$
- Lost dependencies

Principled way of avoiding these?



previous hobbies failures was because the result is a super set of original hobbies

Basically, if the decomposition is a join on a key of one of the tables, then it's ok.



B→C or B→A not in the closure!

In contrast, if had broken it up into AB, AC, would have been lossless join!

Lossless Join Decomposition

Lossless wrt F if and only if F^+ contains

$X \cap Y \rightarrow X$ or $Y \cap X \rightarrow Y$

intersection of X,Y is a key for one of them

FDs: $A \rightarrow C, A \rightarrow B$

A	B	C
1	2	1
5	3	4
9	2	6

→

A	B
1	2
5	3
9	2

A	C
1	1
5	4
9	6

→

A	B	C
1	2	1
5	3	4
9	2	6

OK

In contrast, if had broken it up into AB, AC, would have been lossless join!

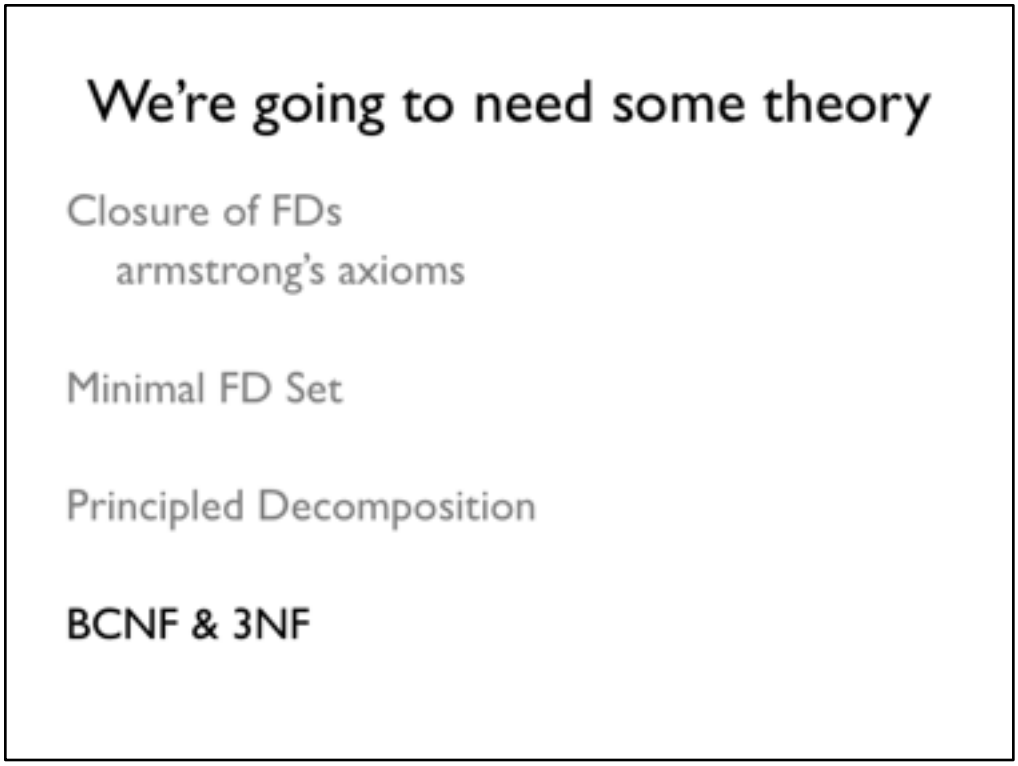
Dependency-preserving Decomposition

Terminology: F_X = Projection of F onto R
FDs U→V in F^+ s.t. U and V are in R

If R decompose to X, Y.
FDs that hold on X, Y equivalent to all FDs on R
$(F_X \cup F_Y)^+ = F^+$

Consider ABCD, C is key, AB→C, D→A
BCNF decomposition: BCD, DA
AB→C doesn't apply to either table!

Fx basically the subset of functional dependencies whose attributes are all in the relation



With a huge amount of setup, the actual algorithms for BCNF and 3NF are straighforward

BCNF

```
while BCNF is violated
   R with FDs F
   if X→Y violates BCNF
      turn R into R-Y & XY
```

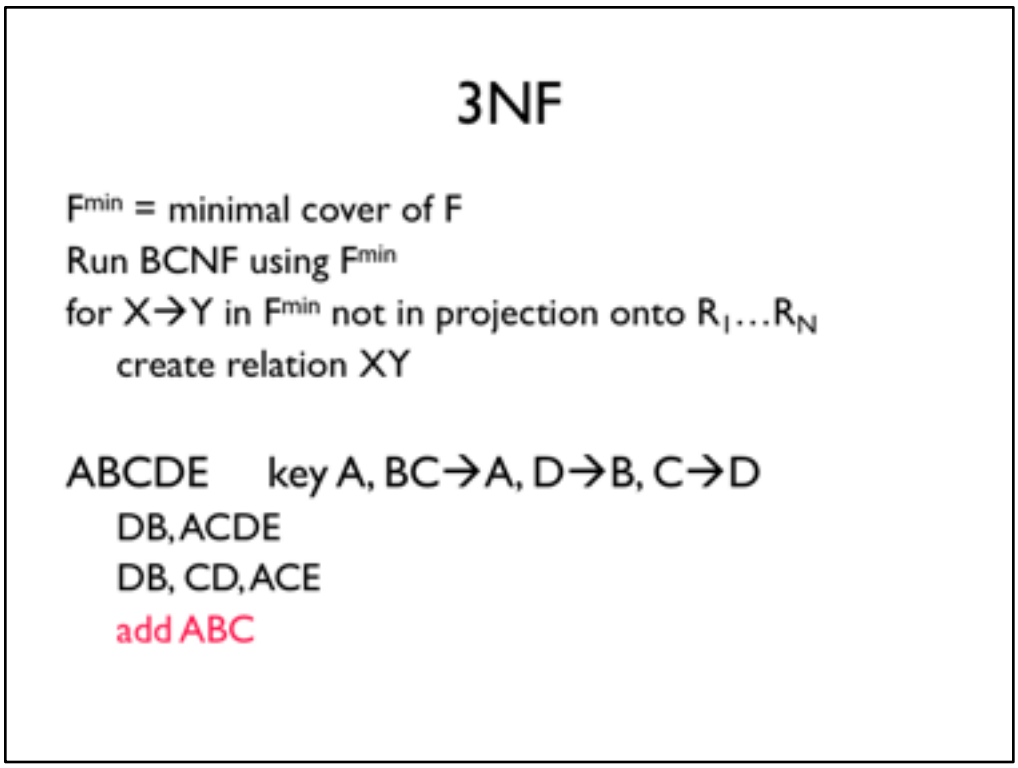
ABCDE key A, BC→A, D→B, C→D

DB, ACDE using D→B

DB, CD, ACE using C→D

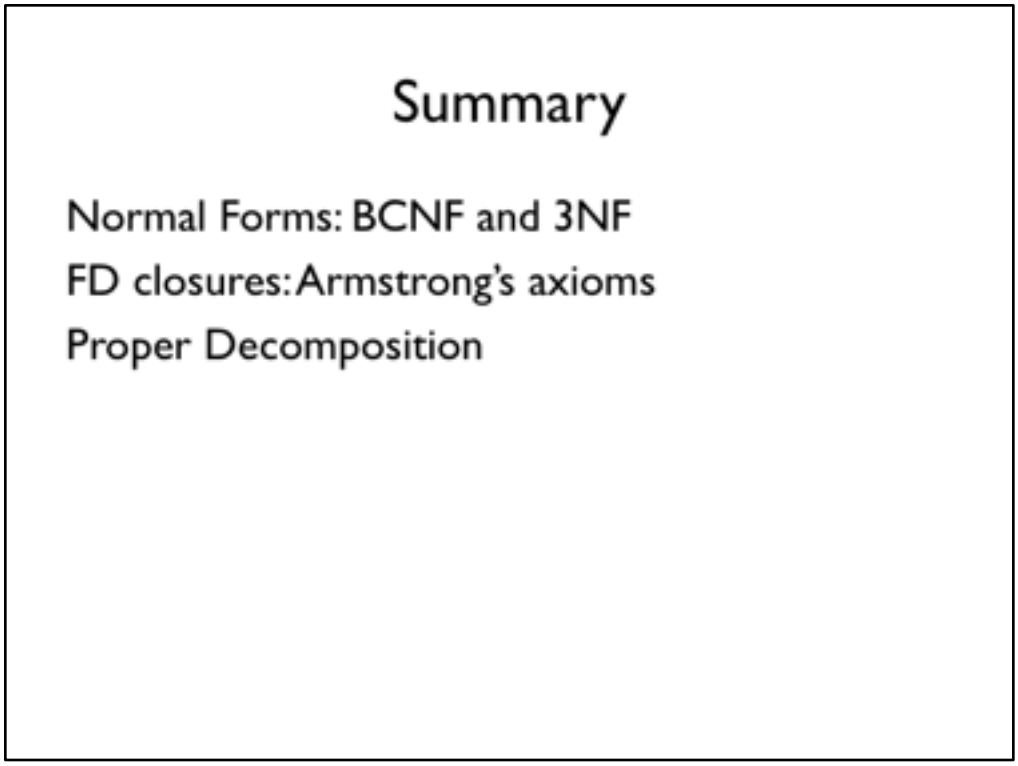
uh oh, lost BC→A

key A means A → ABCDCE



BCNF is lossless join preserving, so we just need to fix the lost dependencies. The we do this is by patching up the result of BCNF by adding relations until the lost dependencies are recovered

Note that this was easy because we already have minimal cover here, so the first step is done



Went through lots of topics today, basically

Summary

Accidental redundancy is really really bad

Adding lots of joins can hurt performance

Can be at odds with each other

Normalization good starting point, relax as needed

People usually think in terms of entities and keys, usually ends up reasonable

What you should know

Purpose of normalization
- Anomalies
- Decomposition problems
- Functional dependencies & axioms

3NF
- properties
- algorithm