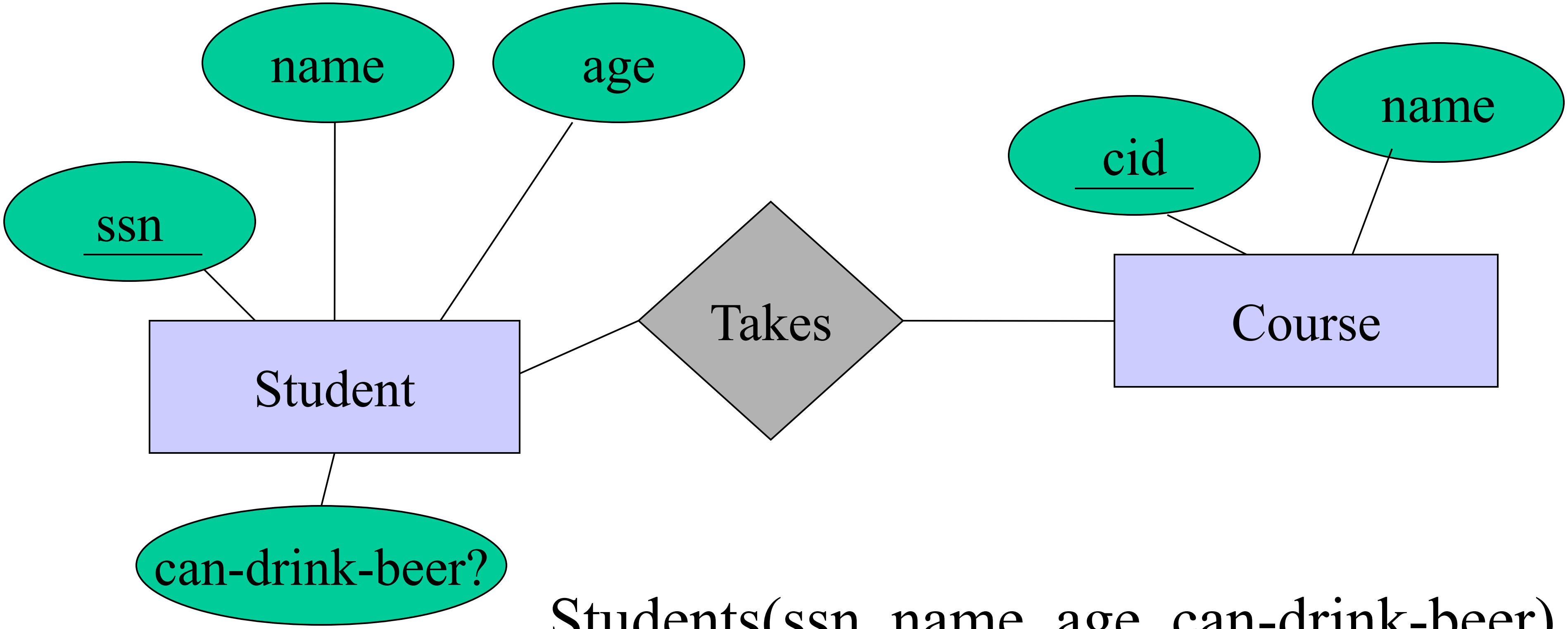
Schema Design & Refinement (aka Normalization)

Motivation

"Crouching Beer, Hidden Bratwurst" Team:



Students(ssn, name, age, can-drink-beer)

Courses(cid, name)

Takes(ssn, cid)

Example of What is Wrong

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

There is redundancy here, leading to all kinds of problems (also called anomalies)

update anomalies = update one item and forget the others
deletion anomalies = delete multiple items
if delete all, then loose information
some other anomalies too

A solultion: refine this table by breaking it down into two tables

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

Age	Can-Drink
16	no
17	no
18	yes
…	…

A solultion: refine this table by breaking it down into two tables

So instead of

Students(ssn, name, age, can-drink)

Courses(cid, name)

Take(ssn,cid)

We will have

Students(ssn, name, age)

Drink-ability(age, can-drink)

Courses(cid, name)

Take(ssn, cid)

Need a general solution that works on any relational schema

- **Intuition**
  - given an ER diagram
  - translate it into a relational schema R
  - think about all dependency constraints that can apply to R
    - such as “age determines can-drink-beer”
  - use these constraints to detect if R is a bad schema
    - such as having some kind of redundancy
  - then refine R into a schema R* with less redundancy

In practice, dependencies such as age ➔ can-drink are called "functional dependencies".

We need to first formalize and study (1) functional dependencies, and (2) keys for tables before we can talk about (1) how to detect bad tables, and (2) how to break them down

Functional Dependencies

- A form of constraint (hence, part of the schema)
- Finding them is part of the database design
- Used heavily in schema refinement

Definition:

If two tuples agree on the attributes

$A_1, A_2, \ldots A_n$

then they must also agree on the attributes

$B_1, B_2, \ldots B_m$

Formally: $A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$

Examples

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN $\rightarrow$ Name, Age, Can-Drink

Age $\rightarrow$ Can-Drink

SSN, Age $\rightarrow$ Name, Can-Drink

Examples

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E1847	John	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	lawyer

- EmpID $\longrightarrow$ Name, Phone, Position
- Position $\longrightarrow$ Phone
- but Phone $\not\longrightarrow$ Position

How Do We Infer FDs?

- Create ER Diagram
- Translate into a relational schema
- Think hard about what FDs are valid for that relational schema
  - think from an application point of view

- An FD is an inherent property of an application
- It is not something we can infer from a set of tuples

How Do We Infer FDs?

- Given a table with a set of tuples
  - the best we can do is confirm that a FD seems to be valid
  - or to infer that a FD is definitely invalid
  - we can never prove that a FD is valid

In General

- To confirm A $\longrightarrow$ B, erase all other columns

…	A	…	B	
	X 1		Y 1	
	X 2		Y 2	
	…		…	

- check if the remaining relation is many-one
  – if yes, then the FD is probably valid
  – if no, then the FD is definitely invalid

Example: Position ➔ Phone

EmpID	Name	Phone	Position
E0045	Smith	1234 ←	Clerk
E1847	John	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234 ←	lawyer

How about Name ➔ Phone?

Keys

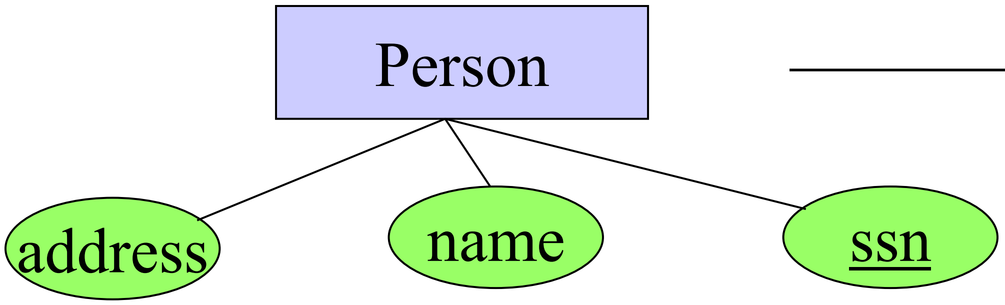
- Key of a relation R is a set of attributes that
  - functionally determines all attributes of R
  - none of its subsets determines all attributes of R
- Superkey
  - a set of attributes that contains a key
- We will need to know the keys of the relations in a DB schema, so that we can refine the schema

Finding the Keys of a Relation

Given a relation constructed from an E/R diagram, what is its key?

Rules:

1. If the relation comes from an entity set,
   the key of the relation is the set of attributes which is the key of the entity set.

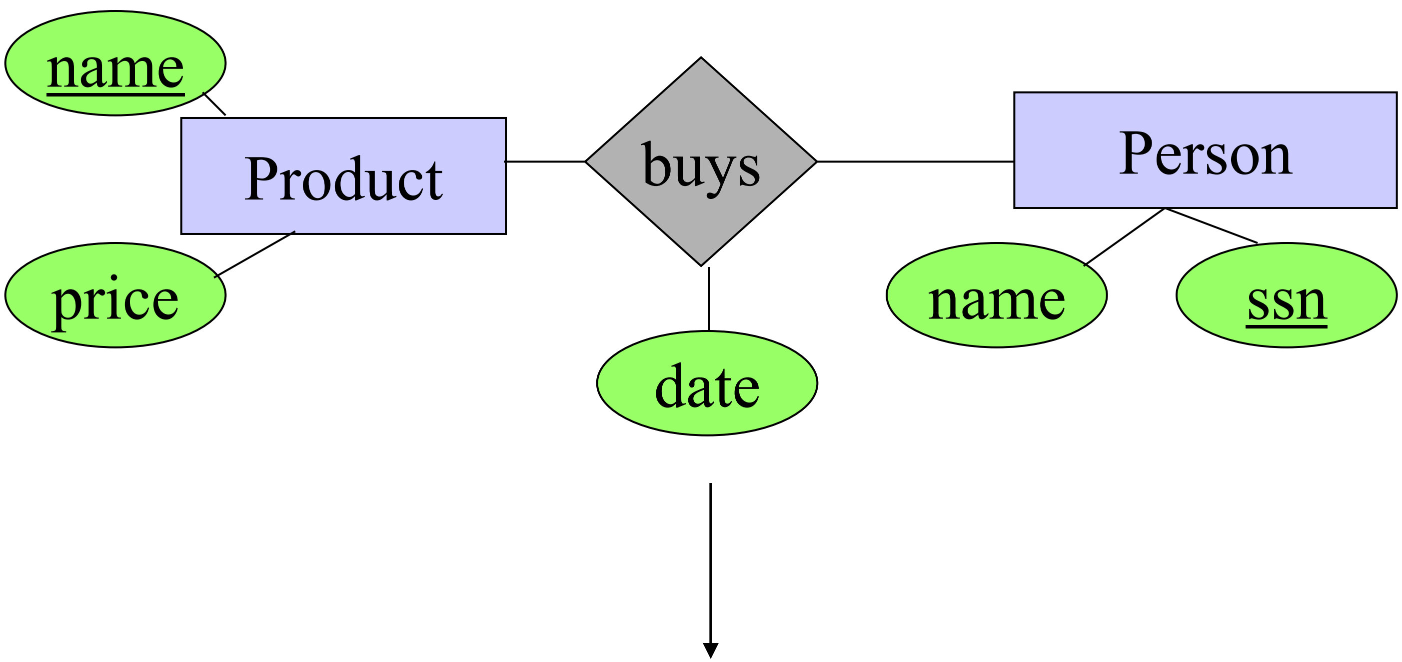


Person(address, name, <u>ssn</u>**)**

Finding the Keys

Rules:

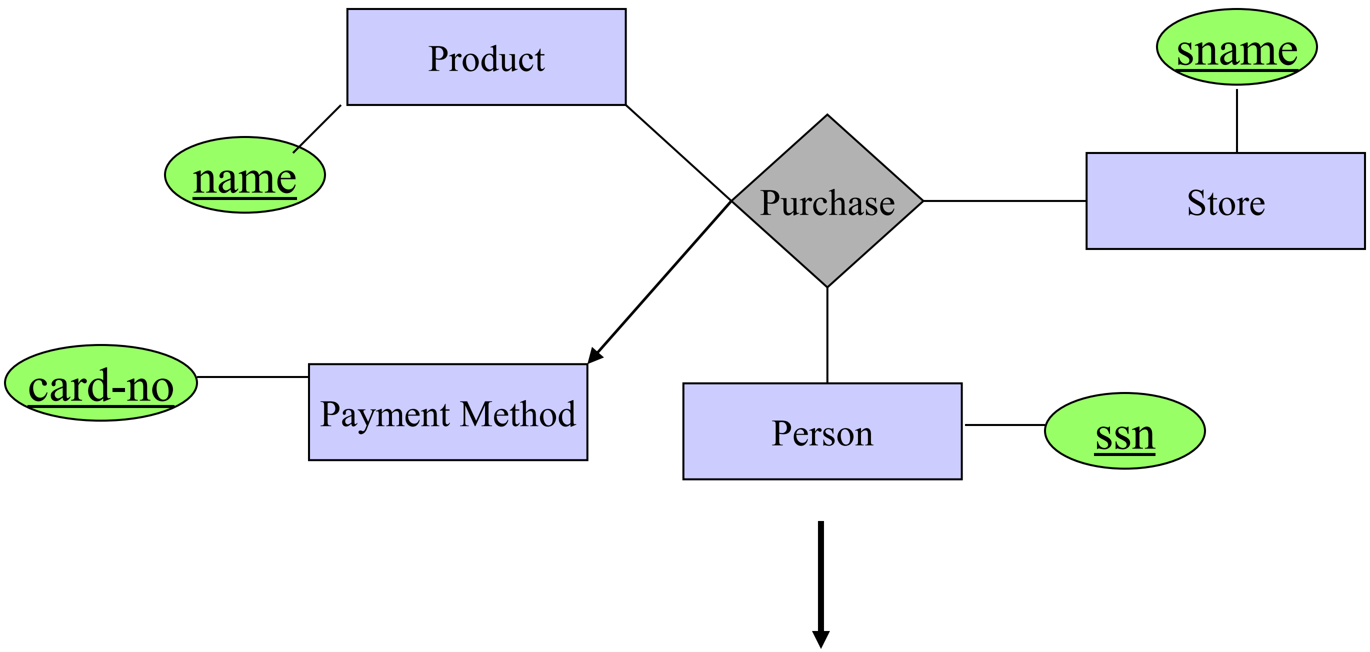
2. If the relation comes from a many-many relationship, the key of the relation is the set of all attribute keys in the relations corresponding to the entity sets



buys(<u>name, ssn</u>, date)

Finding the Keys

But: if there is an arrow from the relationship to E, then we don't need the key of E as part of the relation key.



Purchase(<u>name , sname, ssn</u>, card-no)

Finding the Keys

More rules:

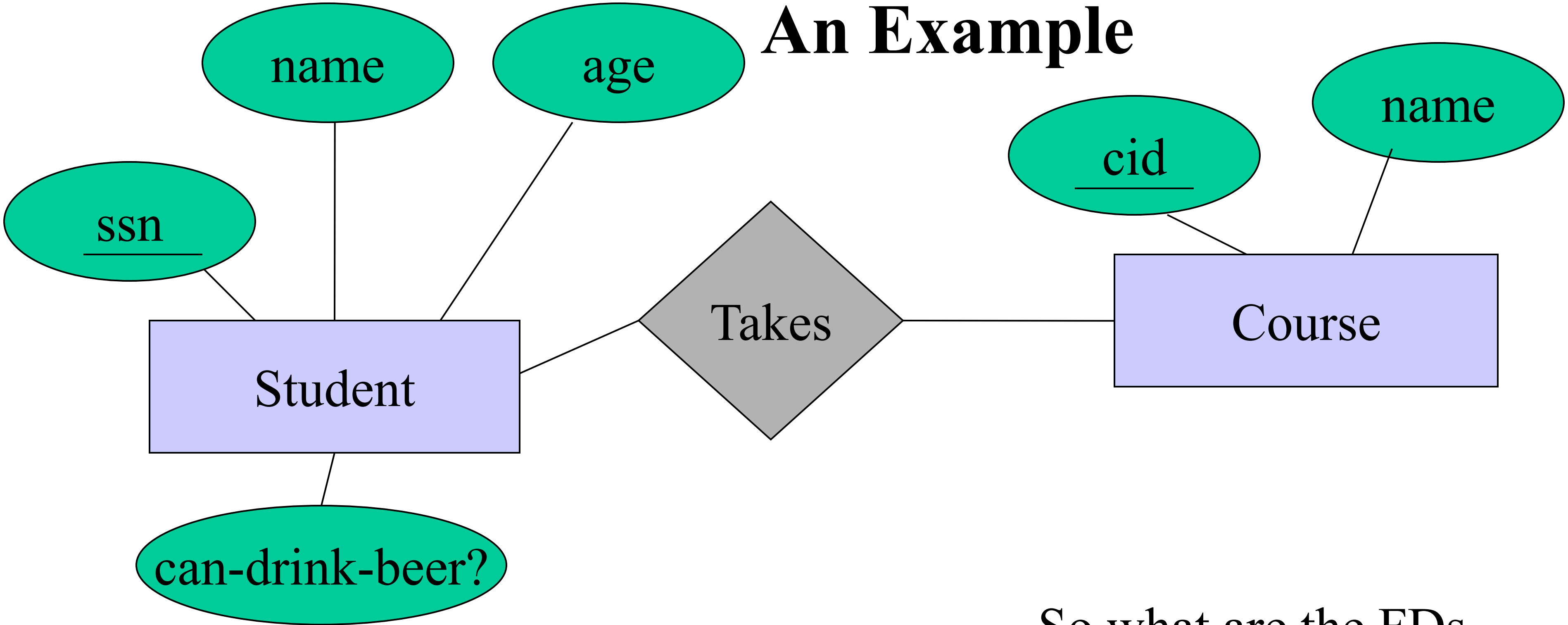
- Many-one, one-many, one-one relationships
- Multi-way relationships
- Weak entity sets

(Try to find them yourself)

Why specifying keys and FDs?

- Why keys?
  - help identify entities/tuples
  - imply certain FDs
- Why FDs?
  - give us more integrity constraints for the application
- More importantly
  - having keys and FDs will help us detect that a table is "bad", and helps us determine how to decompose the table

An Example



Students(ssn, name, age, can-drink)

Courses(cid, name)

Takes(ssn, cid)

So what are the FDs inferred from keys?

ssn ➔ ...

cid ➔ ...

We also add

age ➔ can-drink

Once the team has specified some keys and FDs, we can't just stop there

- We want to infer all FDs that may be logically implied
  - e.g., if team says A $\rightarrow$ B, B $\rightarrow$ C, then we also have A $\rightarrow$ C
- Given a set of attributes, we also want to infer all attributes that are functionally determined by these given attributes
- Knowing these will help us detect if a table is bad and how to decompose it

Inferring All FDs

- Given a relation schema R & a set S of FDs
  - is the FD f logically implied by S?
- Example
  - R = {A,B,C,G,H,I}
  - S = A ➔B, A ➔C, CG ➔ H, CG ➔ I, B ➔ H
  - would A ➔ H be logically implied?
  - yes (you can prove this, using the definition of FD)
- Closure of S: S+ = all FDs logically implied by S
- How to compute S+?
  - we can use Armstrong's axioms

Armstrong's Axioms

- Reflexivity rule
  - A1A2...An ➔ a subset of A1A2...An
- Augmentation rule
  - A1A2...An ➔ B1B2...Bm, then
    A1A2...An C1C2..Ck ➔ B1B2...Bm C1C2...Ck
- Transitivity rule
  - A1A2...An ➔ B1B2...Bm and
    B1B2...Bm ➔ C1C2...Ck, then
    A1A2...An ➔ C1C2...Ck

Inferring S+ using Armstrong's Axioms

- S+ = S
- Loop
  - foreach f in S, apply reflexivity and augment. rules
  - add the new FDs to S+
  - foreach pair of FDs in S, apply the transitivity rule
  - add the new FD to S+
- Until S+ does not change any further
- Basically, just apply rules until can't apply anymore

Additional Rules

- Union rule
  - $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - (X, Y, Z are sets of attributes)
- Decomposition rule
  - $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Pseudo-transitivity rule
  - $X \rightarrow Y$ and $YZ \rightarrow U$, then $XZ \rightarrow U$
- These rules can be inferred from Armstrong's axioms

Find All Attributes that are Functionally Determined by a Set of Attributes

Given a set of attributes **{*A1*, …, *An*}** and a set of dependencies S.
Problem: find all attributes *B* such that:

any relation which satisfies S also satisfies:

$A1, \ldots, An \rightarrow B$

That is, all attributes B that are functionally determined by the Ai

The **closure** of *{A1, ..., An}* is the set of all such attributes *B*

Algorithm to Compute Closure

Start with X={A1, …, An}.

Repeat until X doesn't change **do**:

if $B_1, B_2, \ldots B_n \longrightarrow C$ is in S, **and**

$B_1, B_2, \ldots B_n$ are all in X, **and**

C is not in X

then

add C to X.

Just apply FDs until can't apply anymore

Example

$$
\begin{aligned}
A\;B &\longrightarrow C \\
A\;D &\longrightarrow E \\
B &\longrightarrow D \\
A\;F &\longrightarrow B
\end{aligned}
$$

Closure of {A,B}: X = {A, B, C, D, E}

Closure of {A, F}: X = {A, F, B, D, C, E}

Usage for Attribute Closure

- Test if X is a superkey
  - compute X+, and check if X+ contains all attrs of R
- Check if X ➔ Y holds
  - by checking if Y is contained in X+
- Another way to compute closure S+ of FDs
  - for each subset of attributes X in relation R, compute X+
  - for each subset of attributes Y in X+, output the FD X ➔ Y

Review

- We have learned about keys and FDs
- We have learned about how to reason with them
  - given a set of FDs, infer all new applicable FDs
  - given a set of attributes X, infer all new attributes that are functionally determined by X
- Now we will look at how to use them to detect that a table is "bad".

- We say a table is "bad" if it is not in Boyce-Codd normal form

Boyce-Codd Normal Form

A relation R is in BCNF if and only if:

Whenever there is a nontrivial FD $A_1, A_2, \ldots A_n \longrightarrow B$ for R, it is the case that $\{ A_1, A_2, \ldots A_n \}$ is a super-key for R.

Example: This is not in BCNF

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

ssn ➔ name, age, can-drink

age ➔ can-drink

ssn is a key

For each FD A ➔ B, ask: is A a superkey?

If not, then the FD violates BCNF, relation is not in BCNF

To do so: (a) from current set of FDs, infer all FDs

(b) find the closure of A

Example in BCNF

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

ssn ➔ name, age

Age	Can-Drink
16	no
17	no
18	yes
…	…

age ➔ can-drink

Any relation of only two attributes is in BCNF

Example of non-BCNF

Name	SSN	Phone Number
Fred	123-321-99	(201) 555-1234
Fred	123-321-99	(206) 572-4312
Joe	909-438-44	(908) 464-0028
Joe	909-438-44	(212) 555-4000

What are the dependencies?

SSN $\rightarrow$ Name

What are the keys?

Is it in BCNF?

Example of BCNF

SSN	Name
123-321-99	Fred
909-438-44	Joe

SSN $\rightarrow$ Name

SSN	Phone Number
123-321-99	(201) 555-1234
123-321-99	(206) 572-4312
909-438-44	(908) 464-0028
909-438-44	(212) 555-4000

What About This?

Name	Price	Category
Gizmo	$19.99	gadgets
OneClick	$24.99	camera

Name $\longrightarrow$ Price, Category

How to Detect that a Table is not in BCNF?

A relation R is in BCNF if and only if:

Whenever there is a nontrivial FD $A_1, A_2, \ldots A_n \longrightarrow B$ for R, it is the case that $\{ A_1, A_2, \ldots A_n \}$ is a super-key for R.

So we start by creating the ER diagram, specifying keys

Then translate it into relational tables, specifying keys

Then add as many FDs as we can think of

Then infer all other FDs

Then for each FD X ➔ Y, check if X is a superkey

(a key is also a superkey); one way to do this is to compute the closure of X

Once we know that a table is not in BCNF, how do we decompose it?

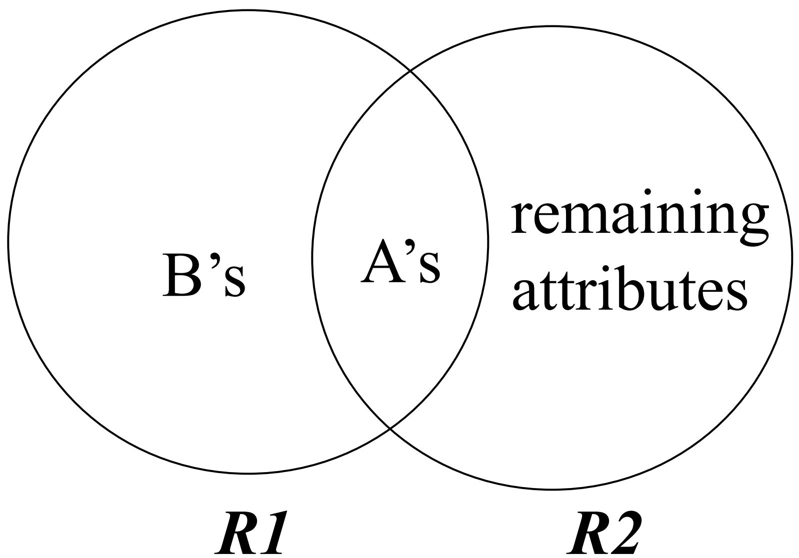
BCNF Decomposition

Find a dependency that violates the BCNF condition:

$$A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$$

Heuristics: expand $B_1, B_2, \ldots B_m$ "as much as possible"

Decompose:



Continue until there are no BCNF violations left.

Any 2-attribute relation is in BCNF.

Decompose into BCNF

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN ➔ name, age, can-drink

age ➔ can-drink

Age	Can-Drink
16	no
17	no
18	yes
…	…

age ➔ can-drink

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

ssn ➔ name, age

Example Decomposition

Person:

Name	SSN	Age	EyeColor	PhoneNumber

Functional dependencies:

SSN $\longrightarrow$ Name, Age, Eye Color

BNCF: R1(SSN, Name, Age, EyeColor),
R2(SSN, PhoneNumber)

Another Example

- Persons(SSN, name, age, eye-color, phone, can-drink)
  – SSN → name, age, eye-color
  – age → can-drink
- What we can infer from the above?
  – SSN → name, age, eye-color, can-drink
  – SSN is NOT a key nor a superkey
  – not in BCNF
- Decomposing
  – use SSN → name, age, eye-color, can-drink (biggest expansion)
  – R1(SSN, name, age, eye-color, can-drink)
  – R2(SSN, phone)

Another Example

- Decomposing
  - use SSN → name, age, eye-color, can-drink
  - R1(SSN, name, age, eye-color, can-drink)
    SSN → name, age, eye-color, can-drink
    age → can-drink
  - R2(SSN, phone)
- Need to decompose R1, using age → can-drink
  - R3(age, can-drink)
    age → can-drink
  - R4(age, SSN, name, eye-color)
    SSN → age, name, eye-color
  - R2(SSN, phone)

We have learned
(a) how to detect that a table in not in BCNF, (b) how to decompose it.

How do we know that this decomposition is a good one? What do we mean by "good" here?

Desirable Properties of Schema Decomposition (that is, Schema Refinement)

1) minimize redundancy
2) avoid info loss
3) preserve dependency
4) ensure good query performance

Decompositions in General

Let R be a relation with attributes $A_1, A_2, \ldots A_n$

Create two relations *R1* and *R2* with attributes

$B_1, B_2, \ldots B_m$ $\qquad$ $C_1, C_2, \ldots C_l$

Such that:

$$B_1, B_2, \ldots B_m \cup C_1, C_2, \ldots C_l = A_1, A_2, \ldots A_n$$

And

-- *R1* is the projection of *R* on $B_1, B_2, \ldots B_m$

-- *R2* is the projection of *R* on $C_1, C_2, \ldots C_l$

Example

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

Age	Can-Drink
16	no
17	no
18	yes
…	…

Desirable Property #1: Minimize redundancy

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

Age	Can-Drink
16	no
17	no
18	yes
…	…

Certain Decomposition May Cause Info Loss

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
DoubleClick	29.99	Camera

Decompose on : **Name, Category** and **Price, Category**

Name	Category
Gizmo	Gadget
OneClick	Camera
DoubleClick	Camera

Price	Category
19.99	Gadget
24.99	Camera
29.99	Camera

When we put it back:

Cannot recover information

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
OneClick	**29.99**	**Camera**
DoubleClick	**24.99**	**Camera**
DoubleClick	29.99	Camera

Lossless Decompositions

A decomposition is *lossless* if we can recover:

R(A,B,C)

Decompose

R1(A,B) R2(A,C)

Recover

R'(A,B,C) should be the same as R(A,B,C)

R' is in general larger than R. Must ensure R' = R

Put Another Way: "Lossless" Joins

- The main idea: if you decompose a relation schema, then join the parts of an instance via a natural join, you might get more rows than you started with, i.e., spurious tuples
  - This is bad!
  - Called a "lossy join".
- Goal: decompositions which produce only "lossless" joins
  - "non-additive" join is more descriptive
  - because we don't want to add more tuples
- Desirable Property #2: Lossless decomposition

Dependency Preserving

- Given a relation R and a set of FDs S
- Suppose we decompose R into R1 and R2
- Suppose
  - R1 has a set of FDs S1
  - R2 has a set of FDs S2
  - S1 and S2 are computed from S
- We say the decomposition is dependency preserving if by enforcing S1 over R1 and S2 over R2, we can enforce S over R

Example

SSN	Name	Age	Can-Drink
1	Dave	16	no
2	Mike	17	no
3	Jane	16	no
4	Liu	19	oh yes

SSN → name, age, can-drink

age → can-drink

SSN	Name	Age
1	Dave	16
2	Mike	17
3	Jane	16
4	Liu	19

ssn → name, age

Age	Can-Drink
16	no
17	no
18	yes
…	…

age → can-drink

Another Example

Unit	Company	Product

FD's: Unit → Company; Company, Product → Unit

Consider the decomposition:

Unit	Company

Unit → Company

Unit	Product

No FDs

So What's the Problem?

Unit	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	databases
Bingo	databases

No problem so far. All *local* FD's are satisfied.

Let's put all the data back into a single table again:

Unit	Company	Product
Galaga99	UW	databases
Bingo	UW	databases

Violates the dependency: company, product -> unit!

Preserving FDs

- Such a decomposition is not “dependency-preserving.”
- Desirable Property #3: always have FD-preserving decompositions

- We will talk about "Desirable Property #4: Ensure Good Query Performance" later

Review

- When decomposing a relation R, we want the decomposition to
  - minimize redundancy
  - avoid info loss
  - preserve dependencies (i.e., constraints)
  - ensure good query performance
- These objectives can be conflicting
- Boyce-Codd normal form achieves some of these

In particular

- BCNF removes certain types of redundancy
- For examples of redundancy that it cannot remove, see "multivalued redundancy"
- BCNF avoids info loss
- BCNF is not always dependency preserving

Recall: Lossless Decompositions

A decomposition is *lossless* if we can recover:

$$R(A,B,C) \xrightarrow{\text{Decompose}} \{\ R1(A,B)\ ,\ \ R2(A,C)\ \} \xrightarrow{\text{Recover}} R'(A,B,C) = R(A,B,C)$$

R' is in general larger than R. Must ensure R' = R

Decomposition Based on BCNF is Necessarily Lossless

R(A, B, C), $A \rightarrow C$

BCNF: R1(A,B), R2(A,C)

Some tuple (a,b,c) in R (a,b',c') also in R
decomposes into (a,b) in R1 (a,b') also in R1
and (a,c) in R2 (a,c') also in R2

Recover tuples in R: (a,b,c), (a,b,c'), (a,b',c), (a,b',c') also in R ?

Can (a,b,c') be a bogus tuple? What about (a,b',c') ?

However,

- BCNF is not always dependency preserving
- In fact, some times we cannot find a BCNF decomposition that is dependency preserving

An Example

Unit	Company	Product

FD's: Unit → Company; Company, Product → Unit

Consider the decomposition:

Unit	Company

Unit → Company

Unit	Product

No FDs

BCNF is called a “normal form”.
Many other types of normal forms exist.

First Normal Form = all attributes are atomic

Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)

Third Normal Form (3NF)

Fourth Normal Form (4NF)

Others...

3rd Normal Form (3NF)

A simple condition for removing anomalies from relations:

A relation R is in 3rd normal form if :

Whenever there is a nontrivial dependency $A_1, A_2, ..., A_n \rightarrow B$ for R , then $\{A_1, A_2, ..., A_n\}$ is a super-key for R,
or B is part of a key.

An Example

Unit	Company	Product

FD's: Unit → Company; Company, Product → Unit

3NF (General Definition)

- A relation is in Third Normal Form (3NF) if whenever $X \rightarrow A$ holds, either X is a superkey, or A is a prime attribute.

Informally: everything depends on the key or is in the key.

- Despite the thorny technical definitions that lead up to it, 3NF is intuitive and not hard to achieve. ***Aim for it in all designs unless you have strong reasons otherwise.***

3NF vs. BCNF

- R is in BFNC if whenever $X \rightarrow A$ holds, then X is a superkey.
- Slightly stronger than 3NF.
- Example: R(A,B,C) with $\{A,B\} \rightarrow C$, $C \rightarrow A$
  - 3NF but not BCNF

Guideline: Aim for BCNF and settle for 3NF

Decomposing R into 3NF

- The algorithm is complicated
- 1. Get a “minimal cover” of FDs
- 2. Find a lossless-join decomposition of R (which might miss dependencies)
- 3. Add additional relations to the decomposition to cover any missing FDs of the cover
- Result will be lossless, will be dependency-preserving 3NF; might not be BCNF

Normal Forms

First Normal Form = all attributes are atomic

Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)

Third Normal Form (3NF)

Fourth Normal Form (4NF)

Others...

Multi-valued Dependencies

SSN	Phone Number	Course
123-321-99	(206) 572-4312	CSE-444
123-321-99	(206) 572-4312	CSE-341
123-321-99	(206) 432-8954	CSE-444
123-321-99	(206) 432-8954	CSE-341

The multi-valued dependencies are:

SSN $\twoheadrightarrow$ Phone Number

SSN $\twoheadrightarrow$ Course

Definition of Multi-valued Dependency

Given $R(A1,\ldots,An,B1,\ldots,Bm,C1,\ldots,Cp)$

the MVD $A1,\ldots,An \twoheadrightarrow B1,\ldots,Bm$ holds if:

for any values of $A1,\ldots,An$
the "set of values" of $B1,\ldots,Bm$
is "independent" of those of $C1,\ldots Cp$

Definition of MVDs Continued

Equivalently: the decomposition into

$R1(A1,\ldots,An,B1,\ldots,Bm)$, $R2(A1,\ldots,An,C1,\ldots,Cp)$

is lossless

Note: an MVD $A1,\ldots,An \twoheadrightarrow B1,\ldots,Bm$

Implicitly talks about "the other" attributes $C1,\ldots Cp$

Rules for MVDs

If $A1,\ldots An \longrightarrow B1,\ldots,Bm$

then $A1,\ldots,An \twoheadrightarrow B1,\ldots,Bm$

Other rules in the book

4^{th} Normal Form (4NF)

R is in 4NF if whenever:

$$A1,\ldots,An \twoheadrightarrow B1,\ldots,Bm$$

is a nontrivial MVD,

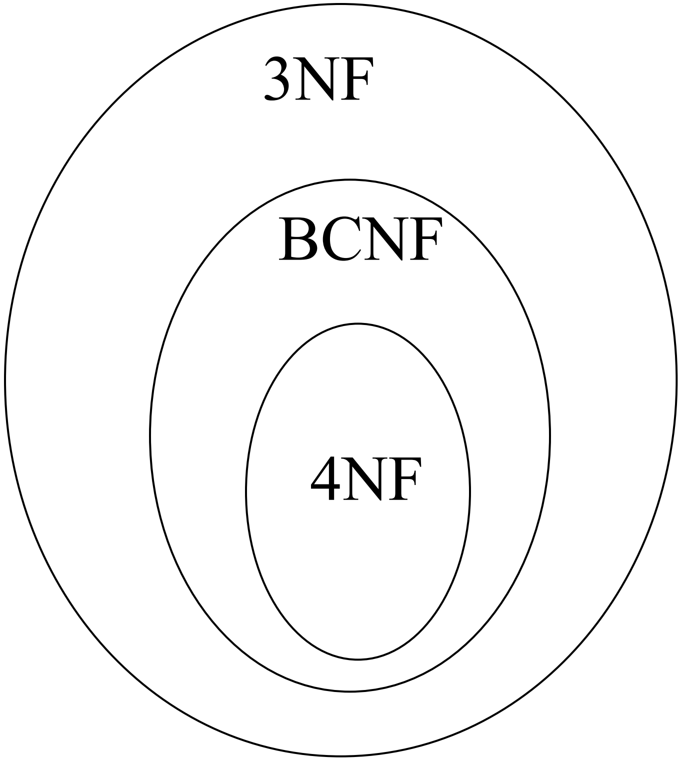
then $A1,\ldots,An$ is a superkey

Same as BCNF with FDs replaced by MVDs

Multivalued Dependencies (MVDs)

- $X \rightarrow \rightarrow Y$ means that given X, there is a unique <u>set</u> of possible Y values (which do not depend on other attributes of the relation)
- PARENTNAME$\rightarrow \rightarrow$CHILDNAME
- An FD is also a MVD
- MVD problems arise if there are two independent 1:N relationships in a relation.

Confused by Normal Forms ?



In practice: (1) 3NF is enough, (2) don't overdo it !

Normal Forms

First Normal Form = all attributes are atomic

Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)

Third Normal Form (3NF)

Fourth Normal Form (4NF)

Others...

Fifth Normal Form

- Sometimes a relation cannot be losslessly decomposed into two relations, but can be into three or more.
- 5NF captures the idea that a relation scheme must have some particular lossless decomposition ("join dependency").
- Finding actual 5NF cases is difficult.

Normalization Summary

- 1NF: usually part of the woodwork
- 2NF: usually skipped
- 3NF: a biggie
  - always aim for this
- BCNF and 4NF: tradeoffs start here
  - in re: d-preserving and losslessness
- 5NF: You can say you've heard of it...

Caveat

- Normalization is not the be-all and end-all of DB design
- Example: suppose attributes A and B are always used together, but normalization theory says they should be in different tables.
  – decomposition might produce unacceptable performance loss (extra disk reads)
- Desirable Property #4: Good query performance
- Plus -- there are constraints other than FDs and MVDs

Current Trends

- Data Warehouses
  - huge historical databases, seldom or never updated after creation
  - joins expensive or impractical
  - argues against normalization
- Everyday relational DBs
  - aim for BCNF, settle for 3NF

Relational Schema Design (or Logical Design)

Conceptual Model:

Relational Model:
- create tables
- specify FD's
- find keys

Normalization
- use FDs to decompose tables to achieve better design