

Normalization

Motivation

- We have designed ER diagram, and translated it into a relational db schema $R = \text{set of } R_1, R_2, \dots$
- Now what?
- We can do the following
 - specify all relevant constraints over R
 - implement R in SQL
 - start using it, making sure the constraints always remain valid
- However, R may not be well-designed, thus causing us a lot of problems

Example of Bad Design

Persons with several phones:

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

Problems (also called "Anomalies"):

Redundancy = repetition of data

update anomalies = update one item and forget others
= inconsistencies

deletion anomalies = delete many items,
delete one item, loose other information

insertion anomalies = can't insert one item without inserting others³

Better Designs Exist

Break the relation into two:

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

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

How do We Obtain a Good Design?

- Start with the original db schema R
- Transform it until we get a good design R^*
- Desirable properties for R^*
 - must preserve the information of R
 - must have minimal amount of redundancy
 - must be dependency-preserving
 - if R is associated with a set of constraints C , then it should be easy to also check C over R^*
 - (must also give good query performance)

OK, But ...

- How do we recognize a good design R^* ?
- How do we transform R into R^* ?
- Answers: use normal forms

Normal Forms

- DB gurus have developed many normal forms
- Most important ones
 - Boyce-Codd, 3rd, and 4th normal forms
- If R^* is in one of these forms, then R^* is guaranteed to achieve certain good properties
 - e.g., if R^* is in Boyce-Codd NF, it is guaranteed to not have certain types of redundancy
- DB gurus have also developed algorithms to transform R into R^* that is in some of these normal forms

Normal Forms (cont.)

- DB gurus have also discussed trade-offs among normal forms
- Thus, all we have to do is
 - learn these forms
 - transform R into R^* in one of these forms
 - carefully evaluate the trade-offs
- Many of these normal forms are defined based on various constraints
 - functional dependencies and keys

Functional Dependencies and Keys

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

$A_1, A_2 \dots A_n$

then they must also agree on the attributes

$B_1, B_2 \dots B_m$

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

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

In General

- To check $A \rightarrow B$, erase all other columns

...	A	...	B	
	X1		Y1	
	X2		Y2	
	

- check if the remaining relation is many-one (called *functional* in mathematics)

Example

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

More examples:

Product: name → price, manufacturer

Person: ssn → name, age

Company: name → stock price, president

More about FDs

- $A \rightarrow B$, we say “A functionally determines B”
- When creating a DB schema, we should list all FDs we believe are valid
- These FDs should be valid on ALL DB database instances conforming to our schema
 - can’t just be valid on one database instance
 - and not valid on another database instance

Relation Keys

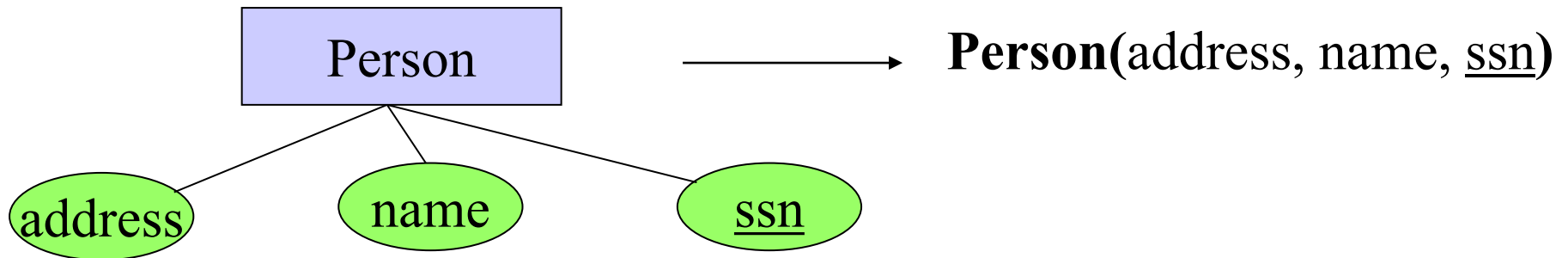
- Key of a relation R is a set of attributes that
 - functionally determines all attributes of R
 - this creates a FD $A \rightarrow B$
 - none of its subsets determines all attributes of R
- Superkey
 - a set of attributes that contains a key
 - so a key is also a superkey

Finding the Keys of a Relation

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

Rules:

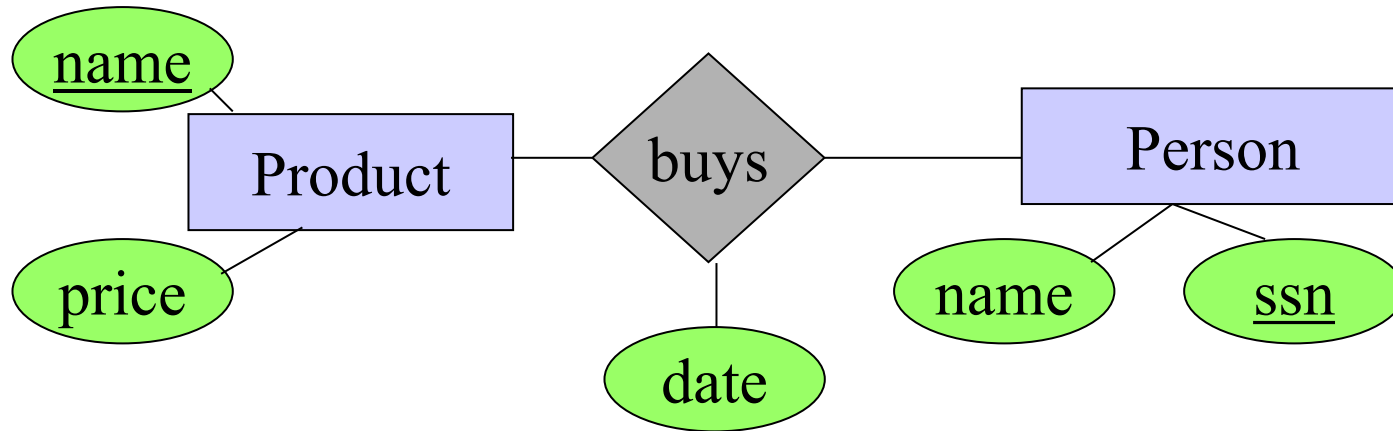
1. If the relation comes from an entity set, the key of the relation is the set of attributes which is the key of the entity set.



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(name, ssn, date)

Finding the Keys

More rules:

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

Note that if you say the set of attributes A is a key, you are basically saying certain FDs are true.

Reasoning with FDs

- 1) closure of FD sets
- 2) closure of attribute sets

Closure of FD sets

- Given a relation schema R & a set S of FDs
 - is the FD f logically implied by S ?
- Example
 - $R = \{A, B, C, G, H, I\}$
 - $S = A \twoheadrightarrow B, A \twoheadrightarrow C, CG \twoheadrightarrow H, CG \twoheadrightarrow I, B \twoheadrightarrow H$
 - would $A \twoheadrightarrow H$ 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
 - $A_1A_2\dots A_n \rightarrow$ a subset of $A_1A_2\dots A_n$
- Augmentation rule
 - $A_1A_2\dots A_n \rightarrow B_1B_2\dots B_m$, then
 $A_1A_2\dots A_n C_1C_2\dots C_k \rightarrow B_1B_2\dots B_m C_1C_2\dots C_k$
- Transitivity rule
 - $A_1A_2\dots A_n \rightarrow B_1B_2\dots B_m$ and
 $B_1B_2\dots B_m \rightarrow C_1C_2\dots C_k$, then
 $A_1A_2\dots A_n \rightarrow C_1C_2\dots C_k$

Inferring S^+ using Armstrong's Axioms

- $S^+ = S$
- Loop
 - for each f in S , apply reflexivity and augment. rules
 - add the new FDs to S^+
 - for each pair of FDs in S , apply the transitivity rule
 - add the new FD to S^+
- Until S^+ does not change any further

Additional Rules

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

Closure of a Set of Attributes

Given a set of attributes $\{A_1, \dots, A_n\}$ and a set of FDs S .

Problem: find all attributes B such that:

any relation which satisfies S also satisfies:

$$A_1, \dots, A_n \rightarrow B$$

The **closure** of $\{A_1, \dots, A_n\}$, denoted $\{A_1, \dots, A_n\}^+$, is the set of all such attributes B

➔ *basically all attributes that are functionally determined by A_1, \dots, A_n*

We will discuss the motivations for attribute closures soon

Algorithm to Compute Closure

Start with $X = \{A_1, \dots, A_n\}$.

Repeat until X doesn't change do:

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

B_1, B_2, \dots, B_n are all in X, **and**

C is not in X

then

add C to X.

Example

A B \longrightarrow C
A D \longrightarrow E
B \longrightarrow D
A F \longrightarrow B

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

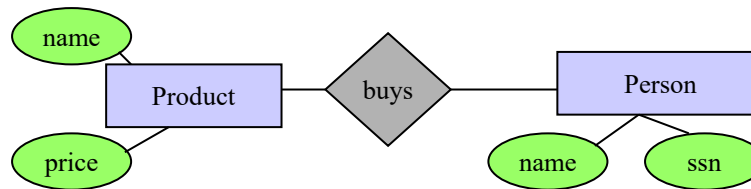
- Test if X is a superkey
 - compute X^+ , and check if X^+ contains all attrs of R
- Check if $X \twoheadrightarrow Y$ holds
 - by checking if Y is contained in X^+
- Another way to compute closure S^+ of FDs
 - for each subset of attributes X in relation R , compute X^+
 - for each subset of attributes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

Desirable Properties of Schema Refinement

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

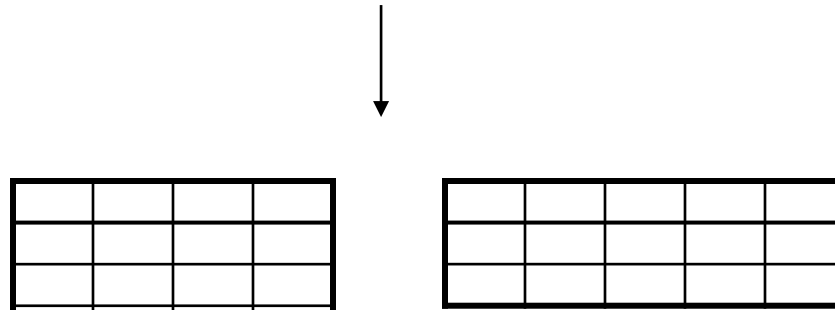
Relational Schema Design (or Logical Design)

Conceptual Model:



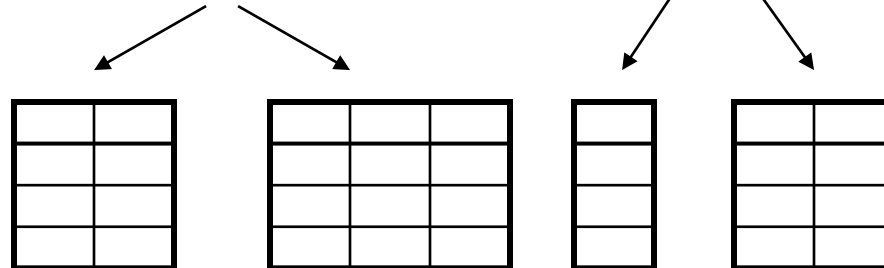
Relational Model:

- create tables
- specify FD's
- find keys



Normalization

- use FDs to
decompose tables
to achieve better design



Recall: Relation Decomposition

The original relation schema

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

Relation Decomposition (cont.)

Break the relation into two:

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

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

- **Desirable Property #1: Minimize redundancy**

Decompositions in General

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

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

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

Such that:

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

And

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

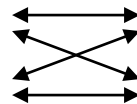
-- $R2$ is the projection of R on $C_1, C_2, \dots C_l$

Certain Decomposition May Cause Problems

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

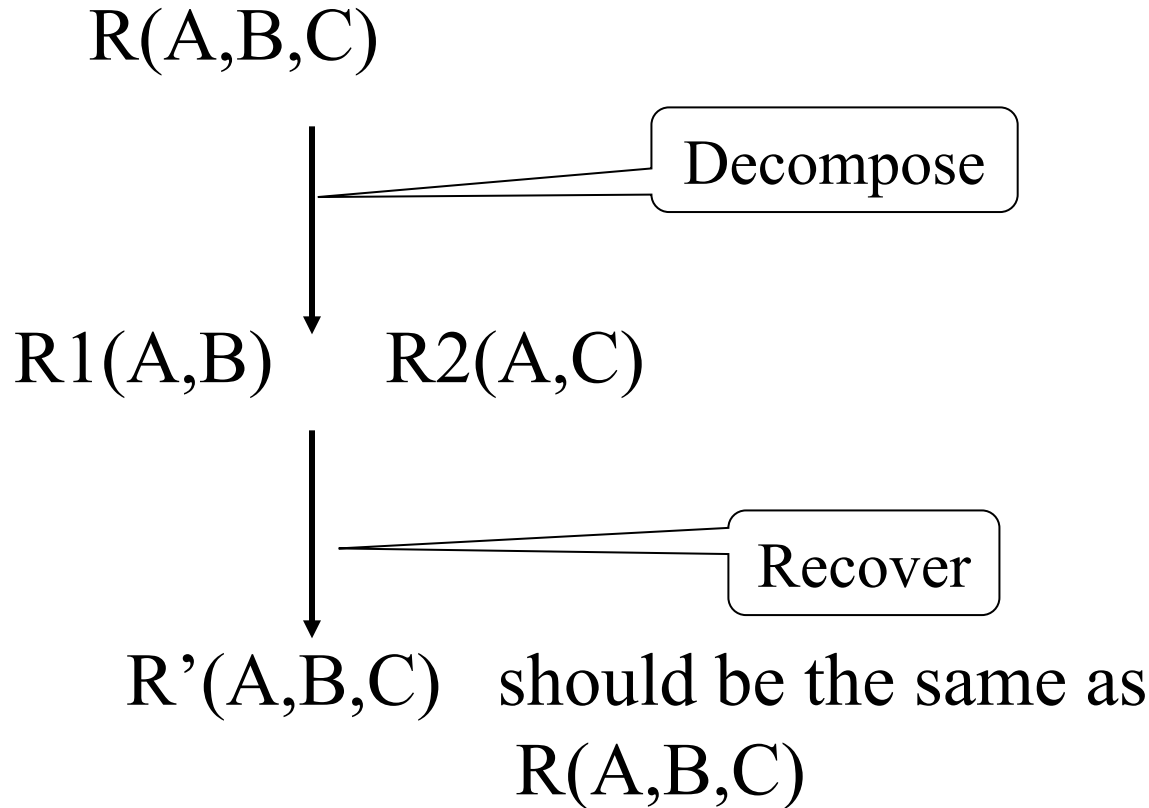
Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
OneClick	29.99	Camera
DoubleClick	24.99	Camera
DoubleClick	29.99	Camera

When we put it back:

Cannot recover information

Lossless Decompositions

A decomposition is *lossless* if we can recover:



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

- Desirable Property #2: Lossless decomposition

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

Dependency Preserving

- Given a relation R and a set of FDs S
- Suppose we decompose R into R_1 and R_2
- Suppose
 - R_1 has a set of FDs S_1
 - R_2 has a set of FDs S_2
 - S_1 and S_2 are computed from S
- We say the decomposition is dependency preserving if by enforcing S_1 over R_1 and S_2 over R_2 , we can enforce S over R

An Example

Unit	Company	Product

FD's: $\text{Unit} \rightarrow \text{Company}$; $\text{Company, Product} \rightarrow \text{Unit}$

So, there is a BCNF violation, and we decompose.

Unit	Company

$\text{Unit} \rightarrow \text{Company}$

Unit	Product

No FDs

So What's the Problem?

Unit	Company	Unit	Product
Galaga99	UI	Galaga99	databases
Bingo	UI	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

- What if, when a relation is decomposed, the X of an $X \rightarrow Y$ ends up only in one of the new relations and the Y ends up only in another?
- 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 to decomposition to
 - minimize redundancy
 - avoid info loss
 - preserve dependencies (i.e., constraints)
 - ensure good query performance
- These objectives can be conflicting
- Various normal forms achieve parts of the objectives

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...

Recall: What We Want to Do with Normal Forms

- Take a relation schema...
- Test it against a normalization criterion...
- If it passes, fine!
 - Maybe test again with a higher criterion
- If it fails, decompose into smaller relations
 - Each of them will pass the test
 - Each can then be tested with a higher criterion

Boyce-Codd Normal Form

A simple condition for removing anomalies from relations:

A relation R is in BCNF if and only if:

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

In English (though a bit vague):

Whenever a set of attributes of R is determining another attribute,
it is a super-key, and thus should determine all attributes of R . (A key
is also a super-key)

Example

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 FDs?

$SSN \rightarrow Name$

Does this FD satisfy the BCNF condition?

Is the relation in BCNF?

Decompose it into BCNF

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

SSN \longrightarrow 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 → Price, Category

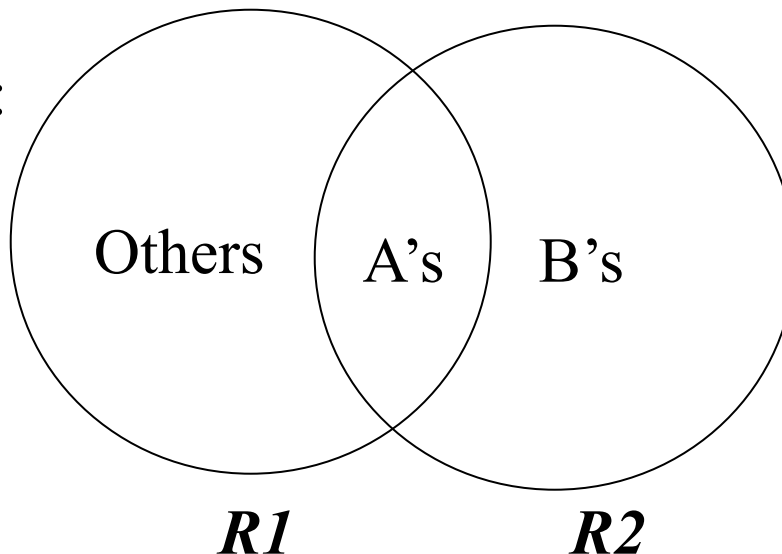
BCNF Decomposition

Find a dependency that violates the BCNF condition:

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

Heuristics: choose $B_1, B_2, \dots B_m$ “as large as possible”

Decompose:



Any
2-attribute
relation is
in BCNF.

Continue until
there are no
BCNF violations
left.

Example Decomposition

Person:

Name	SSN	Age	EyeColor	PhoneNumber

Functional dependencies:

SSN \longrightarrow Name, Age, Eye Color

BCNF: Person1(SSN, Name, Age, EyeColor),
Person2(SSN, PhoneNumber)

What if we also had an attribute Draft-worthy, and the FD:

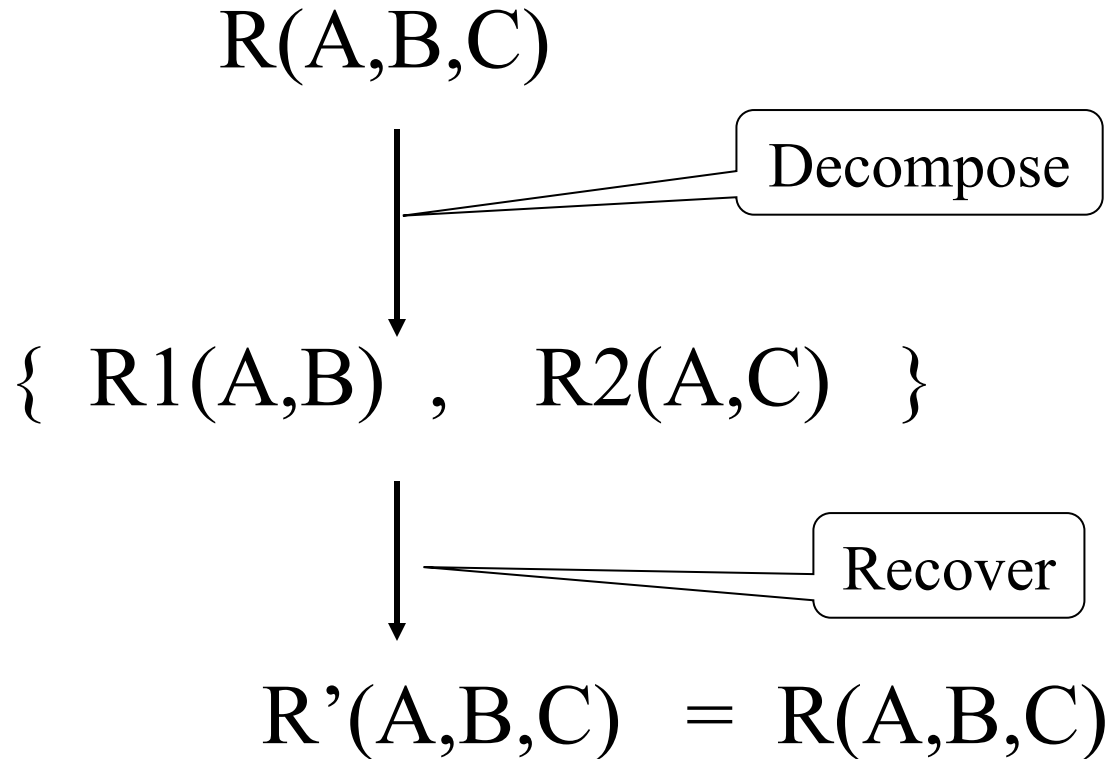
Age \longrightarrow Draft-worthy

Thus,

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

Recall: Lossless Decompositions

A decomposition is *lossless* if we can recover:



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

Decomposition Based on BCNF is Necessarily Lossless

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

BCNF: $R_1(A, B), \quad R_2(A, C)$

Some tuple (a, b, c) in R	(a, b', c') also in R
decomposes into (a, b) in R_1	(a, b') also in R_1
and (a, c) in R_2	(a, c') also in R_2

Recover tuples in R : $(a, b, c), \quad (a, b, c'), (a, b', c), (a, b', c')$ 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
- Can handle this situation using 3NF
- Not covered in this course, see book for examples