Design Theory for Relational Databases: Functional Dependencies and Normalization

Juliana Freire

Some slides adapted from L. Delcambre, R. Ramakrishnan, G. Lindstrom, J. Ullman and Silberschatz, Korth and Sudarshan

Relational Database Design

- Use the Entity Relationship (ER) model to reason about your data---structure and relationships, then translate model into a relational schema (more on this later)
- Specify relational schema directly
  - Like what you do when you design the data structures for a program

Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" collection of relation schemas
- A bad design may lead to
  - Repetition of Information
  - Inability to represent certain information
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints

Design Choices: Small vs. Large Schemas

Which design do you like better? Why?

EMPLOYEE(ENAME, SSN, ADDRESS, PNUMBER)

PROJECT(PNAME, PNUMBER, PMGRSSN)

EMP_PROJ(ENAME, SSN, ADDRESS, PNUMBER, PNAME, PMGRSSN)

An employee can be assigned to at most one project, many employees participate in a project

What's wrong?

EMP(ENAME, SSN, ADDRESS, PNUM, PNAME, PMGRSSN)

The description of the project (the name
and the manager of the project) is repeated
for every employee that works in that department.

Redundancy!

The project is described redundantly.
This leads to update anomalies.

EMP(ENAME, SSN, ADDRESS, PNUM, PNAME, PMGRSSN)

Update Anomalies → Inconsistencies

Insertion anomalies:

if you insert an employee

need to know which department he/she works

need to know the descriptive information for that department

if you want to insert a department, you can't...until there is at least one employe.

Deletion anomalies: if you delete an employee, is that dept. gone? was this the last employee in that dept.?

Modification anomalies: change DNAME, for example, need to change it everywhere!

EMP(ENAME, SSN, ADDRESS, PNUM, PNAME, PMGRSSN)

Solution: Use NULL values

- May make it hard to specify & understand joins
- May make it hard to aggregate (count, sum, …)
- May have different meanings:
  - attribute *does not apply* to this tuple
  - attribute value is *unknown*
  - value is *known but absent* (not yet recorded)
- Make it hard to interpret query answers

•May not store information about a department with no employees

NULL values also cause problems

Design Choices: Small vs. Large Schemas

Which design do you like better? Why?

PROJECT(PNAME, PNUMBER, PMGRSSN)

PROJ_DEPT(PNUMBER,DNUMBER)

PROJECT(PNUMBER, PNAME, PMGRSSN,DNUMBER)

A department can hold many projects, but a project can only belong to one department.

Design Choices: Small vs. Large Schemas

Which design do you like better? Why?

PROJECT(PNAME, PNUMBER, PMGRSSN

PROJ_DEPT(PNUMBER,DNUMBER)

No redundancy: PNUMBER is key for both relations

PROJECT(PNUMBER, PNAME, PMGRSSN,DNUMBER)

What if I need to create a project before deciding which department will manage it?

A department can hold many proj

project can only belong to one de

Design Choices: Small vs. Large Schemas

Which design do you like better? Why?

EMPLOYEE(ENAME, SSN)

EMP_INFO(ENAME, STARTDATE, ADDRESS,PHONE)

EMP(ENAME, SSN, STARTDATE,ADDRESS, PHONE)

Loss of Information in Decomposition

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	9085837689
John	321-54-9876	1975-03-05	5057894567

ENAME	SSN
John	123-45-6789
John	321-54-9876

ENAME	START_DATE	PHONE
John	1999-05-27	9085837689
John	1975-03-05	5057894567

⋈

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	9085837689
John	123-45-6789	1975-03-05	5057894567
John	**321-54-9876**	**1975-03-05**	**5057894567**
John	**321-54-9876**	**1999-05-27**	**9085837689**

Functional Dependencies (FDs) and Database Design

- A FD is yet another kind of integrity constraint
- Vital for the redesign of database schemas to *eliminate* redundancy
  - Enable systematic improvement of database designs
- A functional dependency (FD) on relation R is a statement of the form:

$$A1, A2, \ldots, An \rightarrow B$$

If two tuples of R agree on attributes A1, A2, …, An, then they must also agree on some other attribute B

FD: Example

EMP(ENAME, SSN, STARTDATE,ADDRESS, PHONE)

SSN → ENAME

SSN → STARTDATE

SSN → ADDRESS

SSN → PHONE

Given a SSN, we expect there is a unique employee

- Shorthand:

  SSN → ENAME, STARTDATE, ADDRESS, PHONE

- Do the following FDs hold?

  ENAME → SSN

  PHONE → SSN

FDs: More Examples

- Examples of functional dependencies:

employee-number → employee-name

course-number → course-title

movieTitle, movieYear → length filmType studioName

- Examples that are NOT functional dependencies

employee-name → employee-number **X**

two distinct employees can have the same name

course-number → book **X**

a course may use multiple books

course-number → car-color **X**

????

What is functional in a functional dependency?

A1,…,An → B

A FD is a *function* that takes a list of values (A1,…,An) and produces a unique value B or no value at all (this value can be the NULL value)

x	f(x)
1	2
2	5
4	5

x	g(x)
1	2
2	2
3	5

x	h(x)
1	10
2	20
3	30

We are looking for functional relationships
(that **must** occur in a relation) among attribute values

What is functional in a functional dependency?

A1,…,An → B

There is a function that takes a li

and produces a unique value B c

this value can be the NUL

Unlike *mathematical* functions, you cannot *compute* from first principles – you need to do it by *looking up in a table*

x	f(x)
1	2
2	5
4	5

x	g(x)
1	2
2	2
3	5

x	h(x)
1	10
2	20
3	30

We are looking for functional relationships
(that **must** occur in a relation) among attribute values

FDs and Database Instances

- Which FDs hold for the following table?

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	9085837689
Mary	321-54-9876	1975-03-05	5057894567
Melissa	987-65-4321	1985-03-05	5057899999

FDs and Database Instances

PHONE → SSN?

- Which FDs hold for the following table?

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	
Mary	321-54-9876	1975-03-05	
Melissa	987-65-4321	1985-03-05	

FDs, like any constraint, tell us about the *schema*

- What about for this other table?

ENAME	SSN	START_DATE	PHONE
John	123-45-6789	1999-05-27	9085837689
Mary	321-54-9876	1975-03-05	9085837689
Melissa	987-65-4321	1985-03-05	5057899999

Functional Dependencies

- Let *R* be a relation schema, $A \subseteq R$ *and* $B \subseteq R$. The functional dependency

  $$A \rightarrow B$$

holds on *R* if and only if for any *legal relations r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributes A, they also agree on the attributes *B.* That is,

$$t_1[A] = t_2[A] \Rightarrow t_1[B] = t_2[B]$$

- If the value of the first attribute(s), A, is known, then the value of the second attribute, B, is known (i.e., determined)
- If a relation *R* is legal under a set *F* of FDs, we say *R satisfies F, or that F holds in R*
- Generalization of the notion of a key

Keys and FDs

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

What does it mean to be a key?

The key attributes uniquely identify the tuple.

For one value of the key, there is only one value for all the other attributes.

There is an FD from the key to every other attribute in the table.

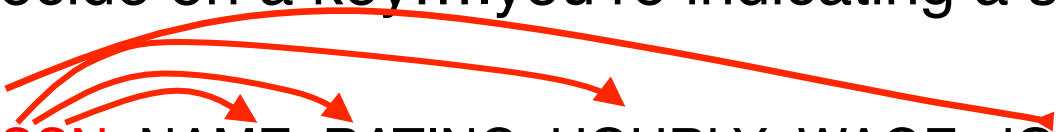
SSN → NAME, RATING, HOURLY_WAGE, JOB_DESC

Keys and FDs (cont.)

What does it mean to be a key?

The key attributes uniquely identify the tuple.

When you decide on a key….you're indicating a set of FDs



EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

But an FD is not necessarily a key,

e.g., *Rating → Hourly_Wage, but Rating is not a key*

SSN	Name	Rating	Hourly_Wage	Job_Desc
123	John	1	10	programmer
234	John	2	20	manager
456	Mary	1	10	QA

Minimality of Keys

- While we suppose designers will keep their design *simple,* we have no way of knowing whether a key is minimal
- FDs allow us to reason about the minimality of keys!
- Minimal: you can't throw anything out
- Minimum: the smallest possible
  - E.g., {city,state} and {zipcode} are both (minimal) keys, while {zipcode} is minimum

Minimality of Keys: Example

Movies(title, year, length, filmType, studioName, starName)

Is {title, year, starName} a key for Movies?

Yes, they functionally determine all other attributes.

Is {title, year, starName} a *minimal* key for Movies?

Need to check all proper subsets:

{title, year} is not a key—title and year do not functionally determine starName (a movie can have multiple stars!)

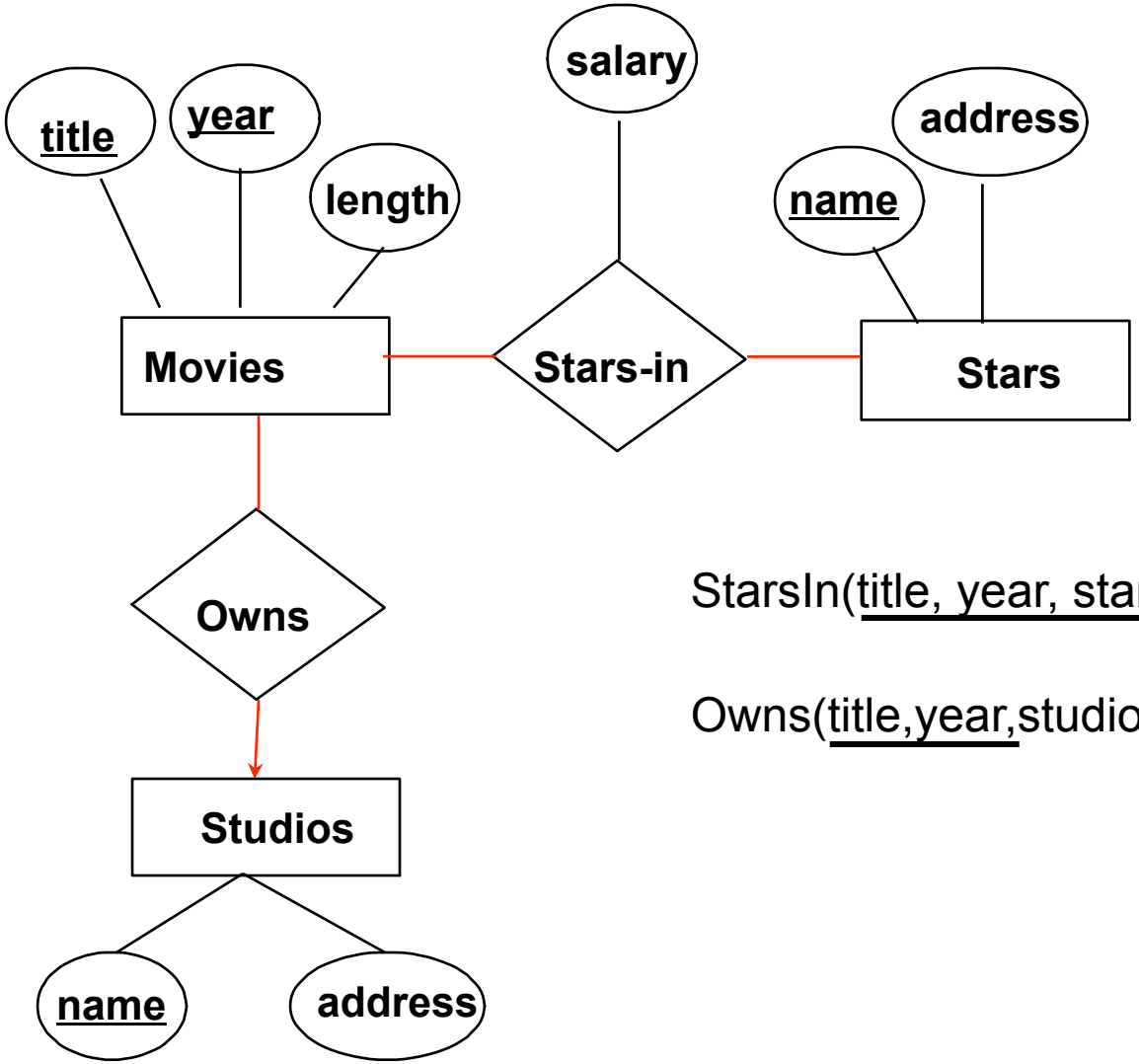
{year, starName} is not a key--there can be a star in 2 movies in the same year

{title, starName} is not a key--two movies with the same title in different years may have a star in common

Discovering Keys for Relations

- Entities --> Relations
  - Key of entity becomes key of corresponding relation (make sure it is minimal!)
- Relationships --> Relations
  - A---n-R-n---B: key(R) = key(A) U key(B)
  - A---n-R-1---B: key(R) = key(A)
  - A---1-R-1---B: key(R) = key(A); or key(R) = key(B)

Discovering Keys for Relations: Example



StarsIn(title, year, starName,salary)

Owns(title,year,studioName)

FDs and Redundancy

- Functional dependencies allow us to express constraints that cannot be expressed using keys

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

rating → hourly_wages

Redundant storage of rating-wage associations

- *Having formal techniques to identify the problem with this design and guide us to a better design is very useful!*

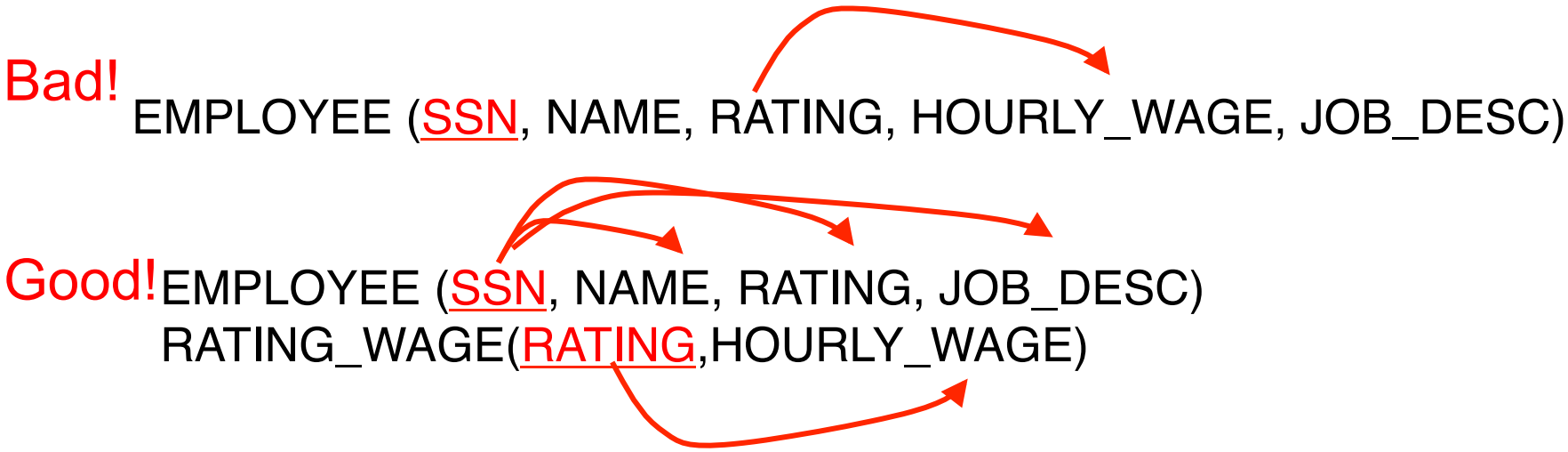
How can FDs help? *

- They help remove redundancy by identifying parts into which a relation can be decomposed
  - *E.g.,* *rating → hourly_wages*

    ssn → name, job_desc

Bad! EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

Good! EMPLOYEE (SSN, NAME, RATING, JOB_DESC)
RATING_WAGE(RATING,HOURLY_WAGE)



Goal: All FDs implied by candidate keys *

If all FDs are “implied by the candidate keys”

The DBMS only needs to enforces keys (it enforces keys anyway)

FDs would be automatically enforced without any extra work on the part of the DBMS

One of our goals: have ALL FDs implied by the candidate keys. No other (non-trivial FDs).

Bad! EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

Good! EMPLOYEE (SSN, NAME, RATING, JOB_DESC)
RATING_WAGE(RATING,HOURLY_WAGE)

Functional Dependencies and Normalization

- FDs are the basis of normalization -- a *formal* methodology for refining and creating *good* relational designs
- Normalization places some constraints on the schema to:
  - Reduce redundancy
  - Alleviate update anomalies
  - Reduce the pressure to have null values
- Normalization puts relations in good form!
- Normalization is a solved problem *(all algorithms & proofs are worked out)*

Schema Refinement Techniques

- Decomposition starts with a relational schema, and uses the FDs to guide the schema decomposition, e.g., to replace a relation R (ABCD) with, two relations, say R1(AB) and R2 (BCD) using the project operator.

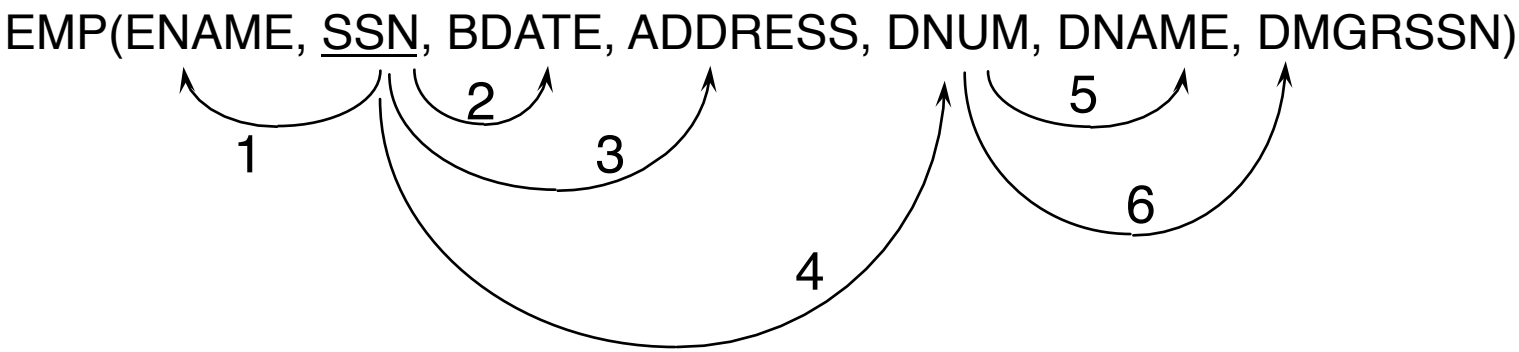
- Synthesis is another refinement technique which takes all attributes over the original relation R, a set of FDs over these attributes, and constructs a *good* schema

What are the FDs?

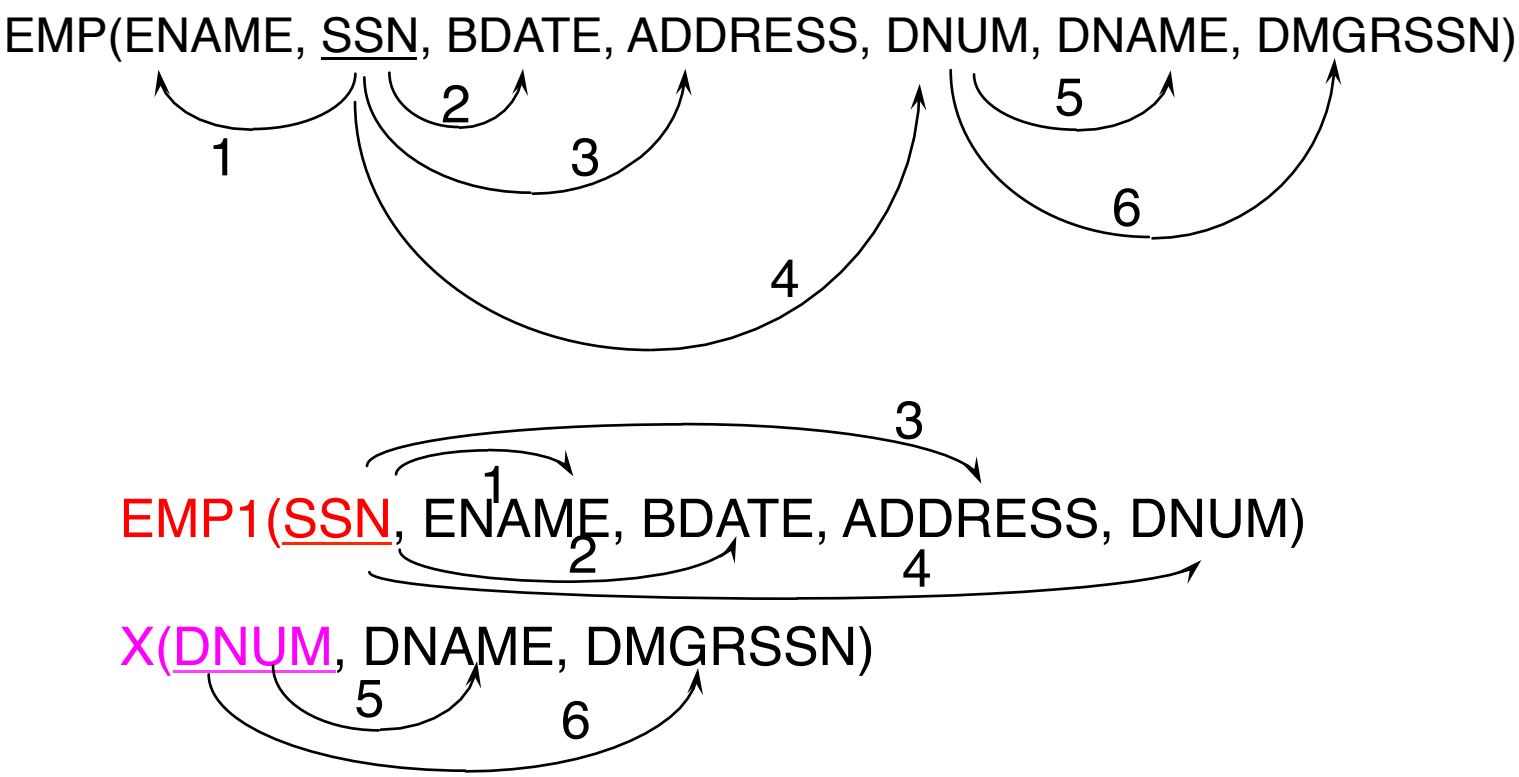
EMP(ENAME, SSN, BDATE, ADDRESS, DNUM, DNAME, DMGRSSN)

EMP_PROJ(SSN, PNUM, HOURS, ENAME, PNAME, PLOCATION)

What are the FDs?

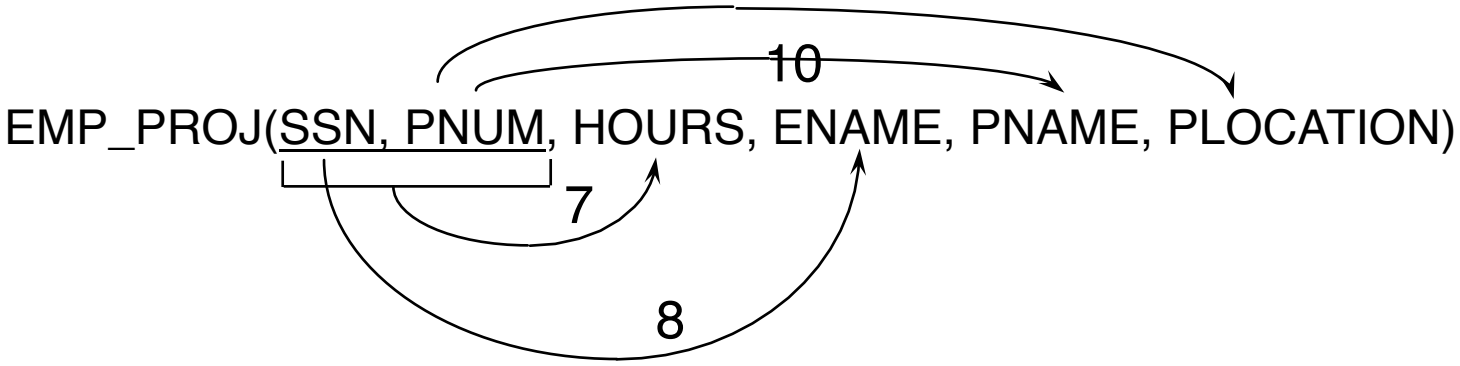


How can we decompose? *

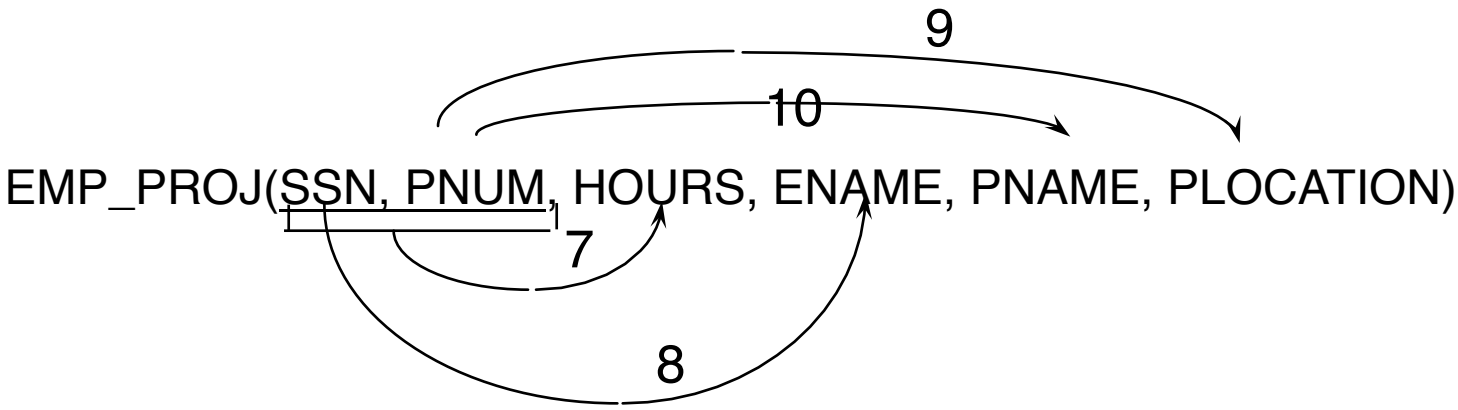


What is a good name for the second table here?

What are the FDs?



How can we decompose?



EMP2(SSN, ENAME)

X(PNUM, PNAME, PLOCATION)

Y(SSN, PNUM, HOURS)

8
9
10
7

How should we name these new tables?

Problems with Decompositions

- There are three potential problems to consider:
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation*!* *(Losslessness)*
  - Checking some dependencies may require joining the instances of the decomposed relations. *(Dependency preservation)*
  - Some queries become more expensive.
    - e.g., In which project does John work? (EMP2 JOIN X)
- *Tradeoff*: Must consider these issues vs. redundancy.

How do we know if a decomposition is correct? That we haven't lost anything?

We have three goals:

Lossless decomposition
(don't throw any information away)
(be able to reconstruct the original relation)

Dependency preservation
all of the non-trivial FDs each end up in just one relation (not split across two or more relations)

Boyce-Codd normal form (BCNF)
no redundancy beyond foreign keys -- all FDs implied by keys

What is a lossless decomposition?

When R is decomposed into R_1 and R_2

If $(R_1 \bowtie R_2) = R$ then decomposition is lossless

if it is a lossy decomposition, then $R_1 \bowtie R_2$ gives you TOO MANY tuples.

Example: A Lossy Decomposition

original

Employee(SS-number, name, project, p-title)

1	smith	p1	accounting
2	jones	p1	accounting
3	smith	p2	billing

decomposition:

Employee (SS-number, name)

1	smith
2	jones
3	smith

Project (project, p-title, name)

p1	account	smith
p1	account	jones
p2	billing	smith

now if we join them with natural join, what happens?

you get at least one extra tuple!!!

1	smith	p2	billing

Test for a Lossless Decomposition

Let R_1 and R_2 form a decomposition of R.
R_1 and R_2 are both sets of attributes from R.

The decomposition is lossless if ...

the attributes in common are a key for *at least one* of the relations R_1 and R_2

$$R_1 \cap R_2 = \text{key}(R_1), \text{ or}$$
$$R_1 \cap R_2 = \text{key}(R_2)$$

Example: testing for a lossless decomposition

Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name)
Project (project, p-title, name)

Which attribute is in common?

name (of the employee)

Is name a key for either of these two tables?

NO! We have a problem.

Example: testing for a lossless decomposition

Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name, project)
Project (project, p-title)

Which attribute is in common?

project

Is project a key for either of these two tables?

Yes!

Example: testing for a lossless decomposition

Employee(SS-number, name, project, p-title)

decomposition: Employee (SS-number, name)
Project (project, p-title)

Which attribute is in common?

None

We have a problem (unless the original Employee relation did not mean to associate an employee with a project).

Testing for a Lossless Join

- If we project R onto R_1, R_2,…, R_k, can we recover R by rejoining?
- Any tuple in R can be recovered from its projected fragments.
- So the only question is: when we rejoin, do we ever get back something we didn't have originally?

The Chase Test

- Suppose tuple t comes back in the join.
- Then t is the join of projections of some tuples of R, one for each R_i of the decomposition.
- Can we use the given FD's to show that one of these tuples must be t ?

The Chase – (2)

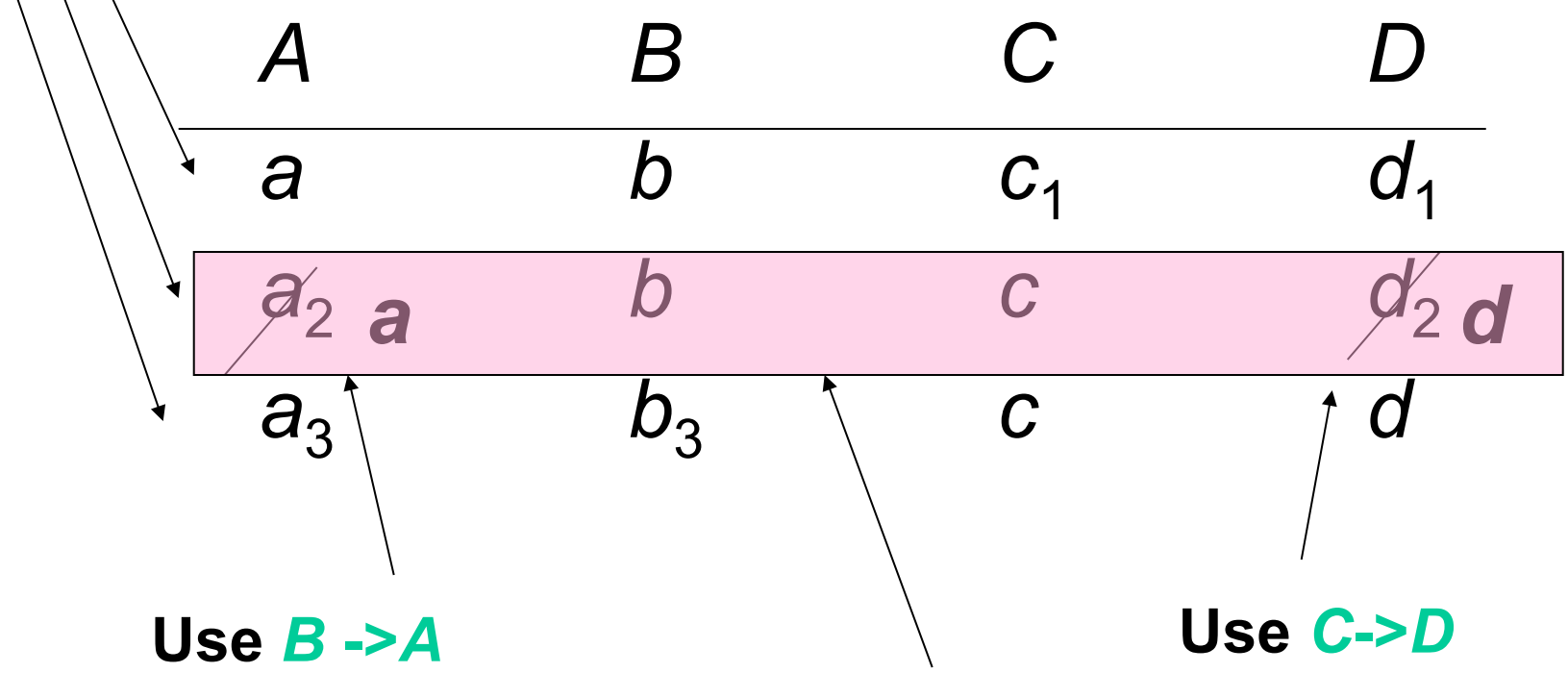
- Start by assuming $t = abc\ldots$.
- For each i, there is a tuple s_i of R that has a, b, c,… in the attributes of R_i.
- s_i can have any values in other attributes.
- We'll use the same letter as in t, but with a subscript, for these components.

Example: The Chase

- Let *R* = *ABCD*, and the decomposition be *AB*, *BC*, and *CD*.
- Let the given FD's be *C->D* and *B ->A*.
- Suppose the tuple t = abcd is the join of tuples projected onto *AB*, *BC*, *CD*.

The tuples of R projected onto AB, BC, CD.

The *Tableau*



We've proved the second tuple must be *t*.

Dependency-Preservation: Example

R = addr(city, state, zip)

FDs = city state → zip, zip → state

Decomposition: R1(zip,state), R2(city,zip)

city state → zip does not hold in R2

Problem: If the DBMS only enforces keys (and not FDs directly), this FD won't be enforced

Testing *city state → zip* requires a join: R1 JOIN R2

Is it possible to guarantee dependency preservation?

More on this later!

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized.
  - This can be used to help us decide whether decomposing the relation will help.

skip

Some nomenclature: Key and Key Attribute

Assign(SSN, PROJECT, S_DATE)

Here the attributes SSN and PROJECT taken together form the *key*. SSN is a *key attribute*; so is PROJECT. S_DATE is a *non-key attribute*.

EMPLOYEE (SSN, NAME, SALARY, JOB_DESC)

Here the attribute SSN is the *key*. SSN is a *key attribute*. All other attributes are *non-key attributes*.

Candidate Key

Employee(SSN, NAME, EID)

If the attribute EID is unique for each employee, then either SSN or EID could be a key. Thus, they are both *candidate keys (minimal!)*. However, only one of them may be designated the *primary key*.

Normalization considers *all* candidate keys of a relation, not just the primary key.

Normal Forms Based on FDs

1NF – all attribute values (domain values) are atomic
(part of the definition of the relational model)

2NF – all non-key attributes must depend on a **whole** candidate key (no partial dependencies)

3NF – table is in 2NF and all non-key attributes must depend on **only** a candidate key (no transitive dependencies)

BCNF – every determinant is a superkey, X --> A, X is a superkey

BCNF >> 3NF >> 2NF >> 1NF

skip

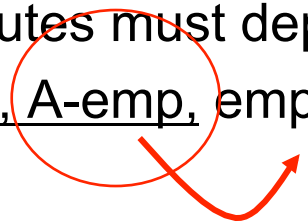
Normalization Made Easy (Ronald Fagin)

Every attribute must depend
upon the key, ← definition of key
the whole key, ← 2NF
and
nothing but the key. ← BCNF (and 3NF)

Examples of Violations

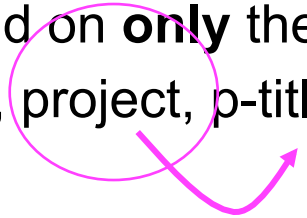
2NF - all *non-key* attributes must depend on the **whole** key

Assigned-to (A-project, A-emp, emp-name, percent)



3NF - all *non-key* attributes must depend on **only** the key

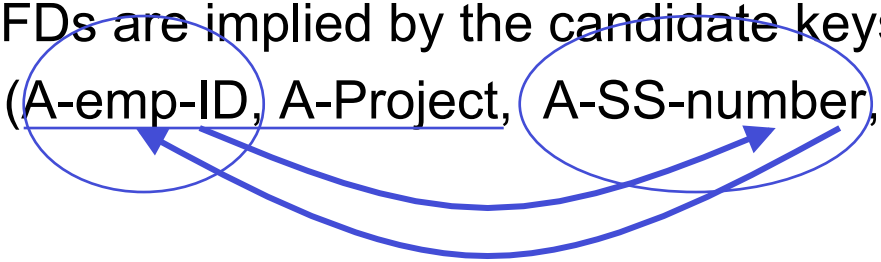
Employee (SS-number, name, address, project, p-title)



BCNF - every determinant is a candidate key

(all FDs are implied by the candidate keys)

Assigned-to (A-emp-ID, A-Project, A-SS-number, percent)



What is the Normal Form of R?

Given: R = { S, B, D, C } (or just SBDC)

Key = { SBD } F = { S→C }

- ?

Key = { SBD, CBD } F = { S→C }

- ?

Key = { SBD } F = { SBD→C }

- ?

What is the Normal Form of R?

2NF - all *non-key* attributes must depend on the **whole** key

3NF - all *non-key* attributes must depend on **only** the key

BCNF - all FDs are implied by the candidate keys

Given: R = { S, B, D, C } (or just SBDC)

Key = { SBD } F = { S→C }

- R violates BCNF – S is not a candidate key
- R violates 3NF – S is not a candidate key, and C is not part of a key
- R violates 2NF (C does not depend on the whole key)

Key = { SBD, CBD } F = { S→C }

- R is in 3NF (C is now a key attribute)
- R violates BCNF because the determinant (S) is not a key (does not matter that C is a key attribute)

Is this in 3NF?

Key = { SBD } F = { SBD→C }

- R is in BCNF because the determinant is a key (the key is the only FD here)

What's the Goal?

BCNF and Lossless and Dependency-Preserving

(first choice)

3NF and Lossless and Dependency-Preserving

(second choice)

because sometimes we can't preserve all dependencies

Reasoning about FDs

- To correctly decompose a relation, we *need to reason about the FDs that hold in the relation*
- Given some FDs, we can usually infer additional FDs:
  - *{ssn → did, did → lot}* implies *{ssn → lot}*
  - *{ssn → name, phone}* implies *{ssn → name, ssn → phone}*
  - *{ssn → name}* implies *{ssn → name, ssn → ssn}*

Trivial dependency

Closure of Attributes

Given a relation R(A1,…,An) and a set of FDs F.

The closure of (A1,…,An) under F, denoted by $(A1,\ldots,An)^+$, is the set of attributes B such that every relation that satisfies F, also satisfies

A1,…,An → B

In other words, A1,…,An → B *follows* from F

Computing Closure of Attributes

- We want to compute $\{A1,\ldots,An\}^+$.

Initialize X = {A1,…,An}

Repeatedly search for some FD B1,B2,…,Bn→C
  such that Bi ∈ X ∀i, and C ∉ X
  X = X U {C}
Until no more attributes can be added to X

Return X

Computing Closure of Attributes: Example

R(A,B,C,D,E,F)
FDs F = {AB→C, BC→AD, D→E, CF→B}

What is the closure of:
{A,B} $^{+}$ = {A, B}
= {A,B,C}
= {A,B,C,D}
= {A,B,C,D,E}

Is AB a key for R?

trivial dependencies
AB→C
BC → D
D → E

Now, we know that AB → CDE
F ==> AB → CDE, or AB → CDE follows from F

Closure of Attributes and of FDs

- If we know how to compute the closure of any set of attributes, we can test if any given FD A1, …,An → B follows from a set of FDs F
  - Compute $\{A1,\ldots,An\}^+$
  - If $B \in \{A1,\ldots,An\}^+$, then A1,…,An → B

Testing Derived FDs: Example

R(A,B,C,D,E,F)
FDs F = {AB→C, BC→AD,D→E, CF→B}

Does AB→D follow from F?

$\{A,B\}^+ = \{A, B\}$
$= \{A,B,C\}$
$= \{A,B,C,D\}$

AB→C
BC → D

Yes! $D \in \{A,B\}^+$

Does D→ A follow from F?

$\{D\}^+ = \{D\}$
$= \{D,E\}$

No! $A \notin \{D\}^+$

Attribute Closure and Keys

- {A1,…,An} is a superkey if and only if

{A1,…,An}$^+$ is the set of all attributes of a relation

- How would you test whether

{A1,…,An}$^+$ is a minimal key?

  - Check if {A1,…,An}$^+$ → all attributes, and then
  - Check that for no X = {A1,…,An} – {Ai}, for each i, X+ → all attributes

Reasoning about FDs

- An FD *f* is *implied by* a set of FDs *S*

  if *f* holds whenever all FDs in *S* hold

- Two sets of FDs S and T are *equivalent*

if the set of relations satisfying S is the same as the set of relations satisfying T

- A set of FDs S *follows* from a set of FDs T

if every relation instance that satisfies all FDs in T also satisfies all the FDs in S

Closure of FDs

- Let F be a set of FDs

F^+ -- the closure of F, is the set of all FDs implied from F

- Closure can be computed by:
  - using a set of *inference rules:* apply rules until no new FDs arise -- until a *fixpoint* is reached
  - Use algorithm for closing a set of attributes

Inference Rules

Armstrong's Axioms

F1. Reflexivity: If Y is a subset of X, then $X \rightarrow Y$

F2. Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$

F3. Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

Others

F4. Split/Decomposition: If $X \rightarrow AB$, then $X \rightarrow A$ and $X \rightarrow B$

F5. Union: If $X \rightarrow A$ and $X \rightarrow B$, then $X \rightarrow AB$

Example: Using Inference Rules

Prove that if $X \rightarrow Y$ and $Z \rightarrow W$, then $XZ \rightarrow YW$

1. $X \rightarrow Y$ (given)
2. $XZ \rightarrow YZ$ (1 and Augmentation)
3. $Z \rightarrow W$ (given)
4. $YZ \rightarrow YW$ (3 and Augmentation)
5. $XZ \rightarrow YW$ (2, 4, and Transitivity)

Is this a correct inference?

- If XY→Z, then X→Z and Y→Z
- title,year → studioName

  title → studioName

  year → studioName

Incorrect!

Computing F+ using Rules

$F^+ = F$

repeat

for each functional dependency f in F^+

apply reflexivity and augmentation rules on f

add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

Computing F+ using Attribute Closure

- Can you suggest an algorithm?
- Recall that:

If we know how to compute the closure of any set of attributes, we can test if any given FD A1, …,An → B follows from a set of FDs F

  - *Compute {A1,…,An}$^+$*
  - *If B ∈ {A1,…,An}$^+$, then A1,…,An → B*

Practice Exercise

- R(A,B,C,D)
- F = AB→C, C→D, D→A
- What are all the non-trivial dependencies that follow from F?

Redundancy in FDs and Minimal Covers

- A → B, B→C, A→C
  - A→C is redundant – it can be inferred from A → B, B→C
- A *minimal cover* (or basis) for a set F of FDs is a set G of FDs s.t.:
  1. Every dependency in G is of the form X→A and A is a single attribute
  2. F+ = G+
  3. If H = (G – X→A), then H+ <> F+

Every dependency is as small as possible, and required for the closure to be equal to F+

Computing Minimal Covers

- Given a set of FDs F:

1. Put FDs in standard form
   - Obtain G of equivalent FDs with single attribute on the right side
2. Minimize left side of each FD
   - For each FD in G, check each attribute on the left side to see if it can be deleted while preserving equivalence to F+
3. Delete redundant FDs
   - Check each remaining FD in G if it can be deleted while preserving equivalence to F+

Solution to Practice Exercise

- R(A,B,C,D)
- F = AB→C, C→D, D→A

Consider the closures of all 15 nonempty sets of attributes.

- For the single attributes we have A+ = A, B+ = B, C+ = ACD, and D + = AD. Thus, the only new dependency we get with a single attribute on the left is C->A.
- Now consider pairs of attributes: AB+ = ABCD, so we get new dependency AB->D. AC+ = ACD, and AC->D is nontrivial. AD+ = AD, so nothing new. BC+ = ABCD, so we get BC->A, and BC->D. BD+ = ABCD, giving us BD->A and BD->C. CD+ = ACD, giving CD->A.
- For the triples of attributes, ACD+ = ACD, but the closures of the other sets are each ABCD. Thus, we get new dependencies ABC->D, ABD->C, and BCD->A.
- Since ABCD+ = ABCD, we get no new dependencies.
- The collection of 11 new dependencies mentioned above is: **C->A, AB->D, AC->D, BC->A, BC->D, BD->A, BD->C, CD->A, ABC->D, ABD->C, and BCD->A.**

Projecting Functional Dependencies

- To check for dependency preservation we need to determine which FDs hold for the decomposed relations—we can do this by *projecting* the dependencies
- Given a relation R with FDs F. Let S = π_A R
- What FDs hold in S?
  - All FDs *f* that follow from F, i.e., $f \in F^+$, that involve only attributes of S

Projecting FDs: Example

R(A,B,C,D)
FDs: A→B, B→C, and C→D

Which FDs hold for S(A,C,D) ?
{A}+ = {A,B,C,D}, thus A→C and A→D hold in S
A→B makes no sense in S!
{B}+ = {B,C,D}
B→anything makes no sense in S!
{C}+ = {C,D}, thus C→D hold in S
{D}+ = {D}, thus D→D hold in S
trivial dependency

Projecting FDs: Example (cont.)

R(A,B,C,D)

FDs: A→B, B→C, and C→D

Which FDs hold for S(A,C,D) ? *A→C, A→D, C→D*

{AC}+ = {A,B,C,D}

{A anything} will add nothing new – everything will *follow* by augmentation

{CD}+ = {C,D}

nothing new

Stop!

Decomposition and Normal Forms

BCNF (reminder)

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form A→ *B,* where A ⊆ *R* and *B* ⊆ *R*, at least one of the following holds:

- A → *B* is *trivial* (i.e., *B* ⊆ A)
- A is a *superkey* for *R*

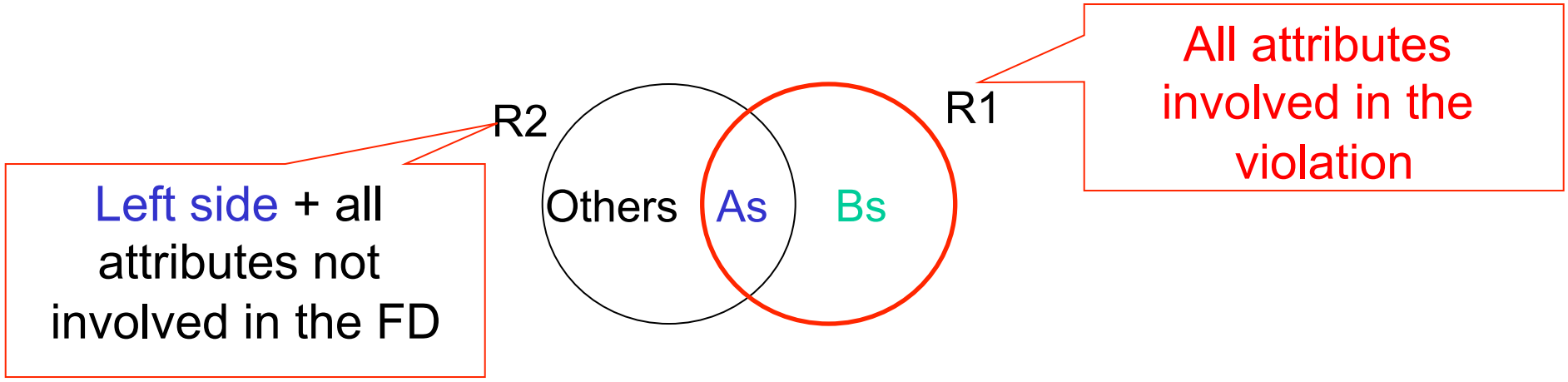
The left side of every nontrivial FD must be a superkey

BCNF: Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF –
  - B→C is non-trivial and B is not a superkey

Decomposition into BCNF

- We can break *any* relation schema R into a collection of subsets Si of its attributes s.t.:
  - Si is in BCNF
  - We don't lose information, i.e., we can *reconstruct* R from Si
- Suppose A1, A2, ..., An → B1,…,Bm violates BCNF. Construct 2 overlapping relations R1 and R2:
  - R1 = A U B
  - R2 = (R - (B-A))



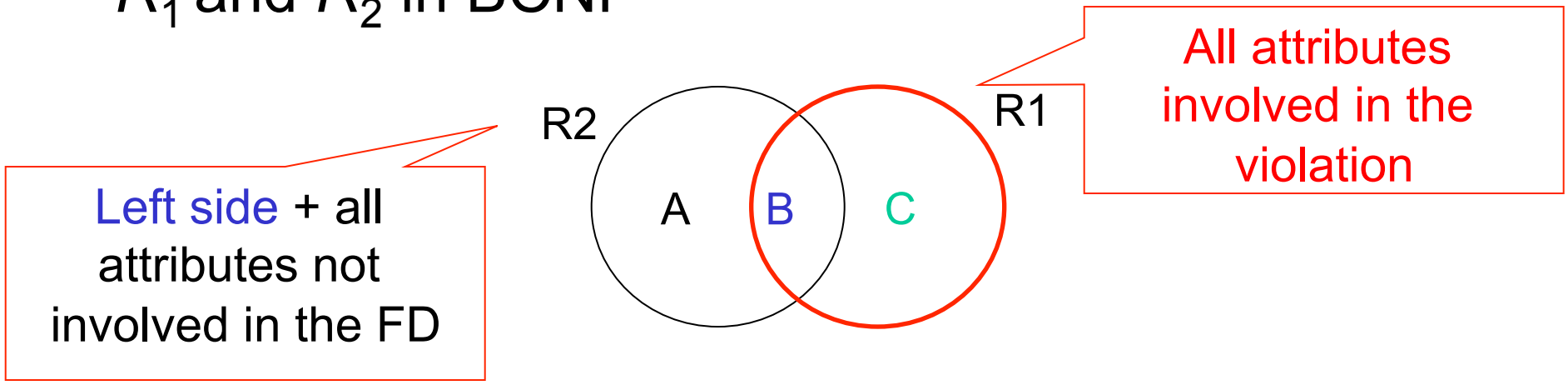
BCNF: Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$ – violates BCNF

Key = $\{A\}$

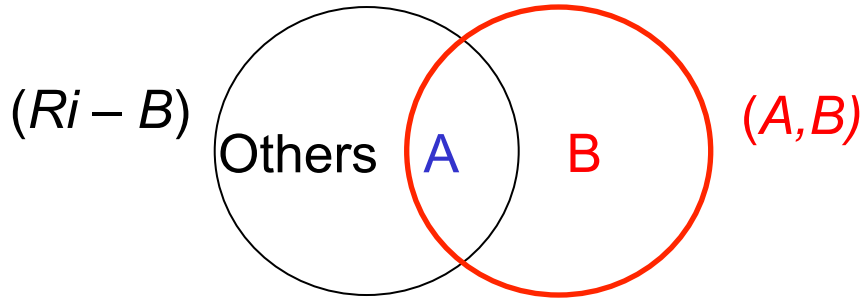
- R is not in BCNF – B is not a superkey
- Decomposition $R_1 = (B, C)$, $R_2 = (A, B)$,

R_1 and R_2 in BCNF



Computing a BCNF Decomposition

result := {*R*};
done := false;
compute F^+;
while (not *done)* **do**



if (there is a schema R_i in *result* that is not in BCNF)
then begin
let $A \rightarrow B$ be a nontrivial functional dependency that holds on R_i such that $A \rightarrow R_i$ is not in F^+, and $A \cap B = \varnothing$;
result := (*result* – R_i) ∪ (R_i – *B*) ∪ (A, *B*);
end
else *done* := **true;**

Testing for BCNF

- To check if a non-trivial dependency A $\rightarrow$ *B* causes a violation of BCNF
  1. Compute A^+
  2. Verify that A is a superkey of *R*.

- To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F*
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

- However, using only F is incorrect when testing a relation in a decomposition of R

Testing for BCNF: Example

- Consider R (A, B, C, D), with $F = \{ A \rightarrow B, B \rightarrow C\}$
- Decompose R into $R_1(A,B)$ and $R_2(A,C,D)$
  - R1 is in BCNF. *Why?*
    - *$A \rightarrow B$ holds in R_1 and A+ = AB*
  - Since neither of the dependencies in F contain only attributes from (A,C,D) , *does this mean R_2 is in BCNF?*
  - $A \rightarrow C$ in F^+ holds in R_2 and A is not a superkey for R_2
  - *R_2 is NOT in BCNF!*

Testing for BCNF: An Easier Check

- *Computing every dependency in F+ can be unnecessary*
- To verify if relation Ri in a decomposition of *R* is in BCNF, *for each subset s* of attributes in Ri, check that
  - Either *s*+ contains no attribute of Ri - *s*, or
  - s+ = Ri, *I.e., s* is a superkey for Ri
- *Try to prove that this test is sound!*

Testing for BCNF: Another Example

$F = \{ A \rightarrow B,\ B \rightarrow C\}$

$R = \{ R_1(A,B),\ R_2(A,C,D)\}$

$s+ \cap (Ri - s) = \varnothing$ *or*
s is a superkey for Ri

- Is R_1 in BCNF?
  - A+ = A**B**C R_1 - A = **B**, but A is a superkey for R_1
  - B+ = BC R_1 - B = A, BC ∩ A = ∅
  - AB+ = AB AB is a superkey

Testing for BCNF: Another Example

$F = \{ A \rightarrow B,\ B \rightarrow C\}$

$R = \{ R_1(A,B),\ R_2(A,C,D)\}$

$s+ \cap (Ri - s) = \varnothing$ or
s is a superkey for Ri

- Is R_2 in BCNF?
  - A+ = AB**C** R_2 - A = **C**D, intersection contains C

    and A is not superkey of R_2

    Immediately discover R_2 is not in BCNF!

- *Which FD violates BCNF?*

  ABC ∩ (ACD – A) = *%% Intersection is not empty!*

  ABC ∩ CD = C

  A → C holds and violates BCNF *%% A is not superkey*

Algorithm for Lossless Join Decomposition into BCNF Relations

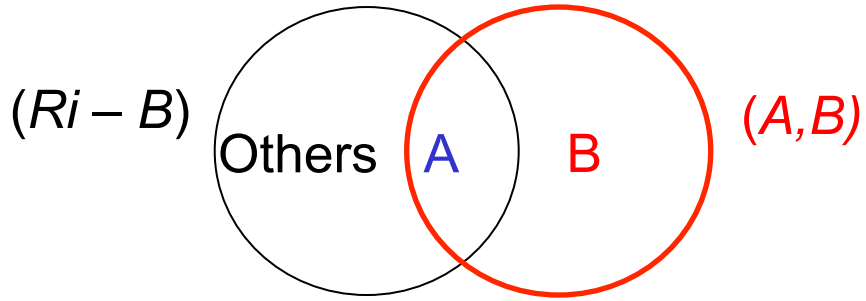
1. set D := { R } (the current set of relations)

2. while there is a relation in R that is not in BCNF
   begin
   choose a relation schema Ri that is not in BCNF
   find an FD X → Y in Ri that violates BCNF
   replace Ri in D by two relations: (Ri - Y) and
   (X U Y)
   end;

3. identify dependencies that are not preserved (X → A).
   add XA as a table to the set D

Computing a BCNF Decomposition

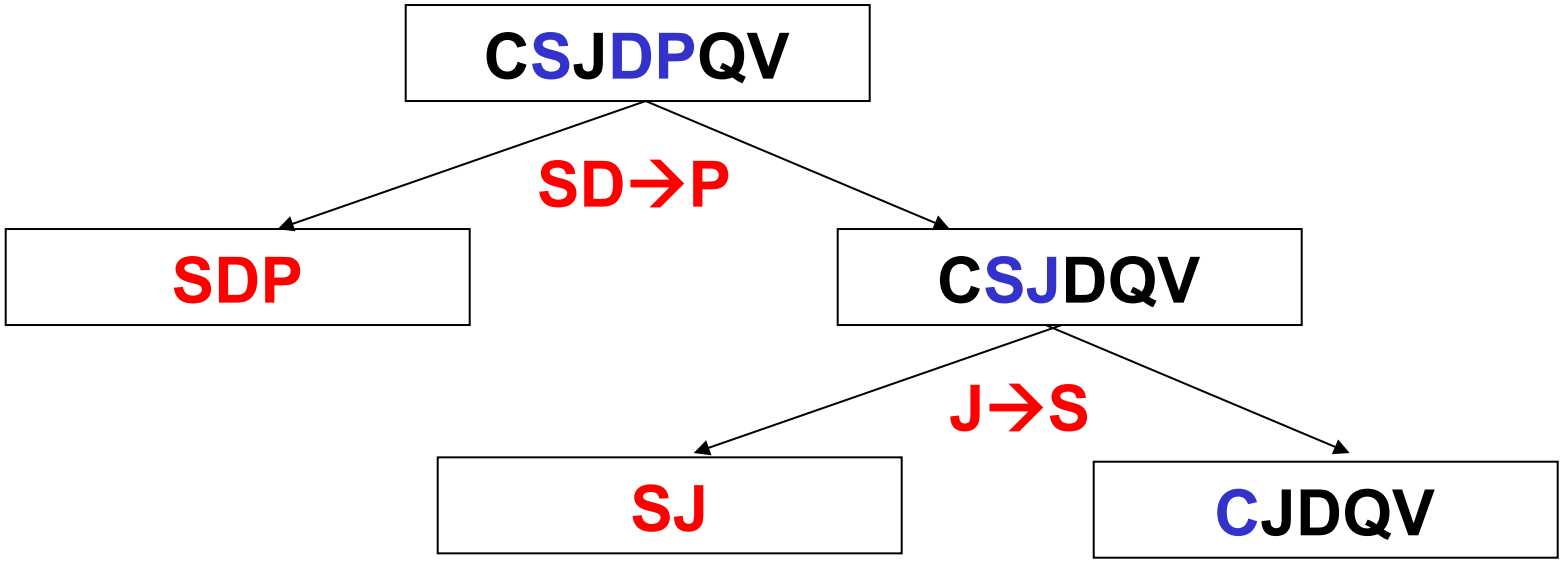
result := {*R*};
done := false;
compute F^+;
while (not *done)* **do**



if (there is a schema R_i in *result* that is not in BCNF)
then begin
let $A \rightarrow B$ be a nontrivial functional dependency that holds on R_i such that $A \rightarrow R_i$ is not in F^+, and $A \cap B = \varnothing$;
result := (*result* – R_i) ∪ (R_i – *B*) ∪ (A, *B*);
end
else *done* := **true;**

Decomposing into BCNF: Example

- R = CSJDPQV
- F = {SD→P; J→S; JP→C}, Key = {C}

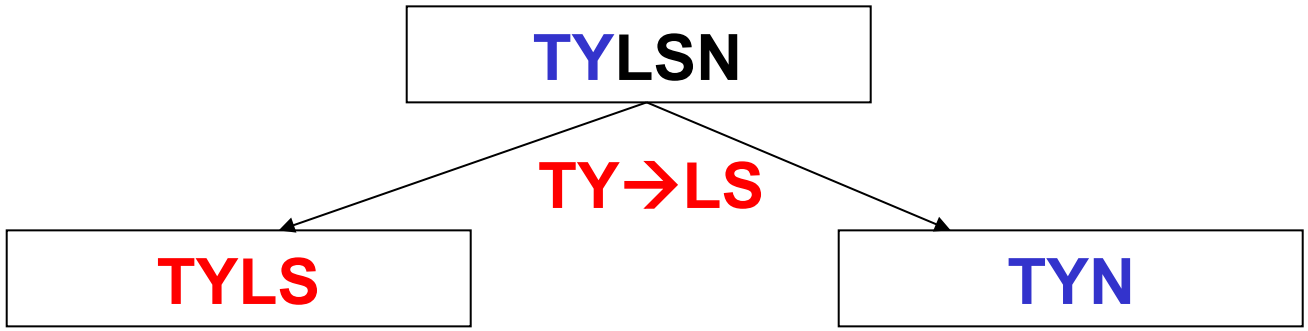


Need to check for BCNF at each step!
Note: any 2-attribute relation is in BCNF (see pg 89 in textbook)

Decomposing into BCNF: Another Example

- Movies = {title,year,length,studio,starName}
- F = {title,year→length,studio},

Key = {title,year,starName}



Movies = {title,year,length,studio}

StarsIn = {title,year,starName}

Decomposition: Desirable Properties

- **Losslessness:** don't throw any information away
  - (R1 ⋈ R2) = R
  - Recall that: a decomposition is lossless if the attributes in common are a key for *at least one* of the relations R_1 and R_2
  - BCNF decomposition is lossless: *Ri is replaced by* ($R_i - B$) and (A, *B), and A → B holds in (A,B)*
- **Dependency preservation:** all of the non-trivial FDs each end up in just one relation
  - Not split across two or more relations
  - Cannot guarantee this for BCNF…

What is “Dependency Preserving”

Suppose F is the original set of FDs and G is set of projected FDs after decomposition

The decomposition is dependency preserving if $F^+ \equiv G^+$. That is, the two closures must be equivalent.

A relation that cannot be decomposed into BCNF in a dependency-preserving manner

R = addr(city, state, zip)

FDs = city state → zip, zip → state

Decomposition: R1(zip,state), R2(city,zip)

city state → zip *does not hold in R2*

Problem: If the DBMS only enforces keys (and not FDs directly), this FD won't be enforced

Testing *city state → zip* requires a join: R1 JOIN R2

Is it possible to guarantee dependency preservation?

An Unenforceable FD

state	zip
Mass	02138
Mass	02139

city	zip
Cambridge	02138
Cambridge	02139

Join tuples with equal zip codes.

state	city	zip
Mass	Cambridge	02138
Mass	Cambridge	02139

Although no FD's were violated in the decomposed relations, FD state city -> zip is violated by the database as a whole.

3NF: The next best to BCNF

- Recall that: A relation schema R is in third normal form (3NF) if for all:

  $A \rightarrow B$ in F^+

  at least one of the following holds:
  - $A \rightarrow B$ is trivial (i.e., $B \subseteq A$)
  - A is a superkey for R

  } BCNF

  - Each attribute t in $B - A$ is contained in a candidate key for R

relaxation of BCNF to ensure dependency preservation

Any relation can be decomposed into 3NF in a dependency-preserving manner

R = addr(city, state, zip)

FDs = city state → zip, zip → state

R is in 3NF

(city state) is a superkey

(state) is contained in a key *%% state-zip*

R preserves all FDs but …

there is some redundancy in this schema

3NF: Example

$R = (J, K, L)$ $FDs = \{JK \rightarrow L, L \rightarrow K\}$

- Two candidate keys: JK and JL
- R is in 3NF

  $JK \rightarrow L$ JK is a superkey

  $L \rightarrow K$ K is contained in a candidate key

- BCNF decomposition has (JL) and (LK)
  - Testing for $JK \rightarrow L$ requires a join!
- There is some redundancy in this schema
- Trade-off: cost (or ability!) to check FD vs. redundancy

3NF and Redundancy

- If X→ A causes a 3NF violation then
  - X is a proper subset of some key K – *partial dependency* (X,A) stored redundantly; or
  - X is not a proper subset of some key K – *transitive dependency* K → X → A
- Examples
  - Partial dependency:

RESERVES(PERSON, ROOM, DATE, CREDIT)

Key = PERSON, ROOM, DATE; PERSON→CREDIT

} BCNF

  - Transitive dependency:

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)

SSN → RATING → HOURLY_WAGE

3NF: Example

$R = (J, K, L)$ $FDs = \{JK \rightarrow L, L \rightarrow K\}$

- Two candidate keys: JK and JL
- R is in 3NF

  $JK \rightarrow L$ JK is a superkey

  $L \rightarrow K$ K is contained in a candidate key

- BCNF decomposition has (JL) and (LK)
  - Testing for $JK \rightarrow L$ requires a join!
- There is some redundancy in this schema
- Trade-off: cost (or ability!) to check FD vs. redundancy

Decomposition into 3NF

- Apply BCNF decomposition
- Identify the set N of dependencies that is not preserved
- For each X→A in N, add relation XA to schema

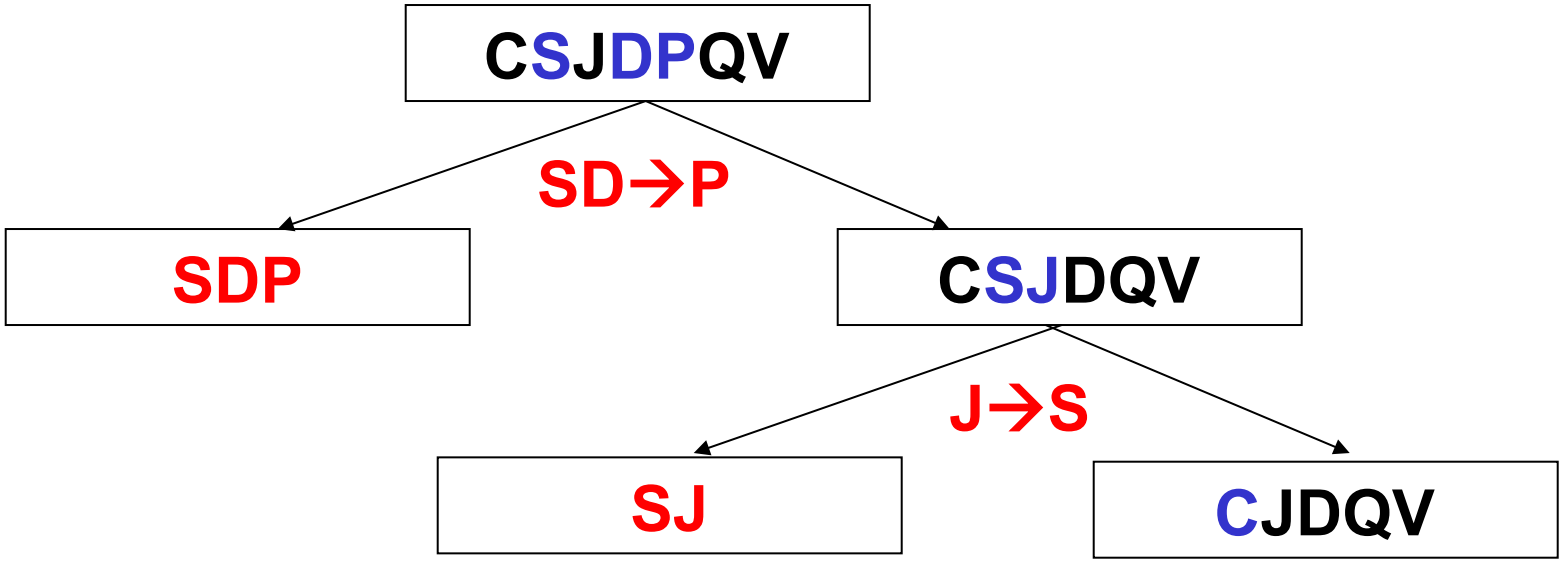
- Note: need to work with the *minimal basis* of the functional dependencies!

Dependency-Preserving Decomposition into 3NF: Example

- R = CSJDPQV
- F = {SD→P; J→S; JP→C}, Key = {C}
- Decomposition: SDP, SJ, CJDQV,CJP
- Note that each relation in decomposition is in BCNF

Dependency-Preserving Decomposition into 3NF: Example

- R = CSJDPQV
- F = {SD→P; J→S; JP→C}, Key = {C}



JP→C is not preserved

Since F is a minimal cover, add CJP to the schema

Minimal Covers and Efficiency

- When relation is updated, DBMS must check whether the update violates any FD
- Can reduce the effort spent checking for violations, by testing a smaller (but equivalent!) set of dependencies

Summary: BCNF and 3NF

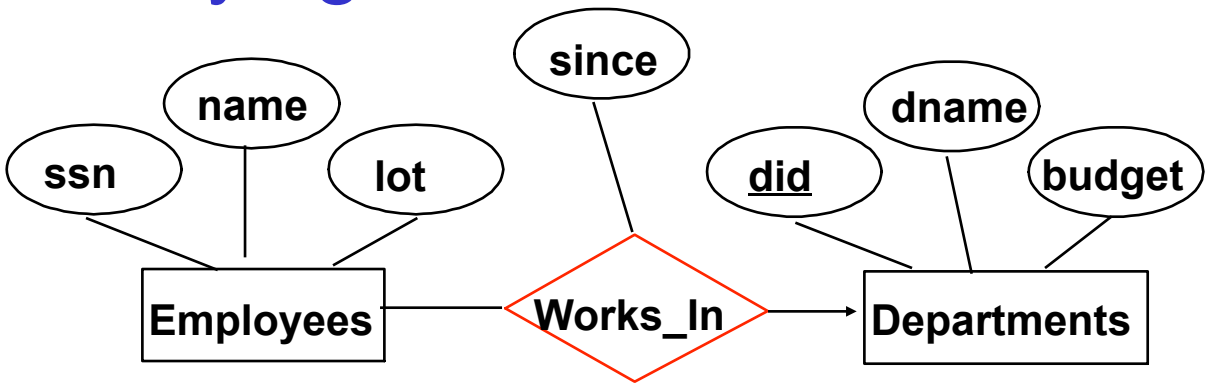
- It is always possible to decompose a relation into relations in 3NF and
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into relations in BCNF and
  - the decomposition is lossless
  - it may not be possible to preserve dependencies
- Trade-off: cost (or ability!) to check FDs vs. redundancy

Constraints on an Entity Set

EMPLOYEE (SSN, NAME, RATING, HOURLY_WAGE, JOB_DESC)
RATING → HOURLY_WAGE

- This FD leads to redundant storage, but…
- It cannot be expressed in the ER model -- only keys can be expressed in the ER model
- Having formal techniques to identify the problem with this design is very useful
  - Especially for large schemas – schemas with more than 100 tables are not uncommon!

Identifying Attributes of Entities



Workers(ssn,name,lot,did,since)

Departments(did,dname,budget)

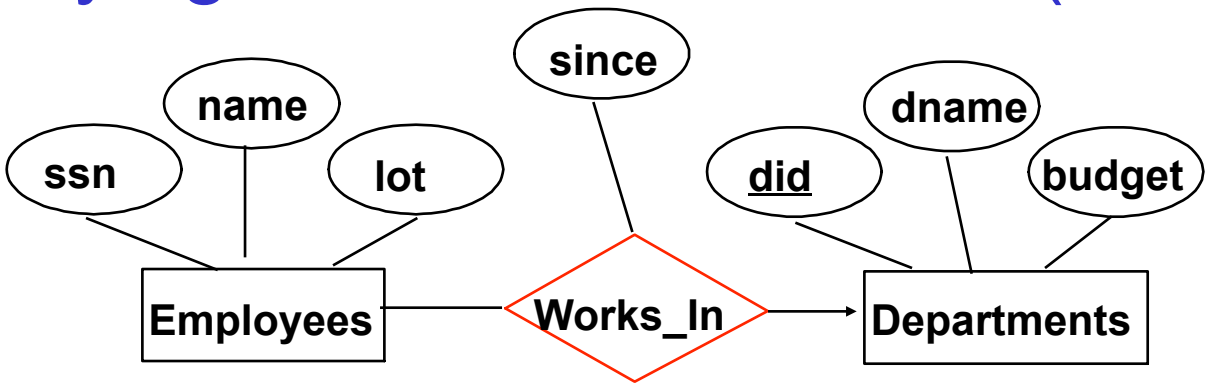
Constraint: Employees are assigned parking lots based on their department, and all employees in a given dept are assigned to the same lot

did → lot not expressible in ER!

Workers(ssn,name,did,since)

DeptLots(did,lot)

Identifying Attributes of Entities (cont.)



Departments(<u>did</u>,dname,budget)

Workers(<u>ssn</u>,name,did,since)

DeptLots(<u>did</u>,lot)

- Can associate a lot with a dept, even if dept has no employees
- Can add employee to dept even if there is no lot assigned to dept

Departments(<u>did</u>,dname,budget,lot)

Redundancy for Performance

- Schema refinement fragments a relation
- Queries may require *joining* the fragments – this can be expensive
  - E.g., every time an account is accessed, need to display name of the customer with account information, account JOIN depositor
- Denormalize for performance
  - Store a relation account JOIN depositor
- A better alternative •– *why?*
  - CREATE VIEW ACC_DEP AS

    SELECT * FROM account, depositor

    WHERE account.id = depositor.acc_id

Decomposition (reminder)

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing R_1 **join** R_2)

Some Known Algorithms

- Compute F^+
- Compute attribute closure
- Find a minimal cover for a set of FDs
- Dependency-preserving decomposition into 3NF
- Lossless join decomposition into BCNF
- Lossless join & dependency preserving decomposition into 3NF

Computing Attribute Closure

- Given a set of attributes α, define the *closure* of α under *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F:*

  $\alpha \rightarrow \beta$ is in F^+ $\rightarrowtail$ $\beta \subseteq \alpha^+$

- Algorithm to compute α^+, the closure of α under *F*

```
result := α;
while (changes to result) do
        for each β → γ in F do
                begin
                        if β ⊆ result then  result := result ∪ γ
                end
```

Example

- *R = (branch-name, branch-city, assets, customer-name, loan-number, amount)*

  F = {branch-name → assets branch-city

  loan-number → amount branch-name}

  Key = *{loan-number, customer-name}*

- Decomposition
  - R_1 = (*branch-name, branch-city, assets)*
  - R_2 = *(branch-name, customer-name, loan-number, amount)*
  - R_3 = *(branch-name, loan-number, amount)*
  - R_4 = *(customer-name, loan-number)*
- Final decomposition

  R_1, R_3, R_4