SE240: Introduction to Database Systems

Lecture 07:

Functional Dependencies and Normalization

Outline

- **Problems Caused by Redundancy**
- Functional Dependencies (FDs)
- Normal Forms Based on Keys
- Decomposition
  - Lossless Join Decomposition
  - Dependency Preserving Decomposition
- Normalization
  - BCNF Decomposition Algorithm
  - 3NF Decomposition Algorithm

Problems Caused by Redundancy

- *Waste of Resources of Storage*
- Potential Inconsistency

Student-ID	Student-Name	Course-ID	Course-Name
123	Sio-Long	COMP231	Database
123	Sio-Long	COMP170	D. Math
567	Mary	COMP104	C++
567	Mary	COMP271	Algorithm
999	Peter	COMP231	Database

Redundancy!
Waste of resources

Problems Caused by Redundancy

- Waste of Resources of Storage
- *Potential Inconsistency*

Student-ID	Student-Name	Course-ID	Course-Name
123	Sio-Long	COMP231	Database
123	Sio-Long LO	COMP170	D. Math
567	Mary	COMP104	C++
567	Mary	COMP271	Algorithm
999	Peter	COMP231	Database

Inconsistency!

Problems Caused by Redundancy

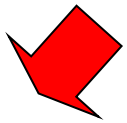
- *Redundant Storage* - Some information is stored repeatedly
- *Update Anomalies* - If one copy of such repeated data is updated, an inconsistency is created, unless all copies are similarly updated
- *Insertion Anomalies* - It may not be possible to store certain information unless some other, unrelated, information is stored
- *Deletion Anomalies* - It may not be possible to delete certain information without losing some other, unrelated, information

Decompositions

- Intuitively, *redundancy* arise when a relational schema forces an association between attributes that is not natural
- *Functional dependencies* can be used to identify such situations and suggest refinements to the schema
- The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of "smaller" relations
- A decomposition of a relation schema R consists of replacing the relation schema by two (or more) relation schemas so that each contains a subset of attributes of R and together include all attributes in R

Decompositions

Student-ID	Student-Name	Course-ID	Course-Name
123	Sio-Long	COMP231	Database
123	Sio-Long	COMP170	D. Math
567	Mary	COMP104	C++
567	Mary	COMP271	Algorithm
999	Peter	COMP231	Database



Student-ID	Student-Name
123	Sio-Long
567	Mary
999	Peter

No Potential Inconsistency

Student-ID	Course-ID
123	COMP231
123	COMP170
567	COMP104
567	COMP271
999	COMP231

Course-ID	Course-Name
COMP231	Database
COMP170	D. Math
COMP104	C++
COMP271	Algorithm

Outline

- Problems Caused by Redundancy
- **Functional Dependencies (FDs)**
- Normal Forms Based on Keys
- Decomposition
  - Lossless Join Decomposition
  - Dependency Preserving Decomposition
- Normalization
  - BCNF Decomposition Algorithm
  - 3NF Decomposition Algorithm

Functional Dependencies

- A functional dependency (FD) is a kind of Integrity Constraint that generalizes the concept of a key

Let R be a relation schema and let $X \subset R$ and $Y \subset R$ be nonempty sets of attributes in R. An instance (relation) r of R satisfies the FD $X \rightarrow Y$. If the following holds for every pair of tuples t_1 and t_2 in r

If $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$

The notation $t_1.X$ refers to the projection of tuple t_1 onto the attributes in X

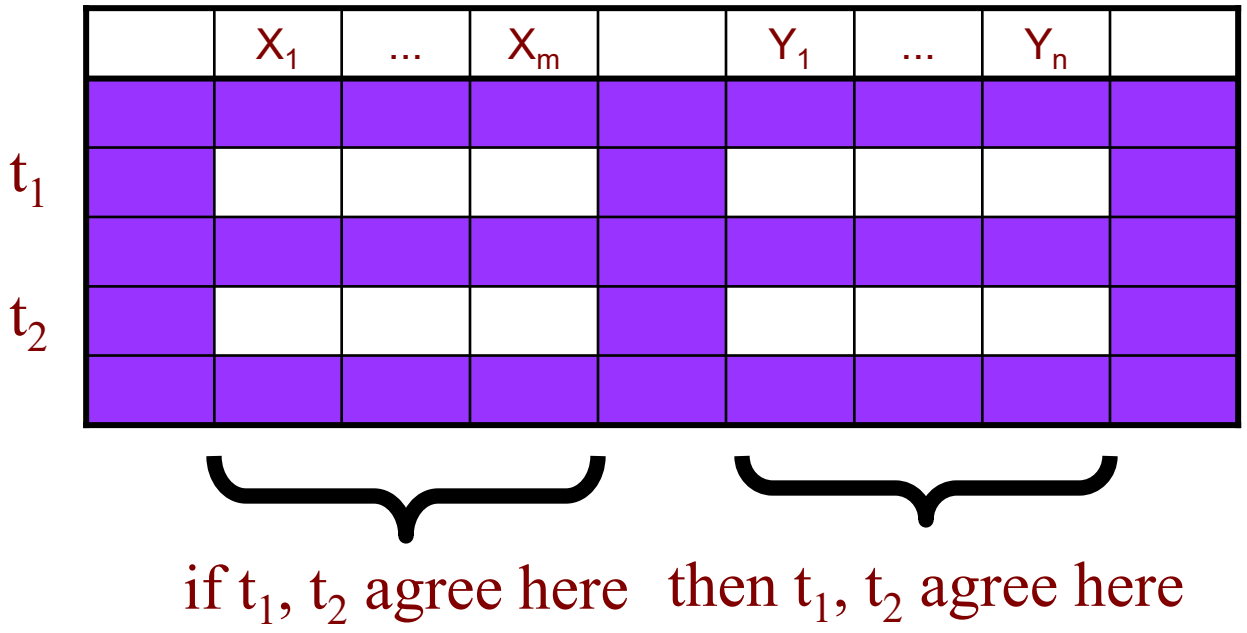
$X \rightarrow Y$ iif, for any two tuples t_1, t_2 of any relation r on R

$$\Pi_X(t_1) = \Pi_X(t_2) \Rightarrow \Pi_Y(t_1) = \Pi_Y(t_2)$$

- An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

Functional Dependencies

- To check $X \rightarrow Y$, erase all other columns; for all rows t_1, t_2



- if $X \rightarrow Y$ is a well-defined function thus, functional dependency

Example: Functional Dependencies

Can you find a functional dependency that satisfy the following relation?

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4

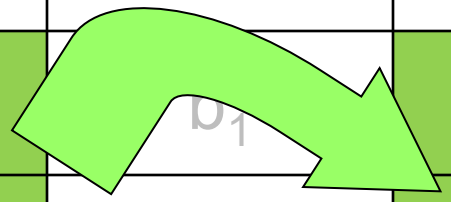
Example: Functional Dependencies

$A \rightarrow C$ is satisfied

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4

Example: Functional Dependencies

$A \rightarrow C$ is satisfied

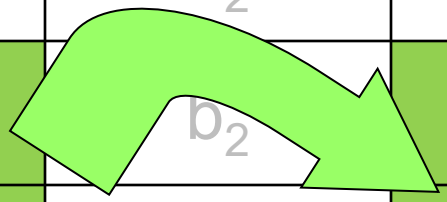


	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4

Example: Functional Dependencies

$A \rightarrow C$ is satisfied

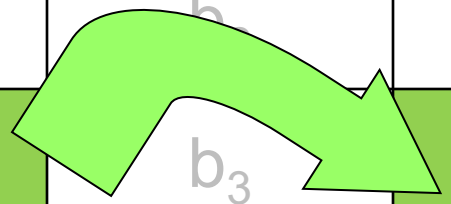
	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4



Example: Functional Dependencies

$A \rightarrow C$ is satisfied

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_3	c_2	d_3
5	a_3	b_3	c_2	d_4



Example: Functional Dependencies

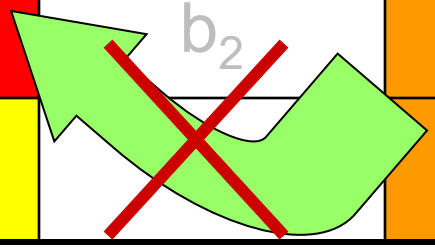
Is $C \rightarrow A$ satisfied?

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4

Example: Functional Dependencies

Is C → A satisfied? ***>>> NO! <<<***

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3		c_2	d_4



Example: Functional Dependencies

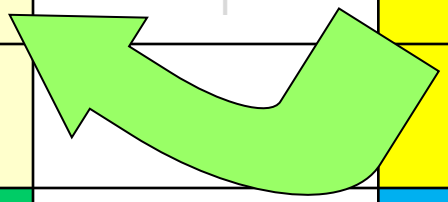
Can you find another functional dependency that satisfies the following relation?

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2	c_2	d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4

Example: Functional Dependencies

D → B is also satisfied

	A	B	C	D
1	a_1	b_1	c_1	d_1
2	a_1	b_2	c_1	d_2
3	a_2	b_2		d_2
4	a_2	b_2	c_2	d_3
5	a_3	b_3	c_2	d_4



Trivial and Non-Trivial FD

- If any two rows *never* agree on the X value, then

  $X \rightarrow Y$ is *Trivially Preserved*

- *Trivial Functional Dependency*
  - $X \rightarrow Y$ where $Y \subseteq X$
  - e.g, AB → A

    Combine set A and B into a new set, denoted by AB
- Non-Trivial Functional Dependency
  - e.g. AB → C

Example: Trivial and Non-Trivial FD

Student-ID	Student-Name	Course-ID	Course-Name
123	Sio-Long	COMP231	Database
567	Mary	COMP231	Database
999	Peter	COMP271	Algorithm
123	Sio-Long	COMP271	Algorithm

- Course-ID → Course-Name (not trivially preserved)
- Student-ID, Course-ID → Course-Name (trivially preserved)
  - All (student_ID, course_ID) value pairs are unique
- Student-Name → Student-Name (trivial dependency)
- Student-Name, Course-Name → Student-Name (trivial dependency)
- Student-ID, Course-ID →

  Student-ID, Student-Name, Course-ID, Course-Name

  (not trivial dependency, but trivially preserved)

Information Deduced from FDs

if & only if

- α is a superkey for R iff $\alpha \rightarrow R$
  - where R is taken as the schema for relation R
- α is a candidate key for R iff

  has the speciality of being a set with minimum size
  - $\alpha \rightarrow R$, and
  - for no γ that is a proper subset of α, $\gamma \rightarrow R$

Example

- $AC \rightarrow R$, $AD \rightarrow R$, $ACD \rightarrow R$

  $R(A,B,C,D) \Leftrightarrow R=ABCD$

  (A, B, C, D are also sets containing data)

A	B	C	D
a_1	b_2	c_1	d_2
a_2	b_2	c_1	d_2
a_1	b_4	c_3	d_3
a_4	b_4	c_3	d_4

Example

A	B	C	D
a_1	b_2	c_1	d_2
a_2	b_2	c_1	d_2
a_1	b_4	c_3	d_3
a_4	b_4	c_3	d_4

- $AC \rightarrow R$, $AD \rightarrow R$, $ACD \rightarrow R$
- AC is a candidate key

Example

A	B	C	D
a_1	b_2	c_1	d_2
a_2	b_2	c_1	d_2
a_1	b_4	c_3	d_3
a_4	b_4	c_3	d_4

- $AC \rightarrow R$, $AD \rightarrow R$, $ACD \rightarrow R$
- AC is a candidate key
- AD is *also* a candidate key

Example

A	B	C	D
a_1	b_2	c_1	d_2
a_2	b_2	c_1	d_2
a_1	b_4	c_3	d_3
a_4	b_4	c_3	d_4

- $AC \rightarrow R$, $AD \rightarrow R$, $ACD \rightarrow R$
- AC is a candidate key
- AD is *also* a candidate key
- but, ACD is *not* a candidate key

Since $AC, AD \subseteq ACD$, and they have been candidate keys.

Note

- A functional dependency must be identified based on semantics of application, not on individual relation instances
  - A→C is a functional dependency only if A→C is satisfied for ***all*** relations for a particular schema
- Given an instance *r* of *R*, we can check if *r* violates some functional dependency *f*, but we cannot tell if *f* holds over *R*

Student-ID	Student-Name	Course-ID	Course-Name
123	Sio-Long	COMP231	Database
567	Mary	COMP271	Algorithm
999	Peter	COMP291	C++
567	John	COMP301	Computer Vision

Course-ID → Student-Name? – Not make sense

Functional Dependencies

- The semantics of an application is a set of constraints imposed by the application
- This set of constraints are set up either explicitly or implicitly
  - Explicitly: by domain experts
  - Implicitly: by ‘common sense’

Example: Functional Dependencies

- Application is to keep track of information about employees in a company
- Information to be kept track of includes:
  - eid: employee's id number
  - ename: employee name
  - address: address of the employee
  - sex: employee's sex
  - dname name of the department that the employee works for
  - dhname: department head's name
  - dhsex: department head's sex

Example: Functional Dependencies

- Let's construct a relation schema as follows:

Employee	eid	ename	address	sex	dname	dhname	dhsex

head name

- Which of the following dependencies are true?

1. eid → ename (T) ✓
   Same name → different ids
2. ename → eid (F) ✗
3. eid → dhname (T) ✓
4. eid → dhsex (T) ✓
5. sex → address (F) ✗
6. dname → dhname (T) ✓
7. dhname → address (F) ✗
8. dhname → dname (F)
   Same name → different departments

Given

a: Employee's id number is unique

b: Each employee works for only one dept

c: Each department has only one head

Implicit: common sense

a: eid → eid,ename,address,sex

b: eid → dname

c: dname → dhname

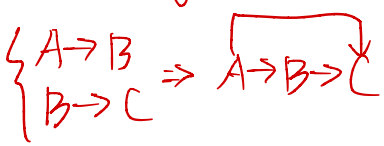
Implicit: dhname → dhsex

Reasoning about FDs

- F - a set of functional dependencies
- f - an individual functional dependency

  f is implied by F if
  - whenever all functional dependencies in F are true

    then f is true
  - e.g, F = {A → B, B → C} implies A → C



Closure of F

- F – { f_1, f_2, …, f_i, …, f_n }, where f_1, f_2, …, f_i, …, f_n are functional dependencies
- f – an individual functional dependency
- F *implies* f, if whenever all f_1, …, f_n are true, f is true
  - e.g. F = {A →B, B→C}, F implies A→C (by transitivity)
- F^+ – *closure* of F: the set of all functional dependencies that F implies

  A→B ⇒ F^+ = {A→A, A→B, B→B, AB→AB, AB→A, AB→B, A→AB}

  augmentation

  - e.g. F = { A →B }, F^+ = { A→A, B→B, AB→AB, A→B, A→AB, AB→A, AB→B }

  the combination of A and B

Closure of a set of FDs

- The set of all FDs implied by a given set F of FDs is called the closure of F, denoted as F^+
- Armstrong's Axioms, can be applied repeatedly to infer all FDs implied by a set of FDs

Suppose X,Y, and Z are sets of attributes over a relation.

Armstrong's Axioms

- Reflexivity: if $Y \subseteq X$, then $X \rightarrow Y$
  e.g., $\{A, C\} \subseteq \{A, B, C\} \Rightarrow ABC \rightarrow AC$
- Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$
  e.g., $A \rightarrow B \Rightarrow CA \rightarrow CB$
- Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
  e.g., $A \rightarrow BD, BD \rightarrow C \Rightarrow A \rightarrow C$

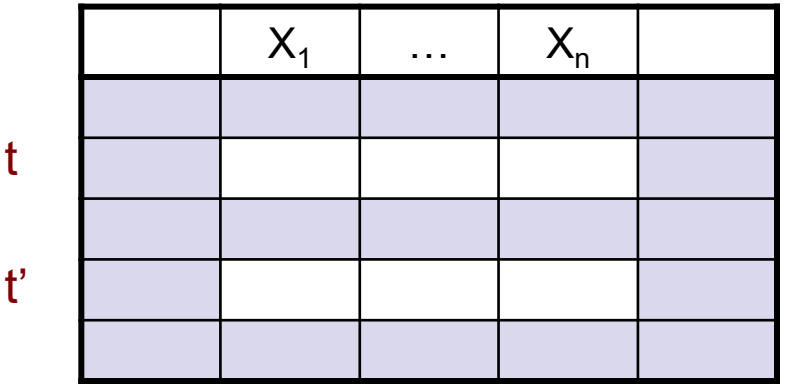
Armstrong's Axioms

- Armstrong's Axioms is sound and complete.
  - Sound: they generate only FDs in F^+
  - Complete: repeated application of these rules will generate all FDs in F^+
- The proof of soundness is straight forward, but completeness is harder to prove

Reflexive Rule: Trivial FDs

$X_1, X_2, \ldots, X_n \rightarrow X_i$

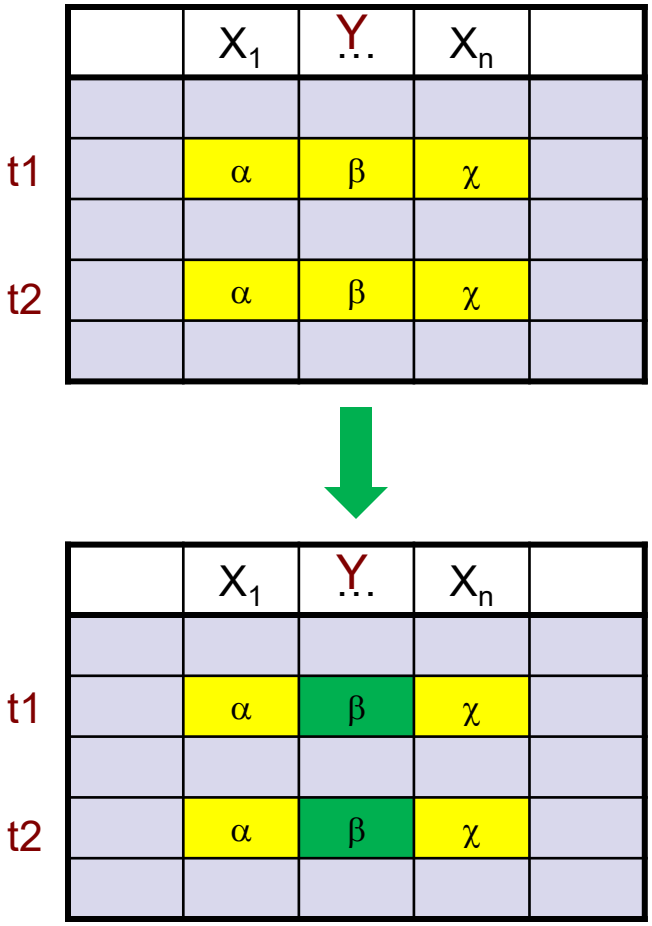
with *i* in 1..*n* is a *trivial FD*



- FD $X_1X_2\ldots X_n \rightarrow Y_1Y_2\ldots Y_k$ may be
  - **Trivial**: Ys are a subset of Xs
  - **Nontrivial**: >=1 of the Ys is not among the Xs
  - **Completely nontrivial**: none of the Ys is among the Xs
- Trivial elimination rule:
  - Eliminate common attributes from Ys, to get an equivalent completely nontrivial FD

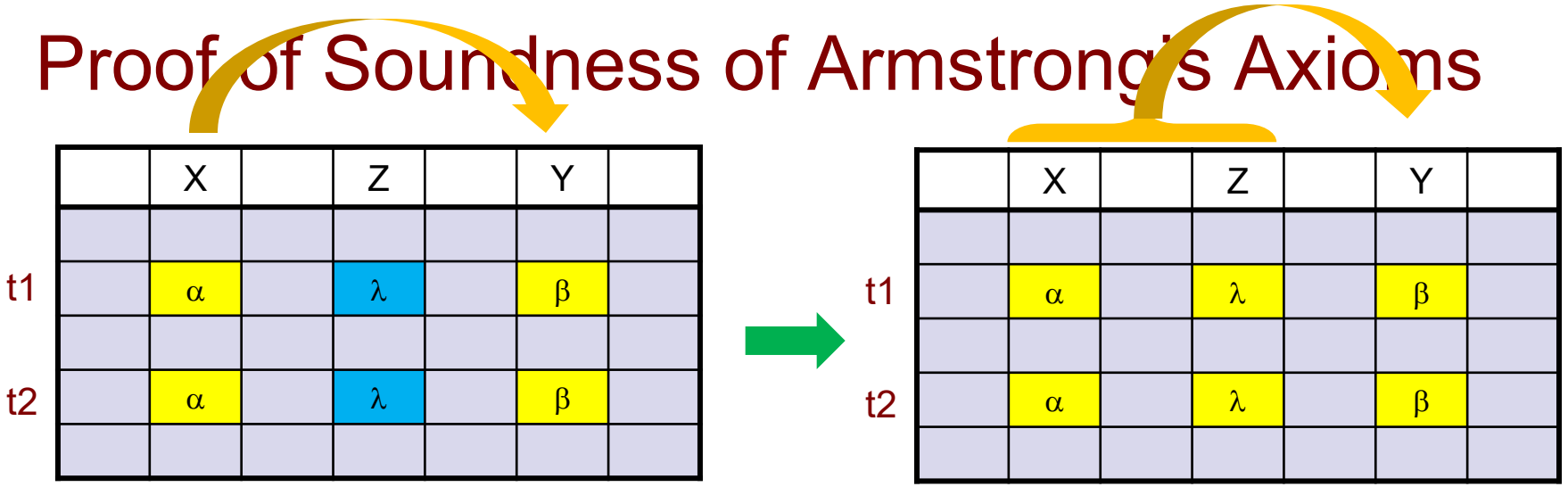
Proof of Soundness of Armstrong's Axioms

- Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$
- Assume $\exists t_1, t_2$ such that $t_1.X = t_2.X$
- Then $t_1.Y = t_2.Y$ since $Y \subseteq X$
- Hence $X \rightarrow Y$



Recall X→ Y : for every pair t_1 and t_2 if $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$

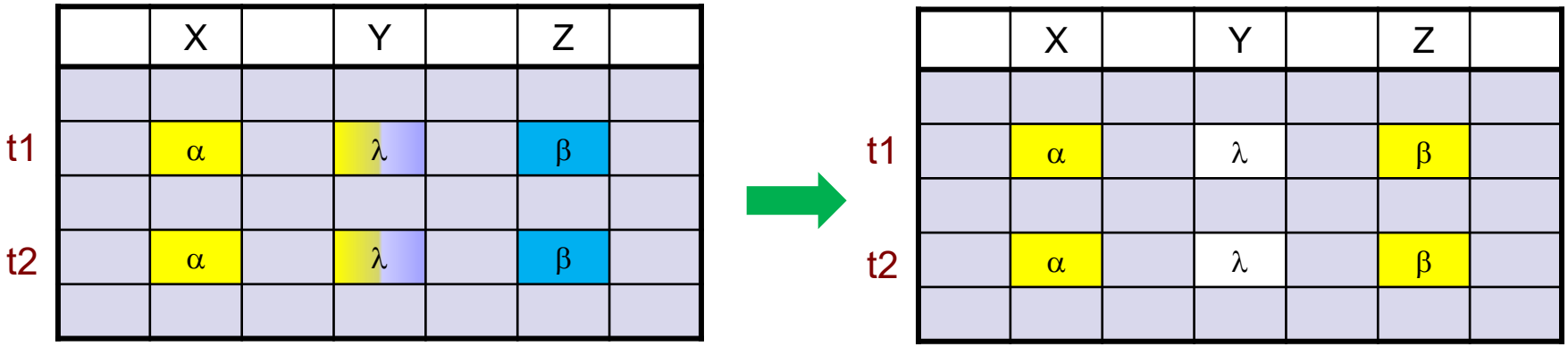
Proof of Soundness of Armstrong's Axioms



- Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$
- Assume $\exists t_1, t_2$ such that $t_1.XZ = t_2.XZ$
- $t_1.Z = t_2.Z$ (1)
- $t_1.X = t_2.X$
- $t_1.Y = t_2.Y$ definition of $X \rightarrow Y$ (2)
- $t_1.YZ = t_2.YZ$ from (1) and (2)
- Hence, $XZ \rightarrow YZ$

Recall $X \rightarrow Y$: for every pair t_1 and t_2 if $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$

Proof of Soundness of Armstrong's Axioms



- Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- Assume $\exists t_1, t_2$ such that $t_1.X = t_2.X$
- Then $t_1.Y = t_2.Y$ definition of $X \rightarrow Y$
- Hence, $t_1.Z = t_2.Z$ definition of $Y \rightarrow Z$
- Therefore, $X \rightarrow Z$

Recall $X \rightarrow Y$: for every pair t_1 and t_2 if $t_1.X = t_2.X$, then $t_1.Y = t_2.Y$

Additional Rules

- Sometimes, it is convenient to use some additional rules while reasoning about F^+

Suppose X,Y, and Z are sets of attributes over a relation.

- Union: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- Decomposition: if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

- These additional rules are not essential in the sense that their soundness can be proved using Armstrong's Axioms

Correctness of the Union Rule

- Union: $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- Proof:

$X \rightarrow Y$	… (1)	(given)
$X \rightarrow Z$	… (2)	(given)
$XX \rightarrow XY$	… (3)	(augmentation on (1))
$X \rightarrow XY$	… (4)	(simplify (3))
$XY \rightarrow ZY$	… (5)	(augmentation on (2))
$X \rightarrow ZY$	… (6)	(transitivity on (4) and (5))

Correctness of the Decomposition Rule

- Decomposition: $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$
- Proof:

$X \rightarrow YZ$	… (1) (given)
$YZ \rightarrow Y$	… (2) (reflexivity)
$X \rightarrow Y$	… (3) (transitivity on (1), (2))
$YZ \rightarrow Z$	… (4) (reflexivity)
$X \rightarrow Z$	… (5) (transitivity on (1), (4))

Example: Armstrong's Axioms

Student-ID	Student-Name	Course-ID	Course-Name	Department-Name
111	Chan	3170	DB	CSE
222	Wong	3170	DB	CSE
333	Tam	3160	Cal	MATH
111	Chan	3160	Cal	MATH

- Reflexivity
  - Student-ID, Student-Name → Student-ID
  - Student-ID, Student-Name → Student-Name

Trivial Dependency
LHS contains RHS

- Augmentation
  - Student-ID → Student-Name
  - ⇒ Student-ID, Course-Name → Student-Name, Course-Name
- Transitivity
  - Course-ID → Course-Name and Course-Name → Department-Name
  - ⇒ Course-ID → Department-Name

Example: Closure of F

Not in Final Exam

(Compute the closure of F)

R = (A, B, C)

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

F^+ = {

$A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$,

$AB \rightarrow AB$, $BC \rightarrow BC$, $AC \rightarrow AC$, $ABC \rightarrow ABC$,

$AB \rightarrow A$, $AB \rightarrow B$,

$BC \rightarrow B$, $BC \rightarrow C$,

$AC \rightarrow A$, $AC \rightarrow C$,

$ABC \rightarrow AB$, $ABC \rightarrow BC$, $ABC \rightarrow AC$,

$ABC \rightarrow A$, $ABC \rightarrow B$, $ABC \rightarrow C$,

Using reflexivity, we can generate all trivial dependencies

$A \rightarrow B$, … (1) (given)

$B \rightarrow C$, … (2) (given)

$A \rightarrow C$, … (3) (transitivity on (1) and (2))

$AC \rightarrow BC$, … (4) (augmentation on (1))

$AC \rightarrow B$, … (5) (decomposition on (4))

$A \rightarrow AB$, … (6) (augmentation on (1))

$AB \rightarrow AC$, $AB \rightarrow C$, $B \rightarrow BC$,

$A \rightarrow AC$, $AB \rightarrow BC$, $AB \rightarrow ABC$, $AC \rightarrow ABC$, $A \rightarrow BC$, $A \rightarrow ABC$ }

Attribute Closure

- Computing the closure of a set of FDs can be expensive (time-consuming)
- In many cases, we just want to check if a given FD

  $X \rightarrow Y$ is in F^+

  X - a set of attributes

  F - a set of functional dependencies

Given a set of attributes $A_1, \ldots, A_n$

The **closure**, $\{A_1, \ldots, A_n\}^+ = \{B \in Atts \mid A_1, \ldots, A_n \rightarrow B\}$

Algorithm: Attribute Closure

- α – a set of attributes
- α^+ – closure of α under F
- Algorithm to compute α^+:

  closure := α

  while(changes in result) do

    for each $\beta \rightarrow \gamma$ do

      if $\beta \subseteq$ closure then closure := closure $\cup \gamma$

    end for

  end while

Example: Attribute Closure

- R = (A, B, C, D)
- F = { A → B, B → C } ⇒ A → C
- To compute A^+ : $A^+ = ABC$, $B^+ = BC$, $C^+ = C$, $AB^+ = ABC$

  closure = A

  closure = AB (A → B)

  closure = ABC (B → C) $A^+ = \{A\ B\ C\}$
- Also, $B^+ = BC$, $C^+ = C$, $AB^+ = ABC$

Example: Attribute Closure

$AG^+ = ABCGHI$

closure = AG, ABG, ABCG, ABCGHI

$A^+ = ABCH$

$G^+ = G$

- R = (A, B, C, G, H, I)
- F = { $A \rightarrow B$, $A \rightarrow C$, $CG \rightarrow H$, $CG \rightarrow I$, $B \rightarrow H$ }
- To compute AG^+:

  closure = AG

  closure = ABG ($A \rightarrow B$)

  closure = ABCG ($A \rightarrow C$)

  closure = ABCGH ($CG \rightarrow H$)

  closure = ABCGHI ($CG \rightarrow I$)

YES!

Is AG a candidate key?

$AG \rightarrow R$ ✓

$A^+ \rightarrow R$? ✗

$G^+ \rightarrow R$? ✗

- To check if a given FD $AG \rightarrow HI$ is in F^+, Check if HI is in AG^+

Example: Attribute Closure

Given:

R(A,B,C,D,E,F)

$F = \{AB \to C, AD \to E, B \to D, AF \to B\}$

$AB^+ = ABCDE$ $AF^+ = ABCDEF$

A, B	→	C
A, D	→	E
B	→	D
A, F	→	B

↓ candidate key

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

Compute: $\{A,F\}^+$ = {A, F, B, C, D, E }

Relational Database Design

- Given a relation schema, we need to decide whether it is a good design or we need to decompose it into smaller relations
- Such a decision must be guided by an understanding of what problems arise from the current schema
- To provide such guidance, several *normal forms* have been proposed
  - If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise

Outline

- Problems Caused by Redundancy
- Functional Dependencies (FDs)
- **Normal Forms Based on Keys**
- Decomposition
  - Lossless Join Decomposition
  - Dependency Preserving Decomposition
- Normalization
  - BCNF Decomposition Algorithm
  - 3NF Decomposition Algorithm

Normal Forms

1st Normal Form	No repeating data groups
2nd Normal Form	No partial key dependency
3rd Normal Form	No transitive dependency
Boyce-Codd Normal Form	Reduce keys dependency
4th Normal Form	No multi-valued dependency
5th Normal Form	No join dependency

$$1NF \supset 2NF \supset 3NF \supset BCNF \supset 4NF \supset 5NF$$

dangerous → safe

Normal Forms

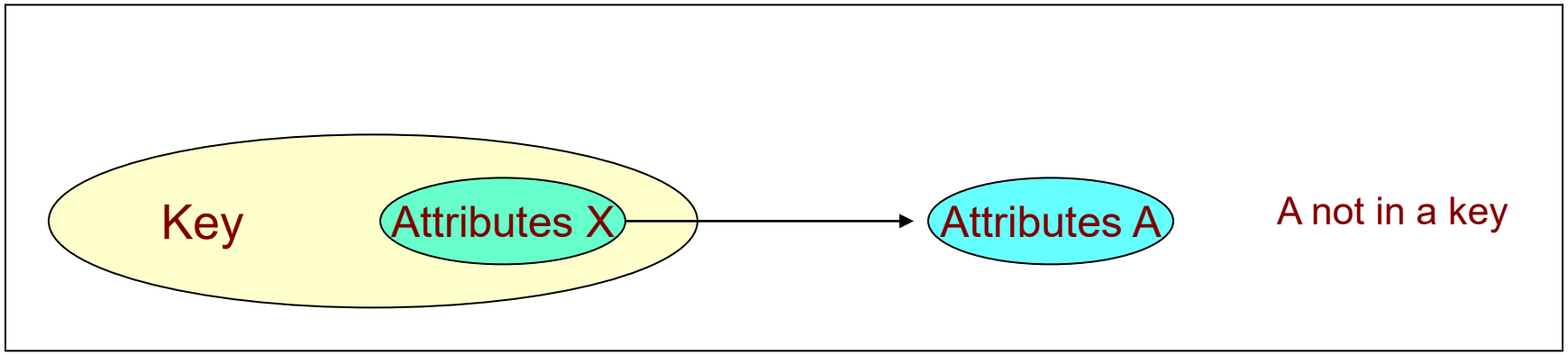
- First Normal Form
  - Every field contains only atomic values
  - No lists or sets
  - Implicit in our definition of the relational model
- Second Normal Form
  - Is in 1NF
  - Every non-key attribute (*non-prime attribute*) is fully functionally dependent on the **ENTIRE** primary key (*any candidate key*), i.e., *no partial key dependency*
  - Mainly of historical interest

NOTE: non-prime attribute, an attribute does not belong to any candidate key

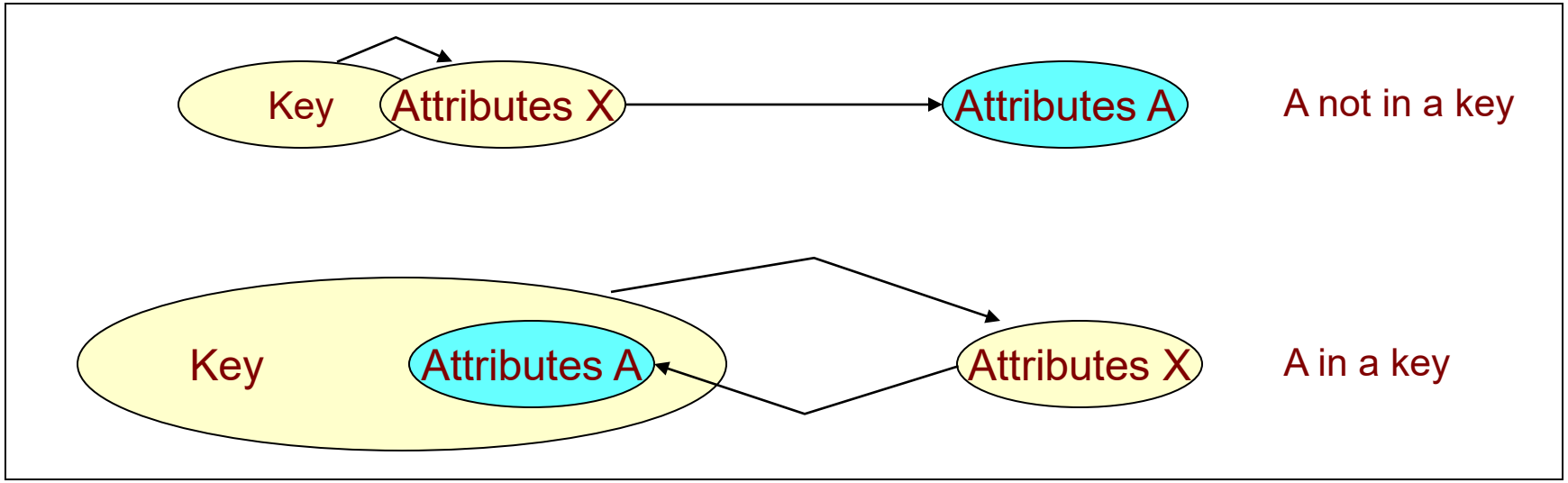
Normal Forms

- Third Normal Form
  - Is in 2NF
  - Every non-key attribute (*non-prime attribute*) is *non-transitively dependent* on a primary key (*any candidate key*), i.e., *no transitive dependency*
- Boyce-Codd Normal Form
  - Is in 3NF
  - *Left side* of any non-trivial FD in the table must be a primary key (*a candidate key*)

NOTE: non-prime attribute, an attribute does not belong to any candidate key



Partial Dependencies (for 2NF)



Transitive Dependencies (for 3NF, BCNF)

Example: 1NF

(a)

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, AliciaJ.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

store a repeating group of data items on a single row

Not 1NF

(c)

EMP_PROJ1

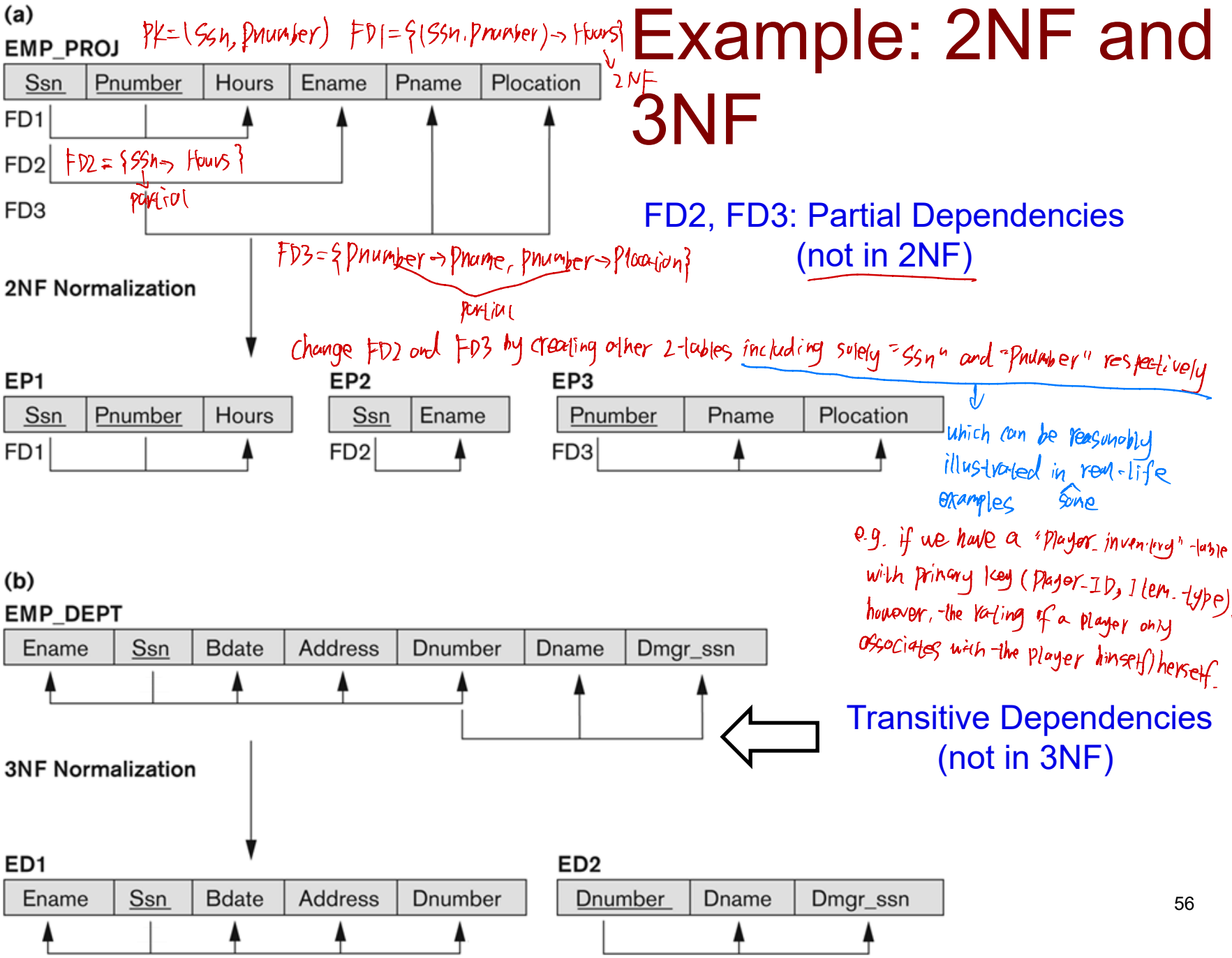
Ssn	Ename

EMP_PROJ2

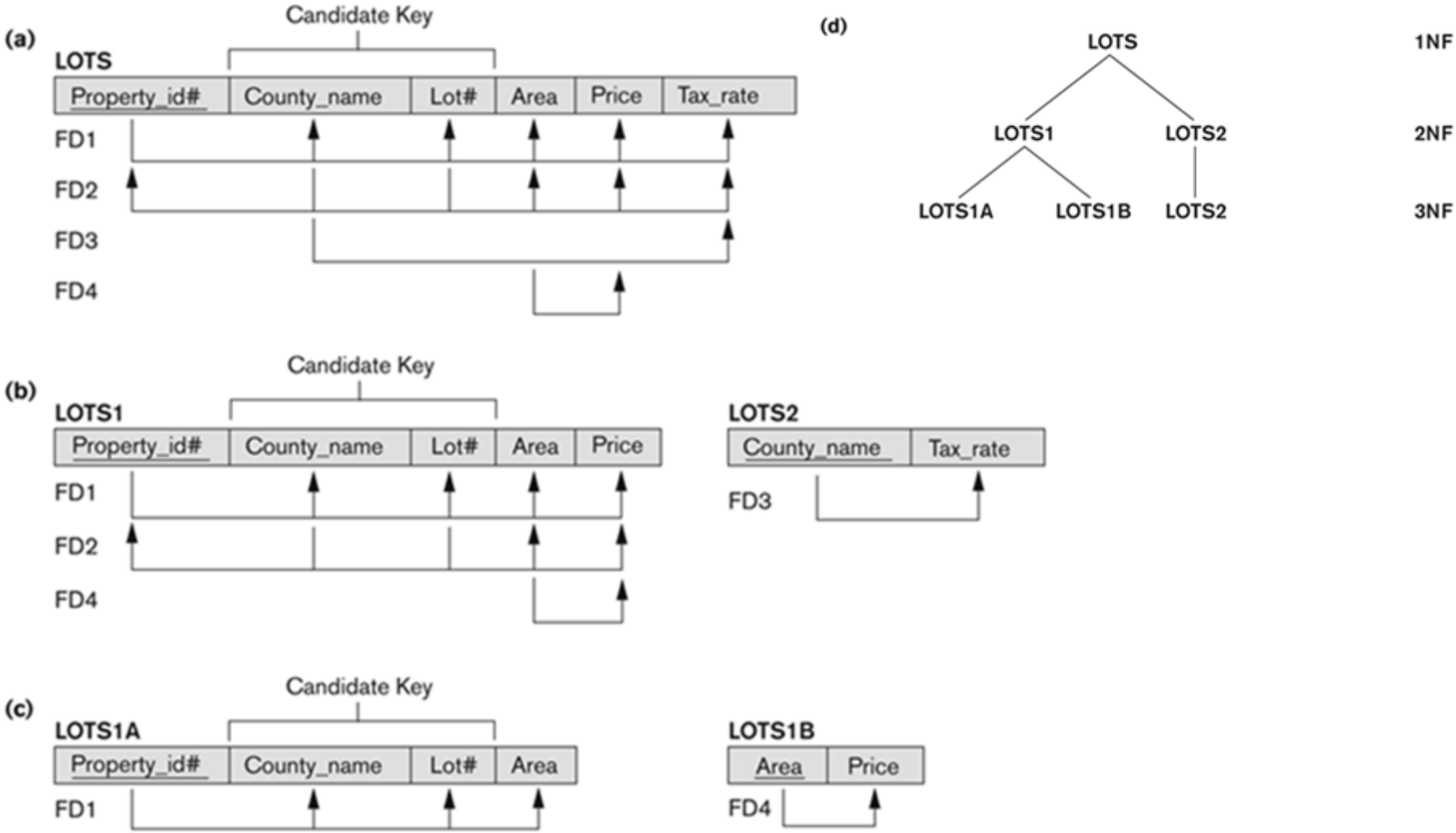
Ssn	Pnumber	Hours

IS 1NF

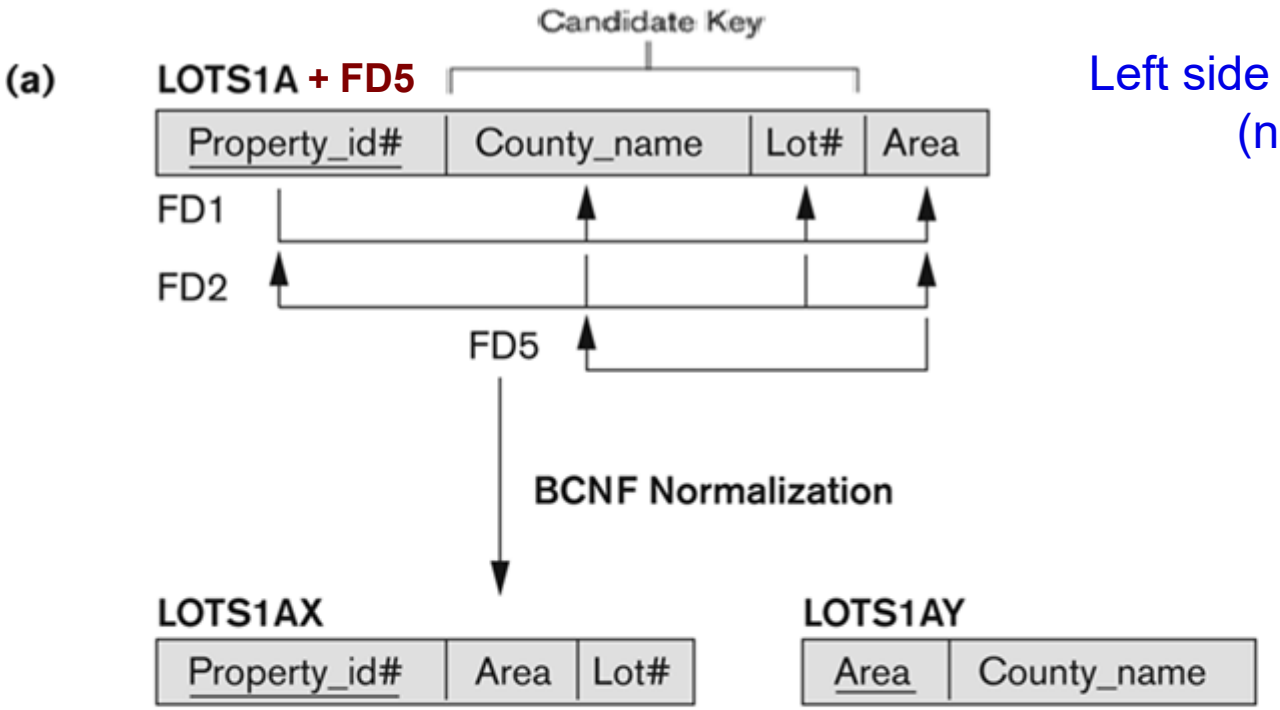
Example: 2NF and 3NF



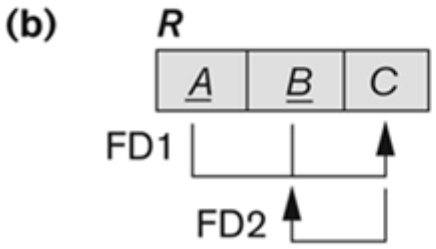
Example: 3NF and BCNF



Example: 3NF and BCNF



Left side of FD5 is not a Key
(not in BCNF)

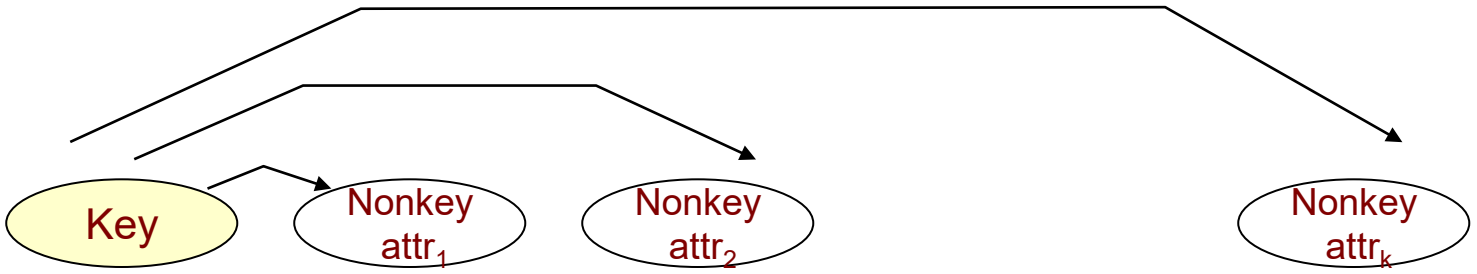


Boyce-Codd Normal Form (BCNF)

- R – a relation schema
- F – a set of functional dependencies on R
- R is in BCNF if for any $\alpha \rightarrow A$ in F
  - $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
  - α is a key for R

Boyce-Codd Normal Form (BCNF)

- Intuitively, in a BCNF relation, the only nontrivial dependencies are those in which a *key* determines some attributes
- Each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes



FDs in a BCNF Relation

Example: BCNF

R is in BCNF if for any $\alpha \rightarrow A$ in F
- $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
- α is a key for R

- R = (A, B, C)
- F = { $A \rightarrow B$, $B \rightarrow C$ } A→C
- Key = { A } candidate key
- R is ***not*** in BCNF, why? functional dependency that is transitive
  - F = {$A \rightarrow B$, $B \rightarrow C$}
  - B is not a key,
  - { C } $\not\subset$ { B } (non-trivial)
  - Decompose R into R_1 = (A, B), R_2 = (B, C), then R_1, R_2 are in BCNF

A	B	C
a1	b1	c1
a2	b1	c1
a3	b1	c1
a4	b2	c2

A	B
a1	b1
a2	b1
a3	b1
a4	b2

$A \rightarrow B$

B	C
b1	c1
b2	c2

$B \rightarrow C$

Example: BCNF

R is in BCNF if for any $\alpha \rightarrow A$ in F
- $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
- α is a key for R

- R = (A, B, C)
- F = { $A \rightarrow B$, $C \rightarrow B$ }
- Key = { AC } $AC^+ = ABC = R$
- *R* is ***not*** in BCNF
- Decomposition into
  - R_1 = (A, B), R_2 = (B, C)
  - R_1 and R_2 are in BCNF
  - What is the result when we join R_1 and R_2?

$\exists$ $AC \rightarrow C$, $C \rightarrow B$ → transitivity functional dependency

A	B	C
a1	b1	c1
a1	b1	c3
a3	b1	c1
a4	b2	c2

A	B
a1	b1
a3	b1
a4	b2

$A \rightarrow B$

B	C
b1	c1
b1	c3
b2	c2

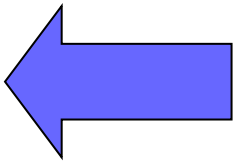
$C \rightarrow B$

A	B	C
a1	b1	c1
a1	b1	c3
a3	b1	c1
a4	b2	c2

$R_1 \bowtie R_2 = R$

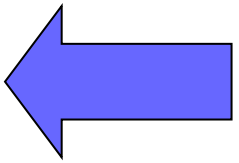
Third Normal Form (3NF)

- R – a relation schema
- F – a set of functional dependencies on R
- A – a single attribute in R
- R is in 3NF if for any $\alpha \rightarrow A$ in F
  - $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
  - α is a key for R, *or*
  - A is *part of some key(s)* for R
    - *part of some key(s)*: AB is a key for R
      - then A is a part of the key
      - and B is also a part of the key
- R is in BCNF $\Rightarrow$ R is in 3NF BCNF는 3NF



Second Normal Form (2NF)

- R – a relation schema
- F – a set of functional dependencies on R
- A – a single attribute in R
- R is in 2NF if for any $\alpha \rightarrow A$ in F
  - $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
  - α is *not a proper subset of a key* for R, *or*
  - A is *part of some key(s)* for R
    - *part of some key(s)*: AB is a key for R
      - then A is a part of the key
      - and B is also a part of the key
- R is in 3NF $\Rightarrow$ R is in 2NF



Example

R is in 3NF if for any $\alpha \rightarrow A$ in F
- $\alpha \rightarrow A$ is trivial ($A \in \alpha$), *or*
- α is a key for R, *or*
- A is *part of* some key(s) for R

- R = (A, B, C, D, E)
- F = { AE $\rightarrow$ BCD, D $\rightarrow$ A }

$AE^+ = ABCDE = R$

$D \rightarrow A \Rightarrow DE \rightarrow AE \Rightarrow DE^+ = R$

- Key = { AE, DE }
- R is in 3NF
- Is R in BCNF?
  - No. D is not a key for R
  - BCNF implies 3NF but 3NF cannot imply BCNF

Motivation of 3NF

- By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only *lossless-join*, *dependency-preserving* decompositions
- Such a guarantee does not exist for BCNF relations.
- It weaken the BCNF requirements just enough to make this guarantee possible

Motivation of 3NF

- Unlike BCNF, *some redundancy* is possible with 3NF
- The problems associate with partial and transitive dependencies persist if there is a nontrivial dependency
  - $X \rightarrow A$ and
  - X is not a key, even if the relation is in 3NF because *A is part of a key*

Example: 3NF

Reserves

sid	bid	day	cardno
1	b1	April 5, 00	2323
2	b2	May 10, 01	1000
1	b3	January 5, 01	2323
3	b4	June 1, 00	4000

- Assume: *sid* $\rightarrow$ *cardno*

  (a sailor uses a unique credit card to pay for reservations)

- Reserves is **not** in 3NF
  - (sid, bid, day) is the only key
  - sid $\rightarrow$ cardno violates 3NF (also violates 2NF) because
  - sid is not a key and cardno is not part of a key
  - (sid, cardno) pairs are redundantly stored

Example: 3NF

Reserves

sid	bid	day	cardno
1	b1	April 5, 00	2323
2	b2	May 10, 01	1000
1	b3	January 5, 01	2323
3	b4	June 1, 00	4000

- Assume: *sid* → *cardno* and *cardno* → *sid* (we know that credit cards also uniquely identify the owner).
- Reserves is in 3NF
  - (sid, bid, day) is a key for Reserves
  - (cardno, bid, day) is also a key for Reserves
  - cardno → sid does not violate 3NF, since sid is in a key
  - sid → cardno does not violate 3NF, since cardno is in a key

(sid, cardno) pairs are redundantly stored

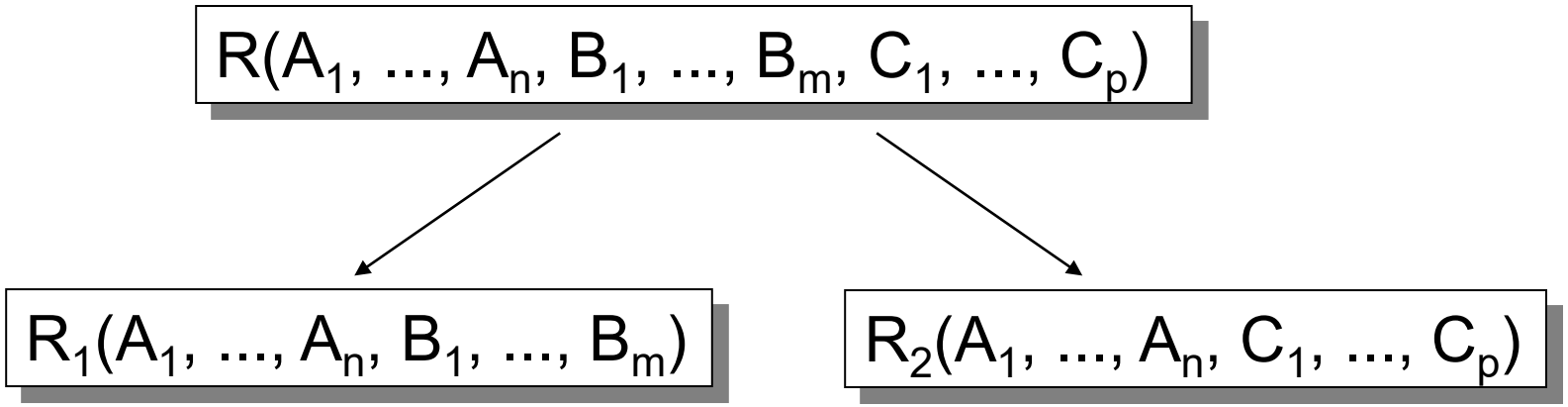
Outline

- Problems Caused by Redundancy
- Functional Dependencies (FDs)
- Normal Forms Based on Keys
- **Decomposition**
  - **Lossless Join Decomposition**
  - Dependency Preserving Decomposition
- Normalization
  - BCNF Decomposition Algorithm
  - 3NF Decomposition Algorithm

Decomposition

- Decomposition is a tool that allows us to eliminate redundancy
- It is important to check that a decomposition does not introduce new problems
  - A decomposition allows us to recover the original relation?
  - Can we check integrity constraints efficiently?

Projection for Decomposition



After decomposition, make sure that for all subsets $R_1, R_2, \ldots, R_m$ such that $R_1 \cup R_2 \cup \cdots \cup R_m = R$

R_1 = projection of R on $A_1, \ldots, A_n, B_1, \ldots, B_m$

R_2 = projection of R on $A_1, \ldots, A_n, C_1, \ldots, C_p$

$A_1, \ldots, A_n \cup B_1, \ldots, B_m \cup C_1, \ldots, C_p$ = all attributes

R_1 and R_2 may (not) be reassembled to produce original R

Example: Decomposition

Supply

sid	status	city	part_id	qty

Supplier

sid	status	city

SP

sid	part_id	qty

Supplier $\cup$ SP = Supply

{ Supplier, SP } is a decomposition of Supply

A set of relation schemas { R_1, R_2, …, R_n }, with n $\geq$ 2 is a **decomposition** of R if

$R_1 \cup R_2 \cup \ldots \cup R_n = R$

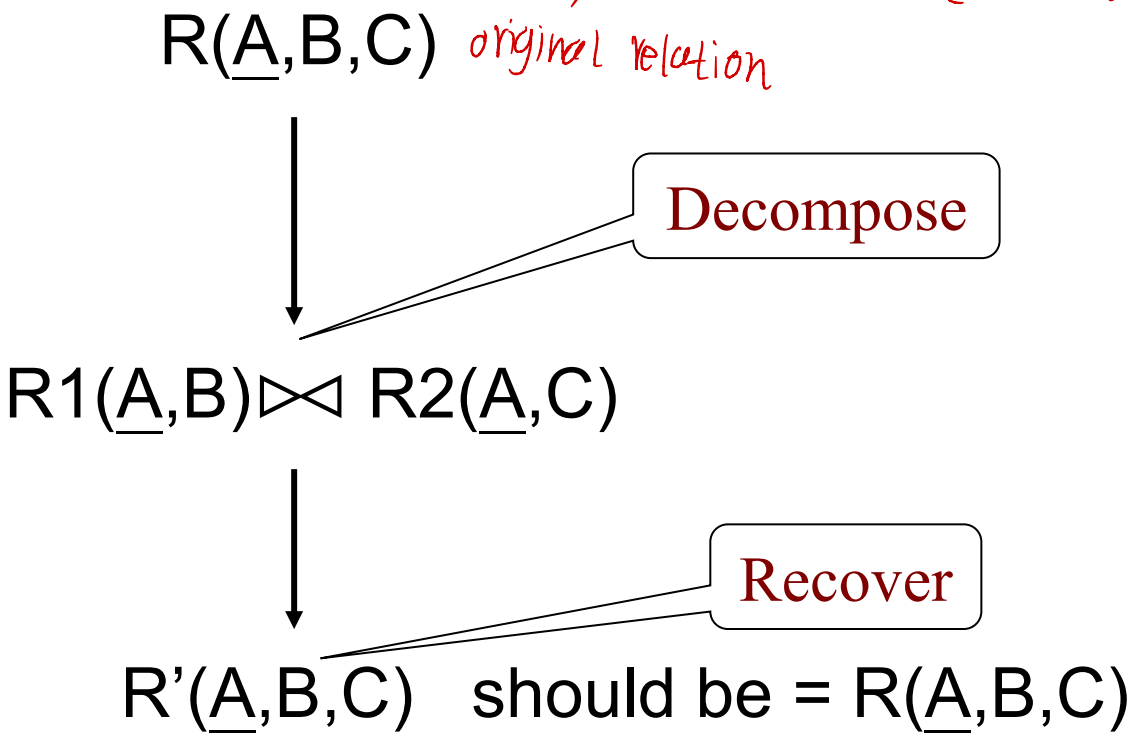
Problems with Decomposition

- Some queries become more expensive
- Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation – information loss
- Checking some dependencies may require joining the instances of the decomposed relations

Lossless Decompositions

A decomposition is *lossless* if we can recover:

which means we can reconstruct the attributes in the original relation

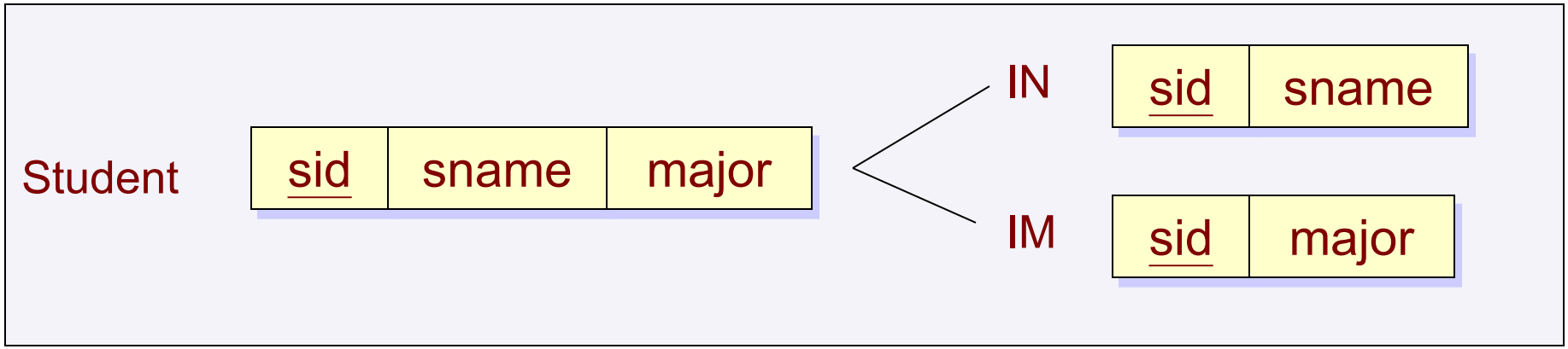


R' is in general larger than R. Must ensure R' = R

Lossless Join Decomposition

- { R_1, …, R_n } – a set of relation schemas
- { R_1, …, R_n } is a *decomposition* of R if $R_1 \cup R_2 \cup \ldots \cup R_n = R$
- { R_1, …, R_n } is a *lossless-join decomposition* of R if, for all relations *r* on schema R, $\Pi_{R_1}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r) = r$

Example: A Lossless Join Decomposition



Student

sid	sname	major
123	Ling Wang	C.S.
456	Hong Zhou	C.S.

IN

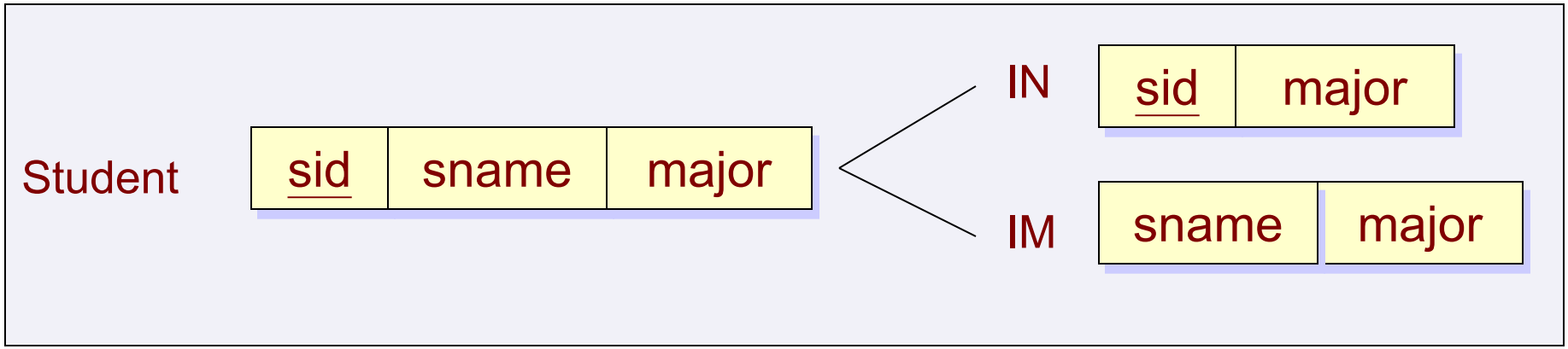
sid	sname
123	Ling Wang
456	Hong Zhou

IM

sid	major
123	C.S.
456	C.S.

'Student' can be recovered by joining the instances of IN and IM

Example: A Non-Lossless Join Decomposition



Student

sid	sname	major
123	Ling Wang	C.S.
456	Hong Zhou	C.S.

IN

sid	major
123	C.S.
456	C.S.

Student = IN ⋈ IM???? No!

IM

sname	major
Ling Wang	C.S.
Hong Zhou	C.S.

Example: A Non-Lossless Join Decomposition

IN

sid	major
123	C.S.
456	C.S.

IM

sname	major
Ling Wang	C.S.
Hong Zhou	C.S.

IN ⋈ IM

sid	sname	major
123	Ling Wang	C.S.
123	Hong Zhou	C.S.
456	Ling Wang	C.S.
456	Hong Zhou	C.S.

≠

Student

sid	sname	major
123	Ling Wang	C.S.
456	Hong Zhou	C.S.

The instance of ‘Student’ cannot be recovered by joining the instances of IM and NM. Therefore, such a decomposition is not a lossless join decomposition

Theorem: Lossless Join Decomposition

- R – a relation schema
- F – a set of functional dependencies on R
- { R_1, R_2 } is a lossless-join decomposition of R

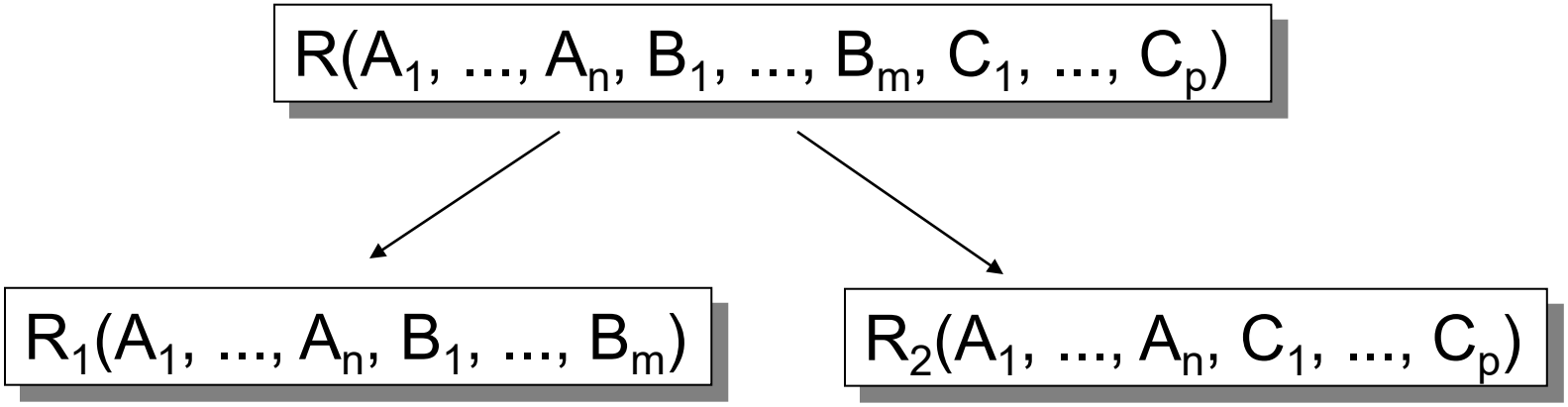
$\Leftrightarrow (R_1 \cap R_2) \rightarrow R_1 \in F^+$ or $(R_1 \cap R_2) \rightarrow R_2 \in F^+$

$\Leftrightarrow (R_1 \cap R_2)$ is a *key* for R_1 or R_2

not refers to "candidate key"

- (the attributes common to R_1 and R_2 must contain a key for either R_1 or R_2)

Lossless Join Decomposition



If $A_1, ..., A_n \rightarrow B_1, ..., B_m$ or $A_1, ..., A_n \rightarrow C_1, ..., C_p$

Then the decomposition is lossless

Note: don't need both

Example: Non-Lossless Join

R(A, B, C) F = {B → C, C → B}

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

$A \rightarrow AB \in F^+?$ no

$A \rightarrow AC \in F^+?$ no

Decompose

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2

A	B
a_1	b_1
a_1	b_2

A	C
a_1	c_1
a_1	c_2

Join

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_1
a_1	b_2	c_2

Example: Lossless Join

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

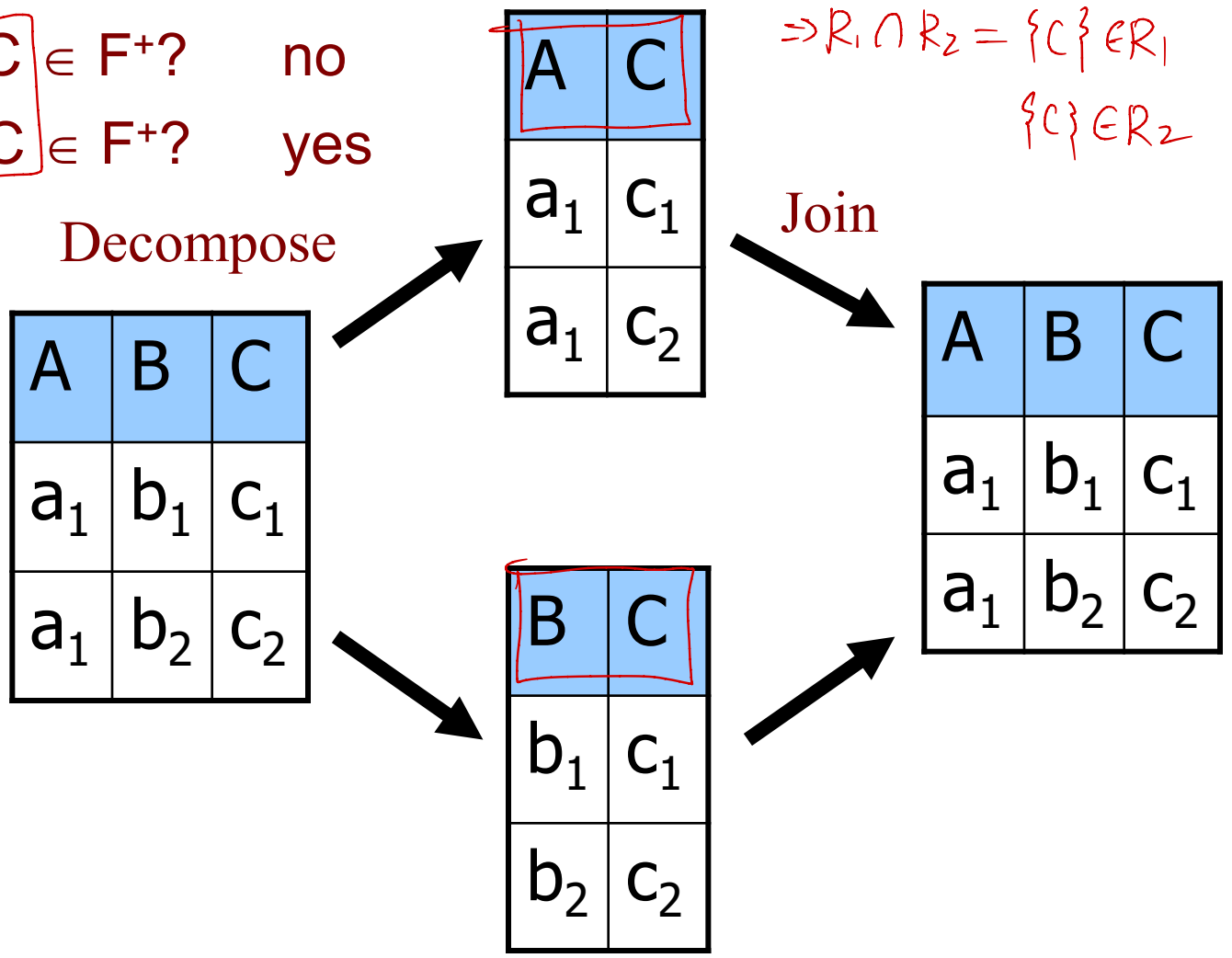
$R(A,B,C)$ $R_1(A,C), R_2(B,C)$

$C \rightarrow AC \in F^+?$ no

$C \rightarrow BC \in F^+?$ yes

$\Rightarrow R_1 \cap R_2 = \{C\} \in R_1$

$\{C\} \in R_2$



Example: Lossless Join Decomposition

- R = (A, B, C)
- F = { $A \rightarrow B$ }
- { A, B } + { A, C } is a lossless join decomposition
- { A, B } $\cap$ { A, C } = {A} is a key in { A, B }
- { A, B } + { B, C } is not a lossless join decomposition
- { A, B } $\cap$ { B, C } = {B} is neither a key in {A,B} nor {B,C}

Another Example: Decomposition

$R = (A, B, C, D)$

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

$R_1 = (A, B)$

$R_2 (A, C, D)$

$R_3 = (A, C)$

$R_4 = (C, D)$

$\Rightarrow R_1(A, B)\ R_2(A, C)\ R_3(A, D)$

$R_1 \bowtie R_3 \bowtie R_2 = (A, B, C) \bowtie R_2 = (A, B, C, D)$

Decomposition: $\{ R_1(A, B), R_2(C, D), R_3(A, C) \}$

Consider it a two step decomposition:

Introduce virtually

1. Decompose R into $R_1 = (A, B)$, $R_2 = (A, C, D)$
2. Decompose R_2 into $R_3 = (C, D)$, $R_4 = (A, C)$

This is a lossless join decomposition

If R is decomposed into (A, B), (C, D)

This is a lossy-join decomposition

Outline

- Problems Caused by Redundancy
- Functional Dependencies (FDs)
- Normal Forms Based on Keys
- **Decomposition**
  - Lossless Join Decomposition
  - **Dependency Preserving Decomposition**
- Normalization
  - BCNF Decomposition Algorithm
  - 3NF Decomposition Algorithm

Dependency Preservation

- R – a relation schema
- F – a set of functional dependencies on R
- { R_1, R_2 } is a decomposition of R
- F_i – a subset of F with only attributes in R_i
  (i.e. the *projection* of F on R_i)
- Dependency is *preserved* if $(F_1 \cup F_2)^+ = F^+$
  e.g.: R = (A, B, C), F = {A→B, B→C}
  R_1 = (A, B) and R_2 = (B, C)
  F_1 = {A→B} and F_2 = {B→C}
- A dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple

Dependency Preserving Decomposition

- E.g., R = (A, B, C), F = {A→B, C→B}

  R_1 = (A, B) and R_2 = (A, C)

  F_1 = {A→B} and F_2 = {}

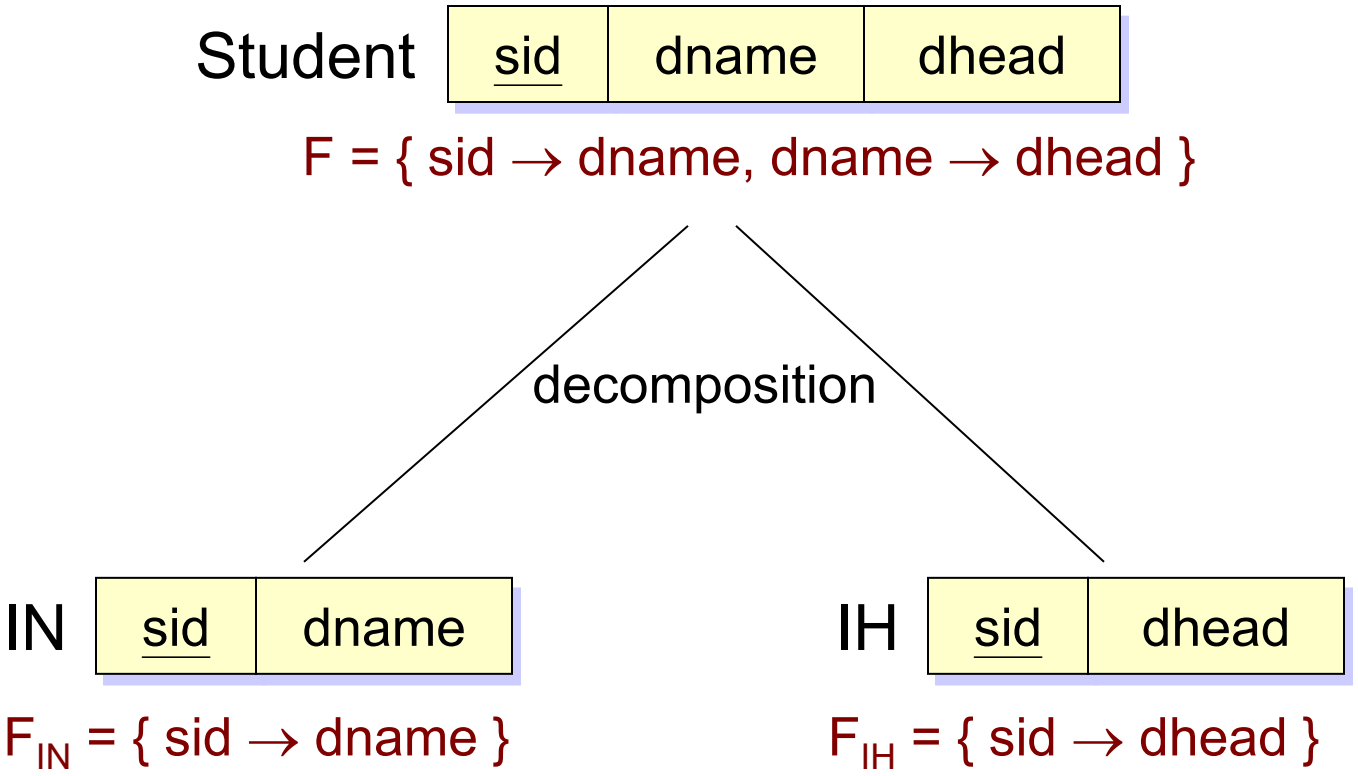
  C→B is not in both F_1 and F_2

- If dependency is **NOT** preserved, *joins must be computed* in order to check if an update is illegal

- Very inefficient … !!!

Example: Dependency Preservation

Dependency set: F = { sid → dname, dname → dhead }



IN

sid	dname

IH

sid	dhead

Dependency set: F = { sid $\rightarrow$ dname, dname $\rightarrow$ dhead }

F_{IN} = { trivial dependencies, sid $\rightarrow$ dname,

sid $\rightarrow$ sid dname}

F_{IH} = { trivial dependencies, sid $\rightarrow$ dhead,

sid $\rightarrow$ sid dhead }

Dependency Preserving?

- (dname $\rightarrow$ dhead) $\in$ F^+ but
- (dname $\rightarrow$ dhead) $\notin$ ($F_{IN} \cup F_{IH}$)$^+$

This decomposition ***does not*** preserve dependency

IN | sid | dname

IH | sid | dhead

Dependency set: F = { sid → dname, dname → dhead }

Student

sid	dname	dhead
123	C.S.	Ying Wang
456	C.S.	Ying Wang

F_{IN} = { sid → dname }

IN

sid	dname
123	C.S.
456	C.S.

F_{IH} = { sid → dhead }

IH

sid	dhead
123	Ying Wang
456	Ying Wang

The update violates the FD

'dname → dhead'

However, it can only be caught

when we join IN and IH

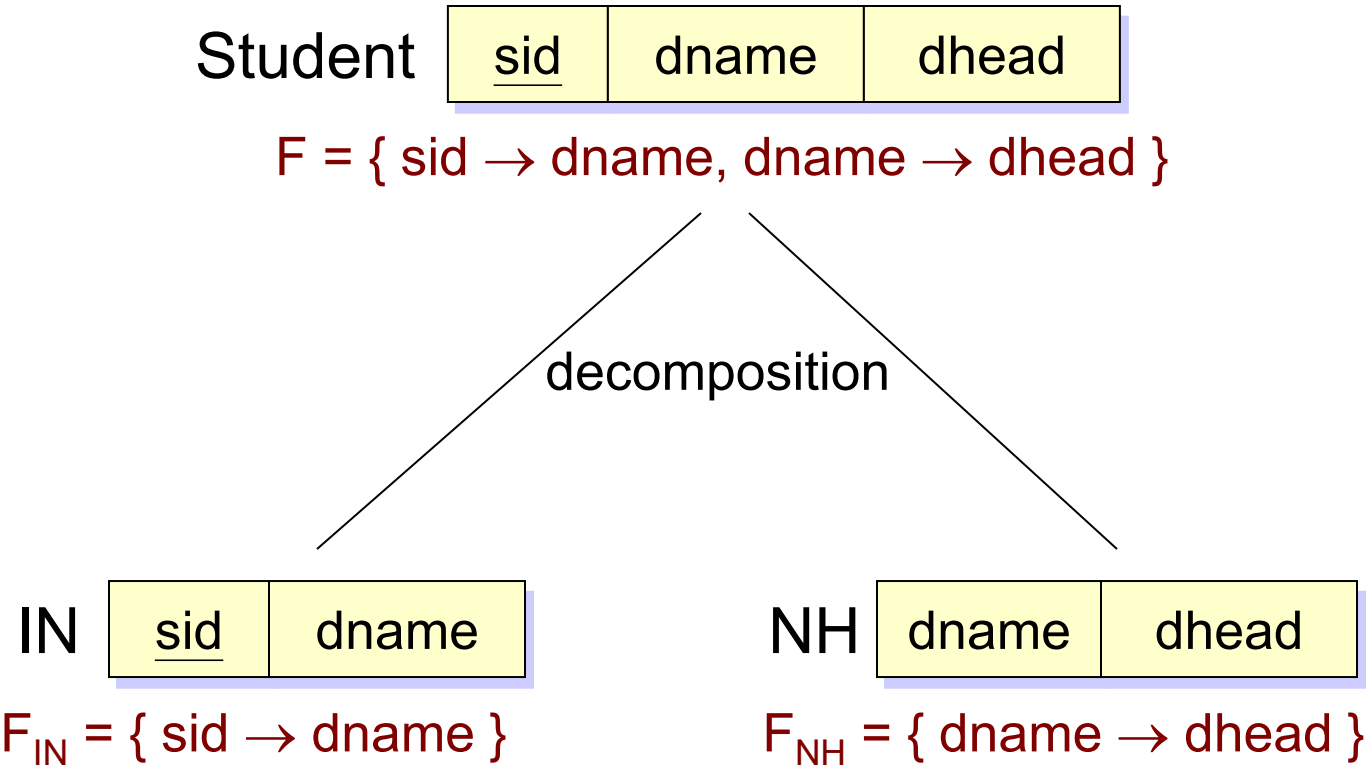
Updated to

sid	dhead
123	Ying Wang
456	Lin Cheung

Example: Dependency Preservation

Dependency set: F = { sid → dname, dname → dhead }

Let's decompose the relation in another way



IN	sid	dname

NH	dname	dhead

Dependency set: F = { sid $\rightarrow$ dname, dname $\rightarrow$ dhead }

F_{IN} = { trivial dependencies, sid $\rightarrow$ dname,

sid $\rightarrow$ sid dname }

F_{NH} = { trivial dependencies, dname $\rightarrow$ dhead,

dname $\rightarrow$ dname dhead }

Dependency Preserving?

- $(F_{IN} \cup F_{NH})^{+} = F^{+}$

This decomposition preserve dependency

IN	sid	dname

NH	dname	dhead

Dependency set: F = { sid → dname, dname → dhead }

Student

sid	dname	dhead
123	C.S.	Ying Wang
456	C.S.	Ying Wang

IN

sid	dname
123	C.S.
456	C.S.

NH

dname	dhead
C.S.	Ying Wang
C.S.	Ying Wang

Updated to

dname	dhead
C.S.	Ying Wang
C.S.	Lin Cheung

The error in NH will immediately be caught by the DBMS,

since it violates F.D. dname → dhead

No join is necessary for the detection

Outline

- Problems Caused by Redundancy
- Functional Dependencies (FDs)
- Normal Forms Based on Keys
- Decomposition
  - Lossless Join Decomposition
  - Dependency Preserving Decomposition
- **Normalization**
  - **BCNF Decomposition Algorithm**
  - 3NF Decomposition Algorithm

Normalization

- Converting relations to BCNF or 3NF
- If a relation schema is not in BCNF
  - it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas
  - Dependency-preserving is *not* guaranteed
- 3NF
  - There is always a dependency-preserving, lossless-join decomposition into a collection of 3NF relation schemas

BCNF Definition (Review)

- A simple condition for removing anomalies from relations:

> A relation R is in BCNF if:
>
> If **A → B** is a non-trivial dependency
>
> in R, then A is a key for R

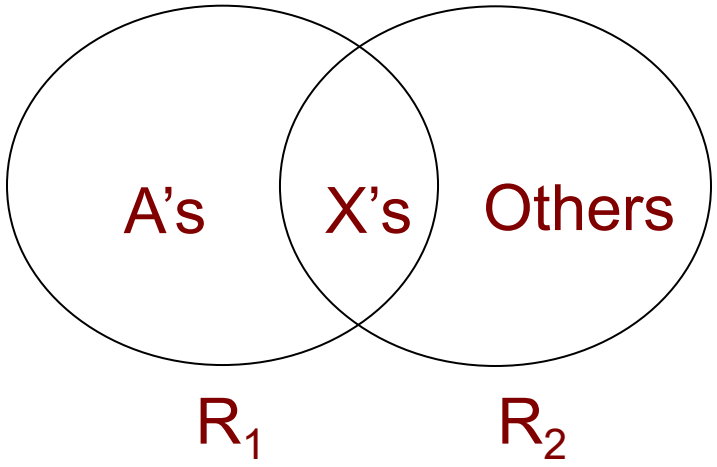
- i.e.: The left side must always contain a key
- i.e.: If a set of attributes determines other attributes, it must determine all the attributes

Algorithm: BCNF Decomposition

Suppose R is not in BCNF, A is an attribute, and $X \rightarrow A$ (X_1, …, X_m → A_1, …, A_n) is a FD that *violates* the BCNF condition.

1. Remove A from R
2. Decompose R into XA and R-A

   $R_1(X_1, \ldots, X_m, A_1, \ldots, A_n)$ and $R_2(X_1, \ldots, X_m, [\text{others}])$
3. Repeat this process until all the relations become BCNF



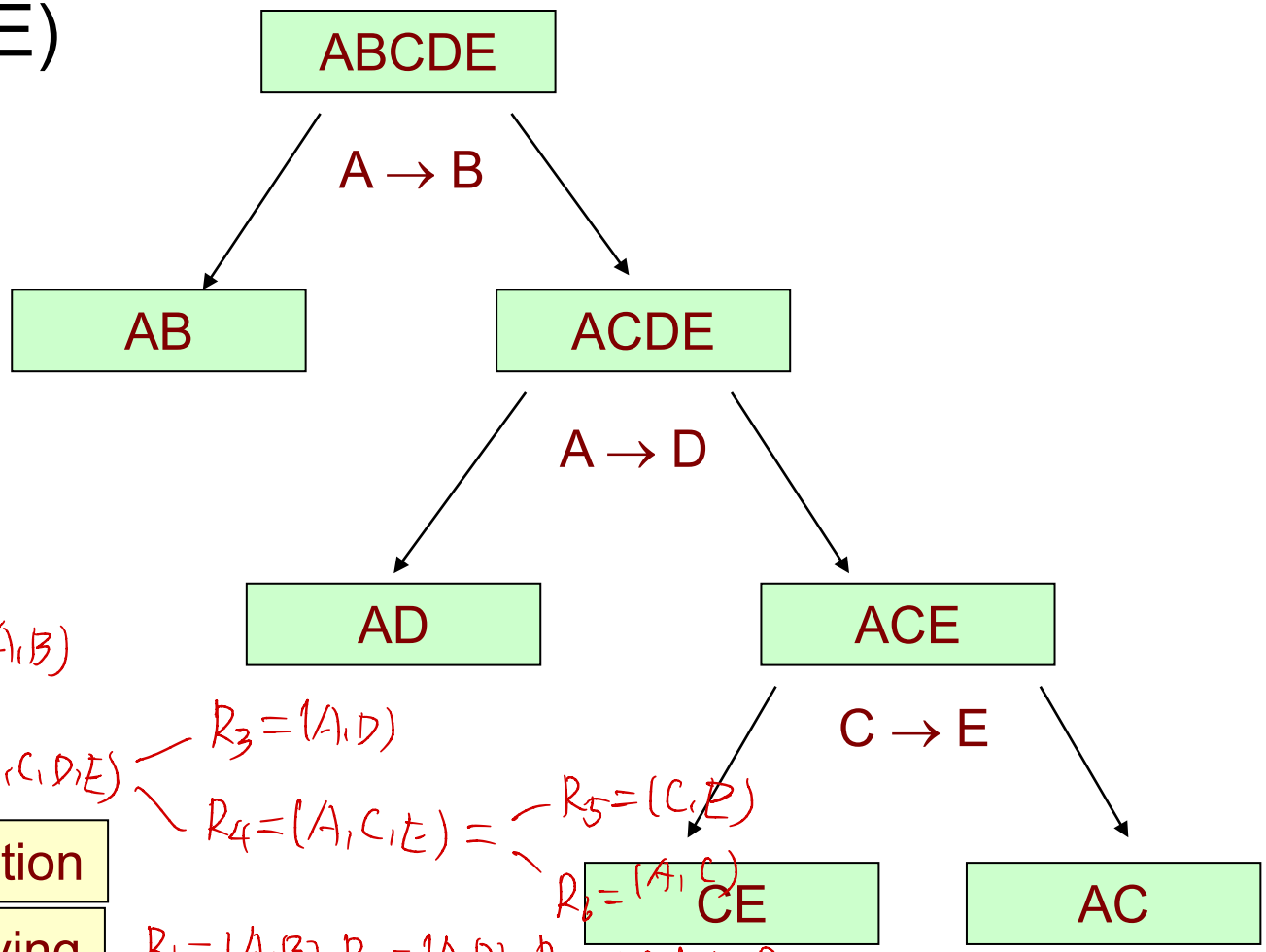
Example: BCNF Decomposition

$F = \{A \to B,\ A \to D,\ C \to E\}$ $AC^+ = ABCDE = R \Rightarrow Key = AC$

- R = (A,B,C,D,E)
- $A \to B$

  $A \to D$

  $C \to E$
- Key = AC



$R = (A,B,C,D,E)$ → $R_1 = (A,B)$, $R_2 = (A,C,D,E)$

$R_2 = (A,C,D,E)$ → $R_3 = (A,D)$, $R_4 = (A,C,E)$

$R_4 = (A,C,E)$ → $R_5 = (C,E)$, $R_6 = (A,C)$

Lossless decomposition

Dependency preserving

$R_1 = (A,B)$ $R_2 = (A,D)$, $R_3 = (C,E)$, $R_4 = (A,C)$

Verification: $R_1 \bowtie R_4 \bowtie R_3 \bowtie R_2 = (A,B,C) \bowtie (C,E) \bowtie (A,D) = (A,B,C,D,E) = R$, lossless

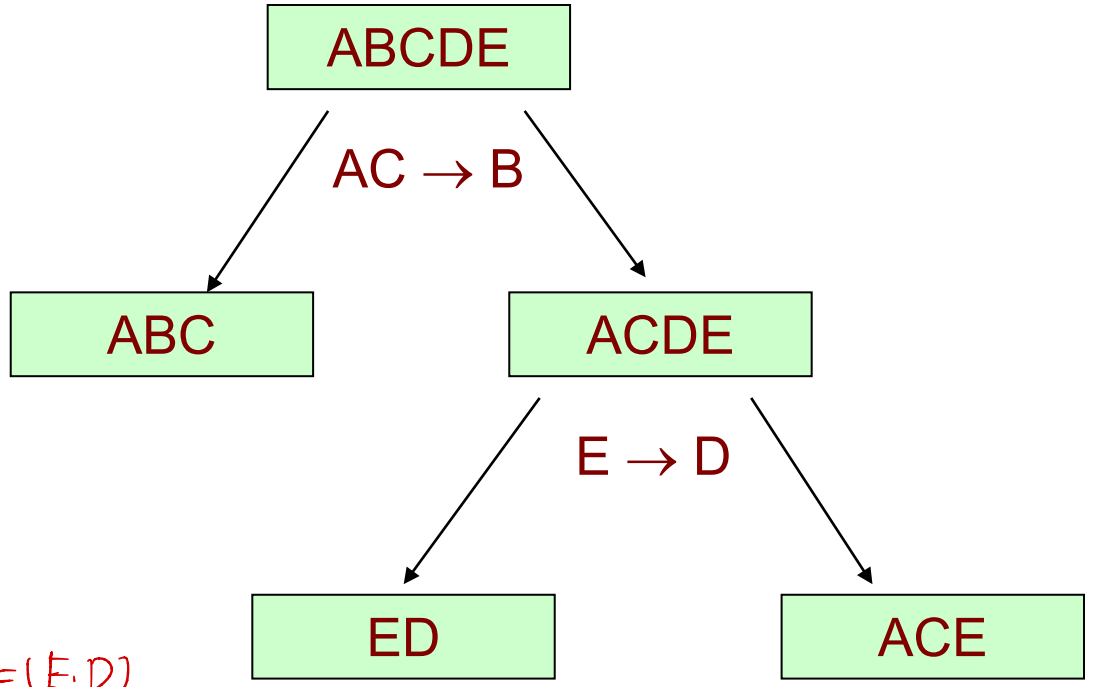
Example: BCNF Decomposition

$F = \{AC \to B, E \to D, BE \to A\} \Rightarrow ACE^+ = ABCDE = R,\ BCE^+ = ABCDE = R \Rightarrow$ key = ACE, BCE

- R = (A,B,C,D,E)
- AC → B

  E → D

  BE → A
- Key = ACE, BCE



$R = (A,B,C,D,E)$ → $R_1 = (A,B,C)$, $R_2 = (A,C,D,E)$ → $R_3 = (E,D)$, $R_4 = (A,C,E)$

$\Rightarrow R_1 = (A,B,C),\ R_2 = (E,D),\ R_3 = (A,C,E)$

Lossless decomposition

Not Dependency preserving

no way to decompose R_4 according to the lost dependency "BE → A" in F

⇒ NOT dependency preserving

Verification: $R_1 \bowtie R_3 \bowtie R_2 = (A,B,C,D,E) = R$

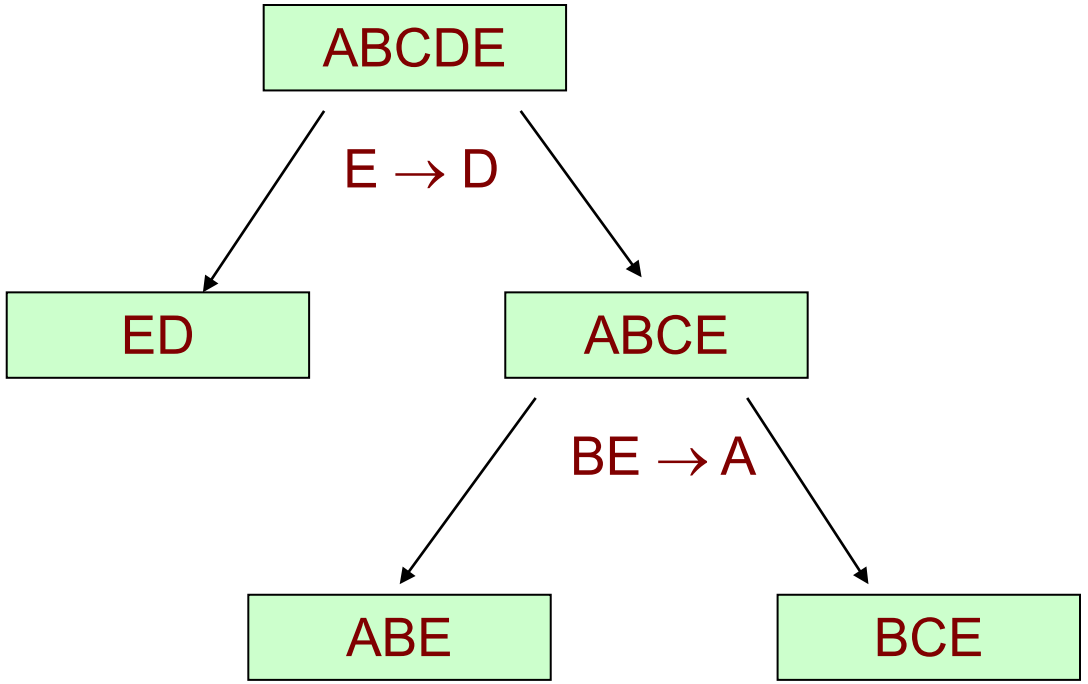
Example: BCNF Decomposition

Same example

- R = (A,B,C,D,E)
- AC → B

  E → D

  BE → A
- Key = ACE, BCE



which means you can choose any order to proceed decomposition.

Different orders of chosen FDs lead to different decompositions!

Lossless decomposition

Not Dependency preserving

BCNF Decomposition

- BCNF decomposition guarantees that the decomposition is a *lossless-join*
- It does **not** guarantees that the decomposition is *dependency preserving*

Outline

- Problems Caused by Redundancy
- Functional Dependencies (FDs)
- Normal Forms Based on Keys
- Decomposition
  - Lossless Join Decomposition
  - Dependency Preserving Decomposition
- **Normalization**
  - BCNF Decomposition Algorithm
  - **3NF Decomposition Algorithm**

3NF Definition (Review)

A relation R is in 3rd normal form if :

For every nontrivial dependency $A_1, A_2, ..., A_n \rightarrow B$

for R, $\{A_1, A_2, ..., A_n\}$ is a key for R, or B is part of a key

- Tradeoff:

  BCNF = no FD anomalies, but may lose some FDs

  3NF = keeps all FDs, but may have some anomalies

Canonical Cover

- Canonical Cover - A **minimal** and **equivalent** set of functional dependency

Two sets of functional dependencies *E* and *F* are **equivalent** if $E^+ = F^+$

- Example:

  R = (A, B, C)

  F = { A → BC, B → C, A → B, AB → C }

  F can be simplified:

  By the decomposition rule:

  A → BC ⇒ A → B and A → C

  Therefore A → B is redundant

STEP 1

Example: Canonical Cover

- R = (A, B, C)
- F = { A → BC, B → C, **A → B**, AB → C } = {A→B, A→C, B→C, A→B, AB→C}
- If A → B is redundant, then A → B can be remove from F
- Consider F' = F – {A → B} = {A → BC, B → C, AB → C} = {A→B, A→C, B→C, AB→C}
- If A → B ∈ F'$^+$, then A → B is redundant

note that **A → B** is missing

- Way 1 to show that A → B ∈ F'$^+$
  - By the decomposition rule:
  - A → BC ⇒ A → B and A → C
  - Therefore A → B ∈ F'$^+$, and A → B is redundant
- F'= { A → BC, B → C, AB → C }

STEP 1

- R = (A, B, C)
- F = { A → BC, B → C, **A → B**, AB → C }
- If **A → B** is redundant, then **A → B** can be remove from F
- Consider F' = F – {A → B} = {A → BC, B → C, AB → C}
- If A → B ∈ F'$^+$, then A → B is redundant

note that **A → B** is missing

- Way 2 to show that A → B ∈ F'$^+$
  - Compute the closure of A:

note that this implies **A → B**

    - closure = A
    - closure = ABC
    - Hence $A^+ = ABC$
    - Therefore A → B ∈ F'$^+$, and A → B is redundant

redundent

- F'= { A → BC, B → C, AB → C } ⇔ { A→B, A→C, B→C, A→C, B→C }

STEP 2

- R = (A, B, C), F' = { A → BC, B → C, **AB → C** }
- F' can be further simplified. If **B** is extraneous in A**B** → C, then **B** can be removed as A → C in F'
- If **B** is extraneous in A**B** → C, then A → B ∈ F''$^+$

  where, F'' = (F' – {AB → C }) = {A → BC, B → C}

  - Way 1 to show that A → B ∈ F''$^+$
    - From A → BC in F''
    - A → B (given)
  - By A → B ∈ F''$^+$ and AB → C ∈ F'
    - A → AB (augmentation)
    - AB → C (given in F')
    - A → C (transitivity), or B is extraneous in AB → C

IF A → B ∈ F''$^+$:
- A → B ⇔ A → AB
- AB → C ⇔ A → C

- F'' = { A → BC, B → C }

STEP 2

- R = (A, B, C), F' = { A $\rightarrow$ BC, B $\rightarrow$ C, **AB $\rightarrow$ C** }
- F' can be further simplified. If **B** is extraneous in A**B** $\rightarrow$ C, then **B** can be removed as A $\rightarrow$ C in F'
- If **B** is extraneous in A**B** $\rightarrow$ C, then A $\rightarrow$ B $\in$ F''$^+$ where, F'' = (F' – {AB $\rightarrow$ C }) = {A $\rightarrow$ BC, B $\rightarrow$ C}
- Way 2 to show that A $\rightarrow$ B $\in$ F''$^+$
  - Compute the closure of A (under F''):
    - closure = ABC ✓ (from A $\rightarrow$ BC)
- By A $\rightarrow$ AB $\in$ F''$^+$ and AB $\rightarrow$ C $\in$ F'
  - A $\rightarrow$ AB (closure)
  - AB $\rightarrow$ C (given in F')
  - A $\rightarrow$ C (transitivity), or B is extraneous in AB $\rightarrow$ C
- F'' = { A $\rightarrow$ BC, B $\rightarrow$ C }

STEP 3

- R = (A, B, C), F" = { A $\rightarrow$ B**C**, B $\rightarrow$ C }
- F' can be further simplified. If **C** is extraneous in A $\rightarrow$ B**C**, then **C** can be removed
- Consider F"' = (F" – {A $\rightarrow$ C}) = {A $\rightarrow$ B, B $\rightarrow$ C}
- If **C** is extraneous in A $\rightarrow$ B**C**, then A $\rightarrow$ BC $\in$ F"'$^+$
- Way 1 to show that A $\rightarrow$ BC $\in$ F"'$^+$
  - From A $\rightarrow$ B and B $\rightarrow$ C
  - we can deduce A $\rightarrow$ C (transitivity)
  - From A $\rightarrow$ B and A $\rightarrow$ C
  - we get A $\rightarrow$ BC (union)
  - So C is extraneous in A $\rightarrow$ BC
- F"' = { A $\rightarrow$ B, B $\rightarrow$ C }.
- This is a canonical cover for F

STEP 3

- R = (A, B, C), F" = { A $\rightarrow$ B**C**, B $\rightarrow$ C }
- F' can be further simplified. If **C** is extraneous in A $\rightarrow$ B**C**, then **C** can be removed
- Consider F"' = (F" – {A $\rightarrow$ C}) = {A $\rightarrow$ B, B $\rightarrow$ C}
- If **C** is extraneous in A $\rightarrow$ B**C**, then A $\rightarrow$ BC $\in$ F"'$^+$
- Way 2 to show that A $\rightarrow$ BC $\in$ F"'$^+$
  - Compute A^+ under F"' as follows:
    - closure = A
    - closure = AB (A $\rightarrow$ B)
    - closure = ABC (B $\rightarrow$ C)
  - So, A $\rightarrow$ BC can be deduced
- F"' = { A $\rightarrow$ B, B $\rightarrow$ C }.
- This is a canonical cover for F

Canonical Cover

- A canonical cover F_c of a set of functional dependency F must have the following properties

1. Every functional dependency $\alpha \rightarrow \beta$ in F_c contains ***no*** ***extraneous*** attributes in α (ones that can be removed from α without changing F_c^+)

   So *A* is extraneous in α if $A \in \alpha$ and

   $(F_c - \{\alpha \rightarrow \beta\}) \cup \{(\alpha\text{-}A) \rightarrow \beta\}$ logically implies F_c

Canonical Cover (cont.)

2. Every functional dependency $\alpha \rightarrow \beta$ in F_c contains no

   extraneous attributes in β (ones that can be removed from

   β without changing F_c^+)

   So A is extraneous in β if $A \in \beta$ and

   $(F_c - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F_c

3. Each left side of a functional dependency in F_c is unique.

   That is there are no two dependencies $\alpha_1 \rightarrow \beta_2$ and $\alpha_1 \rightarrow \beta_2$ in

   F_c such that $\alpha_1 = \alpha_2$

Testing if an Attribute is Extraneous

- Consider a set F of functional dependencies and the functional dependency $f = \alpha \rightarrow \beta$ ($f = \mu\mathbf{A} \rightarrow \beta$) in F.
- To test if attribute **A** $\in \alpha = \mu\mathbf{A}$ is extraneous in $\alpha = \mu\mathbf{A}$
  - *compute* $\mu^+ = (\{\alpha\} - \mathbf{A})^+$ using the dependencies in F
  - *check that* $\mu^+ = (\{\alpha\} - \mathbf{A})^+$ contains **A**; if it does, **A** is extraneous
- To test if attribute **B** $\in \beta = \nu\mathbf{B}$ is extraneous in $\beta = \nu\mathbf{B}$
  - *compute* α^+ using only the dependencies in

$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - \mathbf{B})\}$$

$$= (F - \{f\}) \cup \{\alpha \rightarrow \nu\}$$

  - *check that* α^+ contains **B**; if it does, **B** is extraneous

Testing if $A \in \alpha$ is Extraneous

- If attribute $A \in \alpha = \mu A$ is extraneous in α, consider F = $\{ f_1, \ldots, f, \ldots, f_n \}$, and $f = \alpha \rightarrow \beta = \mu A \rightarrow \beta$
  - If $\mu \rightarrow A$ under F – $\{f\}$ and $f = \mu A \rightarrow \beta$, then by the transitivity:

    $$\mu \rightarrow A,\ \mu \rightarrow \mu A \quad \textit{and}$$

    $$\mu A \rightarrow \beta \Rightarrow \mu \rightarrow \beta$$

  - Then we can use $\mu \rightarrow \beta$ in F insert of $\mu A \rightarrow \beta$ (**A** is extraneous)
- Testing $\mu \rightarrow A$ under F – $\{f\}$ $\Leftrightarrow$ Testing $\mu \rightarrow A$ under F
  - *compute* μ^+ using the dependencies in F
  - *check that* μ^+ contains **A**; if it does, **A** is extraneous

Testing if $B \in \beta$ is Extraneous

- If attribute $B \in \beta = \nu B$ is extraneous in β, consider F = $\{ f_1, \ldots, f, \ldots, f_n \}$, and $f = \alpha \rightarrow \beta = \alpha \rightarrow \nu B$
  - If $\alpha \rightarrow B$ under F' = $(F - \{f\}) \cup \{ \alpha \rightarrow \nu \}$ then by the augmentation:

    $$\alpha \rightarrow B \text{ and } \alpha \rightarrow \nu \Rightarrow \alpha \rightarrow \nu B$$

  - Then we can remove $\alpha \rightarrow B$ in F, using $\alpha \rightarrow \nu$ in F insert of $\alpha \rightarrow \nu B$ (B is extraneous)
- Testing $\alpha \rightarrow B$ (under F')
  - *compute* $(\alpha)^+$ using the dependencies in F'
  - *check that* $(\alpha)^+$ contains B; if it does, B is extraneous

Algorithm: Compute a Canonical Cover

- compute a canonical cover for F

repeat

replace any $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$

by $\alpha_1 \rightarrow \beta_1 \beta_2$

delete any **extraneous attribute**

from any $\alpha \rightarrow \beta$

until F does not change

Example: Canonical Cover

- Given: F = {E→A, E→B, A→BC, B→C}
- Canonical Cover F_c :
  - F = {E→A, E→B, A→BC, B→C}
  - F = {E→AB, A→BC, B→C}
  - F = {E→AB, A→BC, B→C}
  - F = {E→AB, A→B, B→C}
- So, F_c = {E→A, A→B, B→C}

A→BC is **extraneous**

From A→B and B→C
we deduce A→C (transitivity)
From A→B and A→C
we deduce A→BC (union)

Algorithm: 3NF Decomposition

find a canonical cover F_c for F

result = { }

for each $\alpha \rightarrow \beta$ in F_c do

  if no schema in result contains $\alpha\beta$ then

    add schema $\alpha\beta$ to result

end for

if no schema in result contains a candidate key for R

  choose any candidate key α for R

  add schema α to result (*ensure the lossless-join*)

end if

Example: 3NF Decomposition

- R = (A,B,C,D,E,F,G)
- F = { $A \rightarrow B$, $A \rightarrow C$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

$DF^+ = DFEBGAC = R$

$F_c = \{A \rightarrow B, A \rightarrow C, D \rightarrow E, B \rightarrow A, F \rightarrow BG\}$

$= \{A \rightarrow B, A \rightarrow C, D \rightarrow E, B \rightarrow A, F \rightarrow B, F \rightarrow G\}$

$= \{A \rightarrow BC, D \rightarrow E, B \rightarrow A, F \rightarrow BG\}$

Example: 3NF Decomposition (Step1)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

F_c	result
$A \rightarrow BC$	
$D \rightarrow E$	
$B \rightarrow A$	
$F \rightarrow BG$	

Example: 3NF Decomposition (Step2)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

$R = (A,B,C,D,E,F,G)$ → $R_1 = (A,B,C)$, $R_2 = (A,D,E,F,G)$

F_c	result
$A \rightarrow BC$	ABC
$D \rightarrow E$	
$B \rightarrow A$	
$F \rightarrow BG$	

Example: 3NF Decomposition (Step3)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

$R_2 = (A, D, E, F, G)$ < $R_3 = (D, E)$, $R_4 = (A, D, F, G)$

F_c	result
$A \rightarrow BC$	ABC
$D \rightarrow E$	DE
$B \rightarrow A$	
$F \rightarrow BG$	

Example: 3NF Decomposition (Step4)

- R = (A,B,C,D,E,F,G)
- F_c = { A $\rightarrow$ BC, D $\rightarrow$ E, B $\rightarrow$ A, F $\rightarrow$ BG }
- Candidate key = DF

Since relation (A,B,C) has included f = {B→A}

F_c	result
A $\rightarrow$ BC	ABC
D $\rightarrow$ E	DE
B $\rightarrow$ A	
F $\rightarrow$ BG	

Example: 3NF Decomposition (Step5)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

F_c	result
$A \rightarrow BC$	ABC
$D \rightarrow E$	DE
$B \rightarrow A$	
$F \rightarrow BG$	

Example: 3NF Decomposition (Step6)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF

F_c	result
$A \rightarrow BC$	ABC
$D \rightarrow E$	DE
$B \rightarrow A$	
$F \rightarrow BG$	FBG

Example: 3NF Decomposition (Step7)

- R = (A,B,C,D,E,F,G)
- F_c = { A $\rightarrow$ BC, D $\rightarrow$ E, B $\rightarrow$ A, F $\rightarrow$ BG }
- Candidate key = DF
- Result = {ABC, DE, FBG}

F_c	result
A $\rightarrow$ BC	ABC
D $\rightarrow$ E	DE
B $\rightarrow$ A	
F $\rightarrow$ BG	FBG

Example: 3NF Decomposition (Step8)

- R = (A,B,C,D,E,F,G)
- F_c = { $A \rightarrow BC$, $D \rightarrow E$, $B \rightarrow A$, $F \rightarrow BG$ }
- Candidate key = DF
- Result = {ABC, DE, FBG, DF}

R_1 R_2 R_3 R_4

$R_2 \bowtie R_4 \bowtie R_3 \bowtie R_1 = R$

F_c	result
$A \rightarrow BC$	ABC
$D \rightarrow E$	DE
$B \rightarrow A$	
$F \rightarrow BG$	FBG

Example: 3NF Decomposition

- R = (student_id, student_name, course_id, course_name)
- F = { student_id $\rightarrow$ student_name,
- course_id $\rightarrow$ course_name }
- Candidate key = { student_id, course_id }
- 3NF Decomposition:

$F_c = F$

R_1 = (student_id, student_name)

R_2 = (course_id, course_name)

R_3 = (student_id, course_id)

Example: 3NF Decomposition

- R = (A, B, C)
- F = { $A \rightarrow BC$, $B \rightarrow C$ }
- Candidate key = { A }
- R is not in 3NF
- 3NF Decomposition:

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

  R_1 = (A, B)

  R_2 = (B, C)

3NF Decomposition

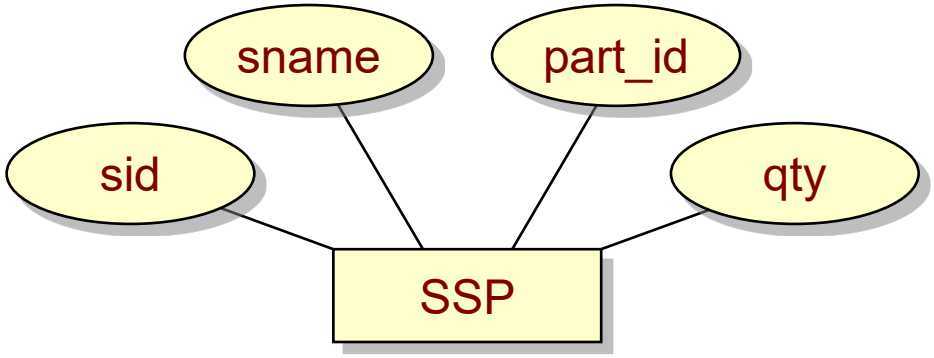
- 3NF decomposition guarantees that the decomposition is a *lossless-join*
- It guarantees that the decomposition is *dependency preserving*

BCNF vs. 3NF

- Always possible to decompose a relation into relations in 3NF and
  - the decomposition is lossless
  - dependencies are preserved
- Always possible to decompose a relation into relations in BCNF and
  - the decomposition is lossless
  - may not be possible to preserve dependencies

More Examples

- Candidate Keys:
  - (sid, part_id) and (sname, part_id)
- FDs:
  - { sid, part_id } → qty
  - { sname, part_id } → qty
  - sid → sname
  - sname → sid



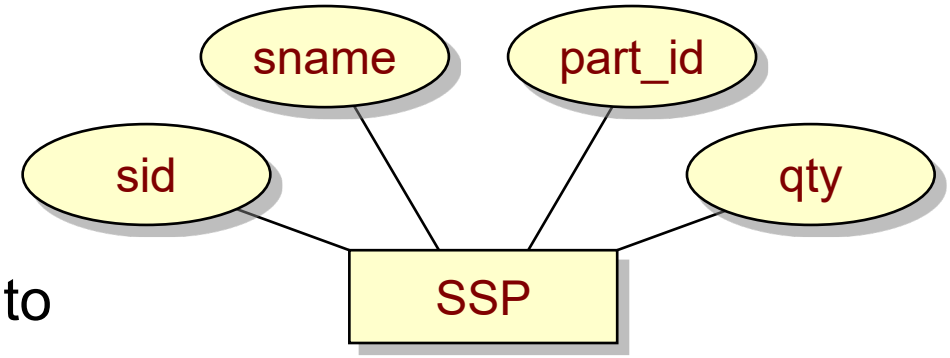
- The relation is in 3NF:

  For sid → sname, … sname is in a candidate key

  For sname → sid, … sid is in a candidate key

- However, this leads to redundancy and loss of information

More Examples

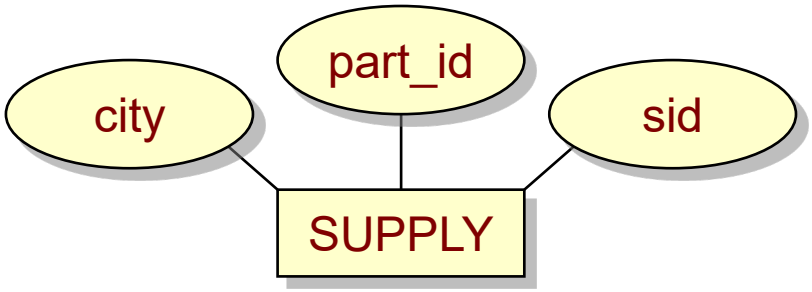


- If we decompose the schema into
  - R_1 = (sid, sname), R_2 = (sid, part_id, qty)
- These are in BCNF
- The decomposition is dependency preserving
  - { *sname*, *part_id* } $\rightarrow$ *qty* can be deduced from

    1. sname $\rightarrow$ sid (given)
    2. { sname, part_id } $\rightarrow$ { sid, part_id } (augmentation on (1))
    3. { sid, part_id } $\rightarrow$ qty (given)

  - And finally transitivity on (2) and (3)

More Examples



SUPPLY | city | part_id | sid

- At a city, for a certain part, the supplier is unique:
  - { city, part_id } $\rightarrow$ sid
  - Also, sid $\rightarrow$ city
- The relation is not in BCNF:
  - sid $\rightarrow$ city is not trivial, and sid is not a superkey
- It is in 3NF:
  - sid $\rightarrow$ city, city is in the candidate key of { city, part_id }
- If we decompose into (sid, city) and (sid, part_id) we have BCNF, however { city, part_id } $\rightarrow$ sid will not be preserved

Design Goals

- Goal for a relational database design is:
  - BCNF
  - Lossless join
  - Dependency preservation
- If we cannot achieve this, we accept:
  - 3NF
  - Lossless join
  - Dependency preservation